

Table of Contents

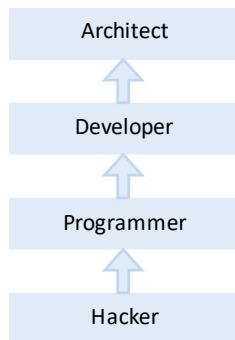
Chapter 00: Software Design.....	1
Unit 0: Algorithm Design	14
Chapter 01 Tool: Flowchart	15
Chapter 02 Tool: Pseudocode	42
Chapter 03 Metric: Efficiency	61
Chapter 04 Metric: Maintainability.....	90
Chapter 05 Quality: Assert.....	114
Chapter 06 Quality: Trace.....	136
Chapter 07 Strategy: Decisions	158
Chapter 08 Strategy: Collections.....	178
Chapter 09 Strategy: Loops	198
Unit 1: Modularization Design.....	223
Chapter 10 Tool: Structure Chart.....	224
Chapter 11 Tool: Data Flow Diagram	241
Chapter 12 Metric: Cohesion	261
Chapter 13 Metric: Coupling	278
Chapter 14 Quality: Test Case	298
Chapter 15 Quality: Driver	315
Chapter 16 Strategy: Recursion.....	331
Chapter 17 Strategy: Top-Down.....	350
Chapter 18 Strategy: Bottom-Up	369
Chapter 19 Strategy: Functional.....	389
Unit 2: Encapsulation Design.....	407
Chapter 20 Tool: Class Diagram I	408
Chapter 21 Metric: Fidelity.....	426
Chapter 22 Metric: Robustness.....	440
Chapter 23 Metric: Convenience	461
Chapter 24 Metric: Abstraction.....	477
Chapter 25 Quality: Unit Test	492
Chapter 26 Quality: Test-Driven.....	516
Chapter 27 Strategy: Noun Identification.....	538
Chapter 28 Strategy: Metaphor	559
Chapter 29 Strategy: Data Protection.....	577
Unit 3: Class Relation Design	593
Chapter 30 Tool: Class Diagram II	594
Chapter 31 Metric: Adaptability.....	615
Chapter 32 Metric: Alignment.....	636
Chapter 33 Metric: Redundancy	652
Chapter 34 Quality: Debugger	668
Chapter 35 Strategy: Polymorphism.....	691
Chapter 36 Strategy: <i>Is-a</i> and <i>Has-a</i>	710
Chapter 37 Strategy: Object Creation.....	727
Chapter 38 Strategy: Algorithm Abstraction	751
Chapter 39 Strategy: State	771
Unit 4: Component & System Design	797
Chapter 40 Tool: Component Diagram	798
Chapter 41 Tool: Design Description	814
Chapter 42 Quality: V-Model.....	834
Chapter 43 Strategy: Large Inheritance Trees	854
Chapter 44 Strategy: Message Passing	868
Chapter 45 Strategy: Separation of Concerns	895
Chapter 46 Strategy: Command Passing.....	914
Chapter 47 Strategy: Interfaces	943
Chapter 48 Strategy: Tokens and Entities.....	965
Chapter 49 Strategy: Layered System Design.....	988
Appendix	1010

Chapter 00 Software Design

Software design is the process of helping programmers become software developers and eventually software architects using the tools which help to visualize, measure, and build high-quality designs.

Computer code (or simply “code” for short) is a set of instructions understood by a computer to perform a task. Code could be machine language (the 0s and 1s that are stored directly in memory and understood by a CPU), or in a higher-level language such as Python, C++, or Java. One would think that we would call an individual who writes code a “coder.” For some reason we don’t. That label is a “computer programmer,” or “programmer” for short.

A programmer is one who writes code



We call an individual who writes software a “software developer” (or “developer”). What, then, is the difference between a programmer and a software developer? For some, there is no difference. For others, the implication exists that the developer has a more holistic view of the entire software system and makes more informed decisions as to what is best for the system. In other words, the developer is a better designer than the programmer. Some may even take this one step further and equate programming with “hacking;” throwing code together for the purpose of achieving a mostly workable solution. However, most would rank a hacker as somewhat below a programmer who, in turn, would defer to a software developer for more complex or larger-scale problems.

A software developer is one who consistently makes high-quality software design decisions

Who, then, formulates the grand design for truly large and unimaginably complex systems? For this level, we have yet another label: architect. A “software architect” (or simply “architect” for short) is the highest level of the coding hierarchy. It is the architect who would lay out the overall system design for a new application. It is the architect who would be consulted when large changes are contemplated for existing systems. It is the architect who would be called upon to solve the complex and mission-critical algorithms. In other words, the architect is the most valuable (and thus most highly paid) member of the coding team.

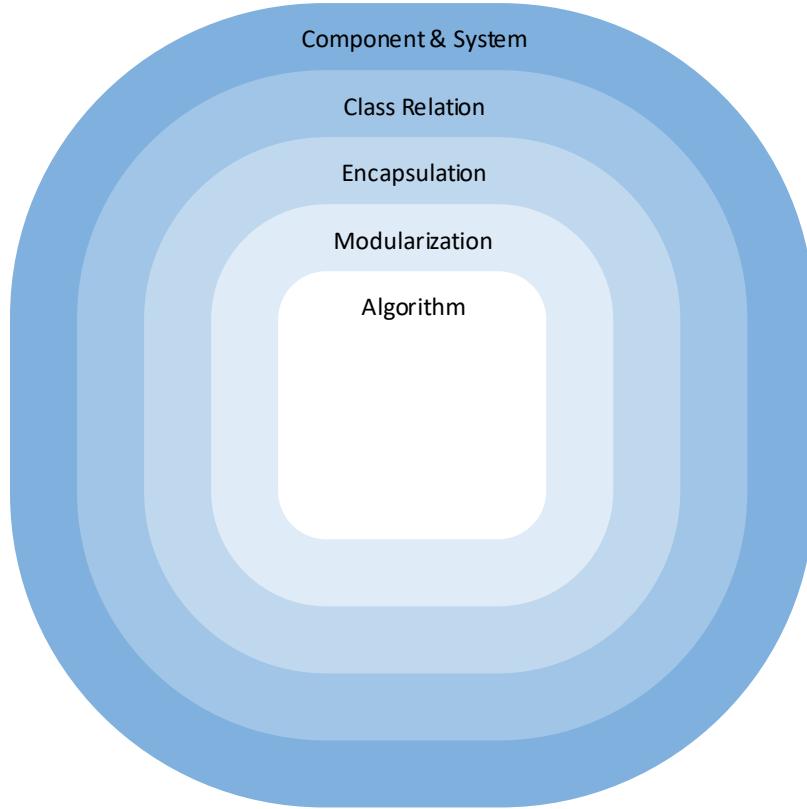
A software architect is an experienced and wise software developer responsible for system-wide design decisions

How does one move up the ladder from a programmer to an architect? On the surface, the answer may seem to be experience. However, many programmers never progress to the software developer level no matter how long they have been writing code. Clearly there is another ingredient. Fundamentally, architects can be distinguished from programmers because they approach problems differently. This book is about adopting the architect’s mentality to software design.

Levels of Design

Designing a software application is much like designing a car. A car is assembled from a collection components (like the engine) which is built from a collection of parts (such as the fuel pump) which is created from individual pieces (such as a bolt) that is comprised of aspects (like the threading on the bolt). At each level of design, a different set of tools and principles are required. An engineer who is very good at designing the fuel flow in the engine might not be so good at selecting the best metallurgical properties of a bolt. Certainly, a different set of tools is required to design at each of these levels.

Just as there are several levels of car design, there are five levels of software design. The levels of software design corresponding to the software tools used at each level.



These levels present different challenges and require a unique approach. The five levels of design are the following:

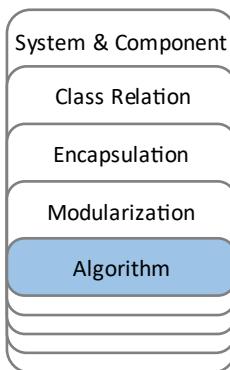
Software Tools	Design Level
Statements	Algorithm design
Functions	Modularization design
Classes	Encapsulation design
Groups of classes	Class relation design
Components and applications	Component and system design

Most architects and software developers are proficient at designing software at a variety of levels: from overall system design down to bit twiddling.

Statements

In the early days of software development, all coding decisions were made at the smallest imaginable level: machine language. Programmers would work with memory locations, registers, and individual instructions. Over time, several techniques emerged which facilitated the code writing process, such as co-locating code relating to specific actions, using overlays to work in memory-constrained environments, and using conditional jumps to loop through elements in a collection. In those early days, there was no operating system support for these tasks (there often were not even operating systems) and high-level programming languages were not yet invented. The first generation of computer languages promised to simplify the programming process by introducing language constructs representing many common programmer tasks. These programming constructs are called statements. The first of these to be commercially available was FORTRAN in 1954 followed by many more in the next decade. Though some programmers still work at the machine language or assembly level, most programmers work with statements: variables, conditionals, loops, and arrays.

A statement is a unit of software representing an action to be carried out



Algorithm Design

Algorithms are defined as a sequence of statements designed to accomplish some task. When we talk about algorithm design, we are primarily focused on what goes on inside a function, though some algorithms necessarily work between functions. In other words, we will define an algorithm as a collection of programming language statements. Algorithm design is the process of effectively utilizing statements to find correct, efficient, adaptable, and robust solutions to programming problems. It is the core to all programming activities. Though some programmers may prefer to design with functions or classes or components, the majority of the programmer's time is spent working with statements.

An algorithm is a sequence of statements designed to accomplish some task

Terms and Concepts

Terms and concepts associated with statements and algorithm design are as follows.

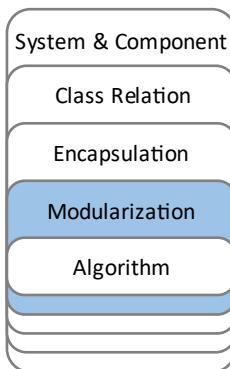
Term	Definition
Machine code	A statement consisting of 1s and 0s that a computer's CPU consumes directly. Generally, not human readable.
Assembly code	A human-readable version of machine code consisting of register designations and memory locations.
Statement	A high-level programming language construct consisting of one or more assembly code elements.
Execute	The process of carrying out one or more statements.
Variable	A named location in memory where data are stored.
Conditions	A type of statement representing a decision.
Loop	A type of statement allowing for a collection of statements to be executed more than once.
Array	A collection of variables accessible by index.
Algorithm	A sequence of statements designed to accomplish a task.

Functions

In the late 1960s, high-level languages started including programming features called procedures and functions. Before this time, code was informally delimited with labels and GOTO statements. It may seem hard to believe today, but the introduction of functions was not universally greeted with enthusiasm. In fact, when Larry Constantine first proposed the idea of modularization (originally called “structured programming”), some wondered whether it would place undue restrictions on the creative process. This pushback was fortunately short-lived and by the early 1970s virtually all programming languages had functions.

A function is a unit of a program containing a collection of statements. There is only one way into a function but there may be more than one way to leave it. One function can invoke another function through the “calling” process. Unless the programming language introduces limitations on how functions are called (using namespaces, packages, or classes), any function can call any other function. This means a function can even call itself through a process called recursion. Since a function’s statements are exactly like all the other statements in the program, the principles of algorithm design apply to all that occurs within a function.

A function is a unit of a program containing a collection of statements



Modularization Design

Modularization is the process of subdividing programs into distinct chunks for the purpose of reusing code, simplifying design, and reducing development costs. When we talk about modularization design, we are primarily focused on how to effectively use functions to solve problems, with the understanding that functions consist of algorithms and may be part of a class.

Modularization is the process of subdividing a program into functions

Terms and Concepts

Terms and concepts associated with functions and modularization design are as follows.

Term	Definition
Function	A named collection of statements to perform a task. Also known as a procedure, sub program, or subroutine.
Invoke	The process of executing a given function.
Caller	The entity invoking a given function.
Callee	A function that is invoked by a caller.
Parameter	A datum sent into a function from the caller. This is also known as the function’s input.
Return	The process of sending data out of a function back to the caller.
Modularization	The practice of subdividing a program into functions.

Classes

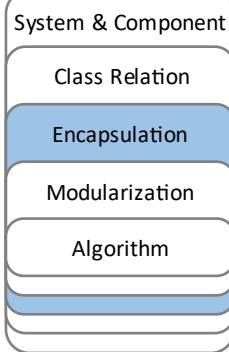
A class is a program entity consisting of data and the operations that act on them. Though the concept of classes and some of the early thinking of object-oriented programming can be traced back to the 1960s (with Simula 67), it was 1981 when Smalltalk was introduced to the larger programming community. As with functions two decades before, this level of abstraction was not universally met with enthusiasm. Some viewed classes as unnecessary and continued to make do with functions and statements. Others whole-heartedly embraced these new constructs; some programming languages even required all code to be contained in classes. By the mid 1990s, classes were widely adopted into the most widely used languages. Today, most modern languages today allow programs to consist of classes, standalone functions, and even statements in the global scope.

A class is a program entity consisting of data and the operations that act on them

A class' data are called attributes or member variables. They are variables declared and used in much the same way that variables are declared and used elsewhere in a program. The main difference is that they are associated directly with a class.

A class' operations are types of functions. They can be called member functions or methods. Methods are like every other function in the system except they are associated directly with a class and have access to the class' attributes. Since methods are just functions, all the principles of modularization design apply to them.

Encapsulation Design



Encapsulation is the process of creating classes to capture components of the system design. Some use the phrase "object-oriented design" to mean the same thing, but there is a subtle difference. Encapsulation design refers to the process of designing a single class, whereas object-oriented design also includes the process of utilizing multiple related classes to capture a single design concern. The goal of encapsulation design is to effectively use classes to simplify code reuse, decrease program complexity, decrease development time, and increase quality.

Encapsulation is the process of creating classes to capture components of the system design

Terms and Concepts

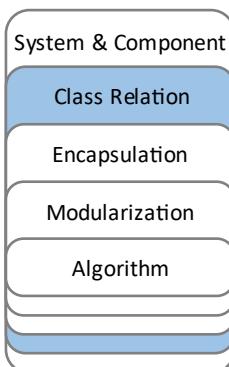
Terms and concepts associated with classes and encapsulation design are the following.

Term	Definition
Class	A programming construct consisting of data and the operations that act on them.
Attribute	The data of a class, also known as a member variable.
Operation	A function associated with a class, also known as a member function or method.
Object	An instantiated class. This is a variable whose data type is a class.
Instantiation	The process of creating an object.
Object-oriented	A design philosophy centered on designing programs with classes.
Encapsulation	The process of creating classes to capture component of the system design.

Groups of Classes

The very first theoretical writings about classes and the very first languages supporting classes allowed for grouping of related classes. There are several types of groupings that are possible in most languages that support classes:

- A class can be contained within another class. This means that a class can have an object as a member variable.
- A class' method can use another class. This means that a member function can have a local variable that is of the type of another class.
- A class can be defined in terms of another class. This means that one class can be an extension of or a special type of another class.



Class Relation Design

The process of grouping related classes together to better represent a design concern is called class relation design. This is done by collecting, arranging, and manipulating groups of classes. When done correctly, the resulting code has less redundancy, is easier to make adjustments and augmentation, and the overall program is easier to understand. The primary tools used in class relation design are composition, inheritance, and polymorphism. Since class relations are built from individual classes, class relation design builds from encapsulation design.

Terms and Concepts

Terms and concepts associated with groups of classes and class relation design are as follows.

Term	Definition
Composition	When the member variable of one class is an object of another class.
Association	When the member variable of one class is a reference to an object of another class.
Nested class	A class defined within another class.
Inheritance	When one class is defined in terms of another class.
Base class	A class that others inherit from. It is the basis of inheritance.
Derived class	A class that inherits from another base class.
Extension	A type of inheritance where one class assumes all the properties and methods of another class and adds a few.
Polymorphism	An inheritance technique where various derived classes have different implementations but the same interfaces.

Component & System

A system is a holistic view of the entire application or software package under development. Often a system is large enough that it makes sense to subdivide it into several semi-autonomous subsystems called components.

The goal of every software engineer is to produce a working system. A working system is defined as a collection of software deliverables that fulfills the client's needs. In one situation, the term "system" might refer to a single application a student submits to a teacher to fulfill an assignment. In another situation, the term "system" might refer to an e-commerce store constituting dozens of applications running on thousands of computers performing millions of transactions a second. In most cases, the client is only concerned with the system. How the system is organized and built is less important than it meets the client's definition of quality: it works as expected, consumes a reasonable amount of resources, can be adjusted to the client's changing needs, was developed in a timely matter, and does not cost more than it should.

A system is a collection of software deliverables that meet the client's needs

A component is a part of the system. In most cases, it is a distinct partition of the system that can be developed, substituted, and even executed independently of the other components within the system. While many simple systems consist of a single component, most large applications have several. There is no formal definition of a component. In some cases, two components in the same system can be implemented with different technology, execute independently on different hardware, and communicate through asynchronous protocols. In other cases, two components can execute using the same technology, execute on the same CPU in the same process space, and communicate through traditional function calls. Finally, some may even consider mostly autonomous blocks of source code in the same file to be distinct components.

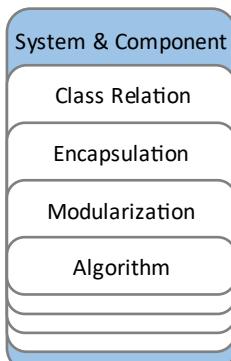
A component is a distinct part of a system that can be developed, substituted, and even executed independently of the rest of the system

Component- and System-Level Design

System and component design are grouped together because we use the same design tools to visualize them, assess the quality of designs with the same metrics, use the same techniques to find and avoid bugs, and use many of the same strategies to design them.

Terms and Concepts

Terms and concepts associated with groups of classes and class relation design are as follows.

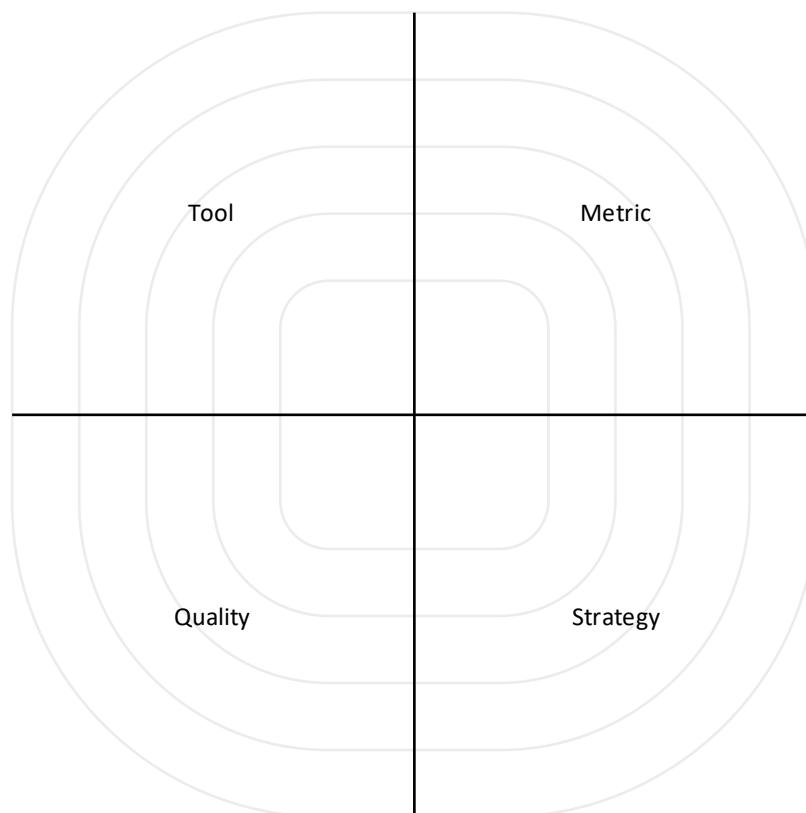


Term	Definition
System	The entire application developed for the client.
Component	A distinct part of the system.
Interface	A method of communication between components.
Protocol	The language utilized in an interface.

Components of Design

The difference between a novice programmer and a skilled engineer is not just talent and experience, but also the mastery of the various components of software design. These components span all the levels of software development, though how they are manifested is often highly dependent on the design level. To illustrate this point, recall our car analogy. The tools used to visualize a bolt on a car are quite different from the tools used to visualize the wiring, software, or cooling system. At each level of design, the mechanical engineer uses tools specifically created to work at that level. Similarly, the metrics used to determine the quality of a bolt are quite different than those used to determine the quality of a fuel pump or the radiator.

At each level of design, there are four categories of tools and techniques that need to be mastered: design visualization tools (called “tool”), quality metrics (called “metric”), defect identification techniques (called “quality”), and best practices or techniques (called “strategy”).



At each level of design, there can be any number of tools, metrics, quality assurance techniques, and strategies. For example, at the modularization level, there are two visualization tools (structure chart and data flow diagram), two metrics (cohesion and coupling), two quality assurance techniques (test cases and drivers), and four strategies (recursion, top-down, bottom-up, and functional programming).



Design Visualization Tools

Design visualization tools help developers create, visualize, and communicate design ideas. Though almost every programming language is textual (built from words), most design visualization tools are graphical in nature. Typically, symbols are used to represent program entities and lines are used to represent relationships.

There are a very large number of different design visualization tools. Some are widely used (such as flowcharts and class diagrams), others are rather obscure (such as object diagrams and deployment diagrams). Just like a master mechanic has many tools in his or her toolbox and knows how to use each, a master software designer strives to always use the right tool for the job. The most useful design visualization tools are:

Tool	Use
Flowchart	A graphical tool used to represent algorithms, flowcharts are particularly good for communicating designs with non-technical people and for visualizing complex decisions.
Pseudocode	A textual tool used to represent algorithms. Pseudocode can be very close to a programming language. This makes it easy to translate a design into working code.
Structure Chart	A graphical tool used to represent modularization decisions. A structure chart represents functions and how they call each other.
Data Flow Diagram	A graphical tool used to represent how data flows through a program. Though it is particularly useful at the function level, a data flow diagram (DFD) can also be useful for representing system designs or even the flow of data within a single function.
Class Diagram	A graphical tool used to represent the attributes and operations associated with a class. It is also used to describe how multiple classes relate to each other. The class diagram is perhaps the most widely used design visualization tool.
Component Diagram	A graphical tool used to represent the components of a system and the protocols they use to communicate with each other. It is particularly useful for large-scale system visualization.
Design Document	A mixture between a graphical and a textual tool, a design document can be used to completely represent a system design, from the highest possible level down to individual statements. Design documents are aggregate tools, meaning they are a fusion of several visualization tools.

In addition to these visualization tools, there are many others that are useful in a variety of situations. These include entity relationship diagrams (for representing database design), sequence diagrams (for representing concurrency challenges), timing diagrams (to coordinate asynchronous events), state machines (for untangling complex states or modality), use cases (for understanding how external entities interface with the system), and many others.

Many philosophers suggest that the language a person speaks has an important role in the complexity of a thought they can articulate. If this is true, then the more expressive our design tools, the better we can solve complex software problems.

Metrics



Metrics are rulers allowing us to measure the quality of a design idea. They help us quantify designs so we can better choose the best alternative. Perhaps this is best explained by example.

Consider three software designers arguing about how to best solve a problem. Each has a unique approach to the problem and believes that their solution is the best. The question is: whose solution should be used? All too often, the answer is: “the one who shouts the loudest” or “the one who has the most political clout” or even “the most persistent.” None of these answers are particularly satisfying. It should be that the best solution wins. In order to do this, we need to have a well-defined characterization of what is necessary for something to be “best.”

Quality metrics are universally understood and proven rulers that allow software designers make informed decisions as to which approach is best. They also help us find defects with designs by pointing out how a given design may fall short. Well-designed metrics have four properties: valid (they give insight on that which they are designed to measure), reliable (two different designers are likely to reach the same conclusion), understandable (easy to learn, understand, and explain to others), and instructive (suggest an obvious way in which the design can be improved). The most useful software metrics are the following:

Metric	Use
Efficiency	How algorithm execution time is related to input size
Maintainability	How much time or effort is required to fix defects or make enhancements
Cohesion	How well a unit of software represents one concept or performs one task
Coupling	The complexity of the interface between units of software
Fidelity	The suitability of a class in representing a design concern
Robustness	The degree of resistance a class has from being placed in an invalid state or sending invalid data to the client
Convenience	How easy it is for a client to use a class in an application
Abstraction	The amount the client needs to know about the implementation details of a class in order to use it effectively
Adaptability	The suitability of the base classes in an inheritance hierarchy to facilitate the creation of derived classes which can fulfill previously unknown or unanticipated design concerns
Alignment	The degree in which the inheritance tree models relationships in the problem domain
Redundancy	The extent in which there are common elements in an inheritance tree



Defect Identification Techniques

Defect identification techniques, collectively called “quality” for short, are a collection of tools and techniques that help a designer avoid, find, and eliminate bugs. Until the past dozen or so years, quality was not a topic for software developers. Instead, testers or quality assurance engineers would handle this issue. Developers would complete a project and “throw it over the fence” with the expectation that the testers would miraculously identify every defect in the system. Today we recognize that quality needs to be integrated into all aspects of the software development process, from problem identification to system deployment. It remains one of the fastest-evolving fields of study within the software engineering discipline to this day. Nevertheless, there are several quality tools and techniques which are widely utilized in industry that have proven to be effective in avoiding, identifying, and fixing defects:

Quality	Use
Asserts	Checks placed in a program representing an assumption the developer thinks are always true
Trace	The process of systematically stepping through the code, one statement at a time, to track the values of the variables
Test Cases	A set of actions used to verify a product, feature, or module behaves as expected
Drivers	A simple program designed to test another function or program by generating input parameters and validating output parameters
Unit Test	White box developer-written automation used to validate a single function or class in isolation of the rest of the program
Test-Driven Development	A process where tests are written before the corresponding production code is written to minimize the lag between bug introduction and removal
Debuggers	A tool that runs a program so the developer can gain insight how the program is functioning
V-Table	A method to incorporate quality assurance planning into the product development process

Quality needs to be the focus of the entire development team. Traditionally, quality initiatives are driven by the test engineers or quality-assurance engineers. A wise software developer appreciates the quality needs of the project, respects the contribution of the testing and quality assurance members of the team, and seeks opportunities to improve quality at every phase of the project.



Best Practices, Techniques, and Strategies

At every level of the design process, there exists a collection of best practices, proven techniques, and problem-solving strategies. In the software design literature, strategies go by many names: design patterns, heuristics, tips, considerations, and best practices. The best strategies provide an insight into the design process.

Though software engineering is a younger discipline, we can benefit from those who have gone before us. Bernard of Chartres, a French philosopher, stated:

We see more and farther than our predecessors, not because we have keener vision or greater height, but because we are lifted up and borne aloft on their gigantic stature.

Sir Isaac Newton put it more succinctly:

If I have seen further it is by standing on the shoulders of giants.

This basically means that a humble software developer can outperform the pioneering giants who preceded us if we apply their discoveries to our practices. These strategies can be categorized into several parts:

Strategy	Use
Decisions	Adopting best practices for one-case, two-cases, several cases, and many-case decisions
Collections	Choosing the best collection strategy and using it effectively
Loops	The different flavors of loops
Recursion	How to design a function that calls itself intentionally
Top-Down	The process of specifying the entire system and then dividing them into successively smaller subcomponents
Bottom-Up	The process of writing code beginning with the smallest and simplest components, gradually working to the most complex and interconnected ones
Functional Programming	Making every function's behavior solely dependent on its input, completely independent of system state
Noun Identification	Finding classes through a careful study of the requirements
Metaphors	Using familiar metaphors to make programs easier to understand
Data Protection	Maximizing robustness by offering assurances that an object is always in a valid state
Polymorphism	The process of one class having more than one variation, each of which honors the same contract but has a different behavior
Is-A and Has-A	The process of choosing the best class relation model: inheritance or composition
Object Creation	Where a derived class object is created to satisfy a client's request. This includes the factory, abstract factory, builder, prototype, and singleton
Algorithm Abstraction	The process of reducing complicated algorithms to their components. This includes the strategy, template method, and decorator design patterns
State	Many complex software problems can be simplified by turning them into state machines

Inheritance Trees	Techniques for working with large inheritance trees
Message Passing	Design patterns that simplify coupling by channeling messages through standard interfaces. This includes the mediator, chain of responsibility, observer, and visitor
Separation of Concerns	The process of dividing the three responsibilities of a class (store data, implement public interfaces, and perform business logic) into separate classes
Command Passing	Design patterns that simplify passing orders between components. This includes delegate invocation, encapsulated invocation, and the interpreter
Interfaces	Design patterns to simplify coupling between components. This includes the adapter, bridge, façade, and proxy design patterns
Tokens and Entities	A behavioral design pattern where all the information necessary to represent a state is encapsulated into a single object
Layered System Design	A collection of strategies for organizing distributed components

As a final note, there is no sure-fire technique to solve all imaginable problems. However, there are a few considerations that, if we keep them in mind, can make our software much better.

Organization

This text is organized in five units, each unit correspond to a level of design:

Unit	Topic
Unit 0	Algorithm design – that which goes on within a single function
Unit 1	Modularization design – the process of designing functions and how they interact with each other
Unit 2	Encapsulation design – the process of designing individual classes
Unit 3	Class relation design – the process of designing program entities consisting of multiple classes
Unit 4	Component and System design – designing entire applications or significant parts thereof.

Each unit will consist of approximately ten chapters. The chapters will be one of four categories, each corresponding to a component of design:

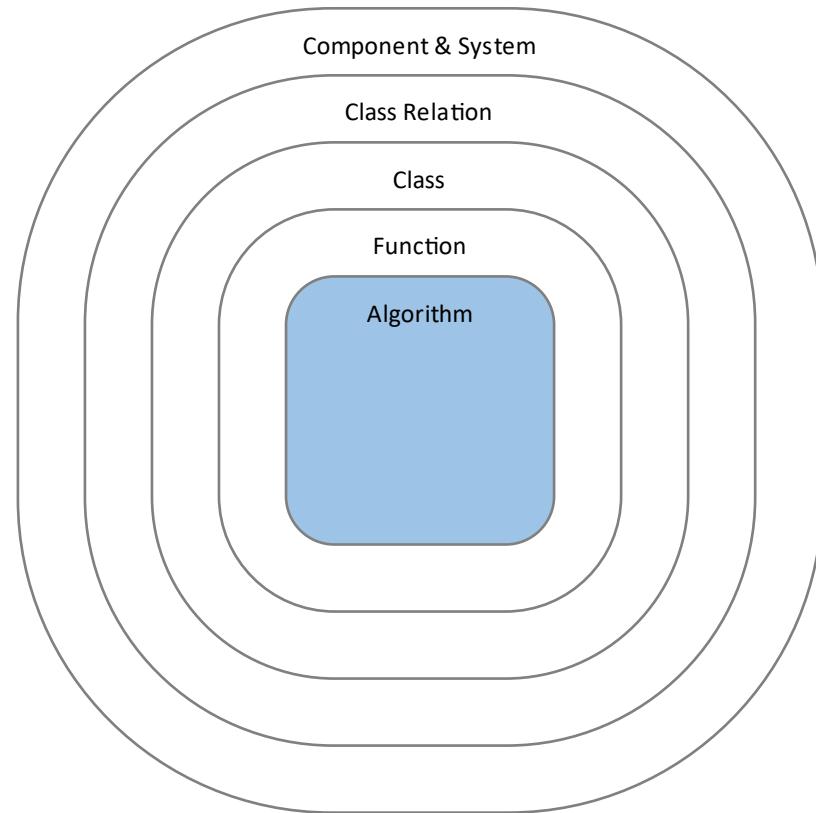
Chapters	Topic
Tool	Design representation tools
Metric	Ways to measure the quality of a design
Quality	Ways to build quality into the system
Strategy	Techniques for designing high-quality software

Every chapter will have several examples, describing how to apply the principles discussed therein to real-world problems. There will also be exercises: fact- and comprehension-based questions to help the reader access his or her understanding. There will be problems which will require some thought but should be relatively straightforward derivations of examples. Finally, there will be several challenges, designed to require significant creativity or take a significant amount of time.

Algorithm Design

An algorithm is a process or set of steps to be followed to solve a problem. An algorithm can be trivial involving only a small number of steps, or it can also be vast and complex. To say something is an algorithm makes no assumptions about the size or complexity of the task.

In the context of software design, we will use the term “algorithm” to describe everything that happens inside a function. Tools that are traditionally at the algorithm designer’s disposal include variables, expressions, conditionals, collections, and loops.



Flowchart

Chapter 01

The flowchart is a graphical tool describing program flow, facilitating both designing algorithms and communicating how algorithms work to others.

Writing code is hard. This difficulty comes from many sources, not the least of which is the need to distill abstract design concepts into the rigid syntax of a programming language. How can one be expected to come up with grandiose ideas when we are bogged down with the minutiae of details that programming languages demand? It is for this reason that design tools like the flowchart are created.

A flowchart is a graphical tool facilitation of the creation and communication of algorithm ideas. It is designed to be both easy to create and easy to read, so even non-technical stakeholders can understand a design.

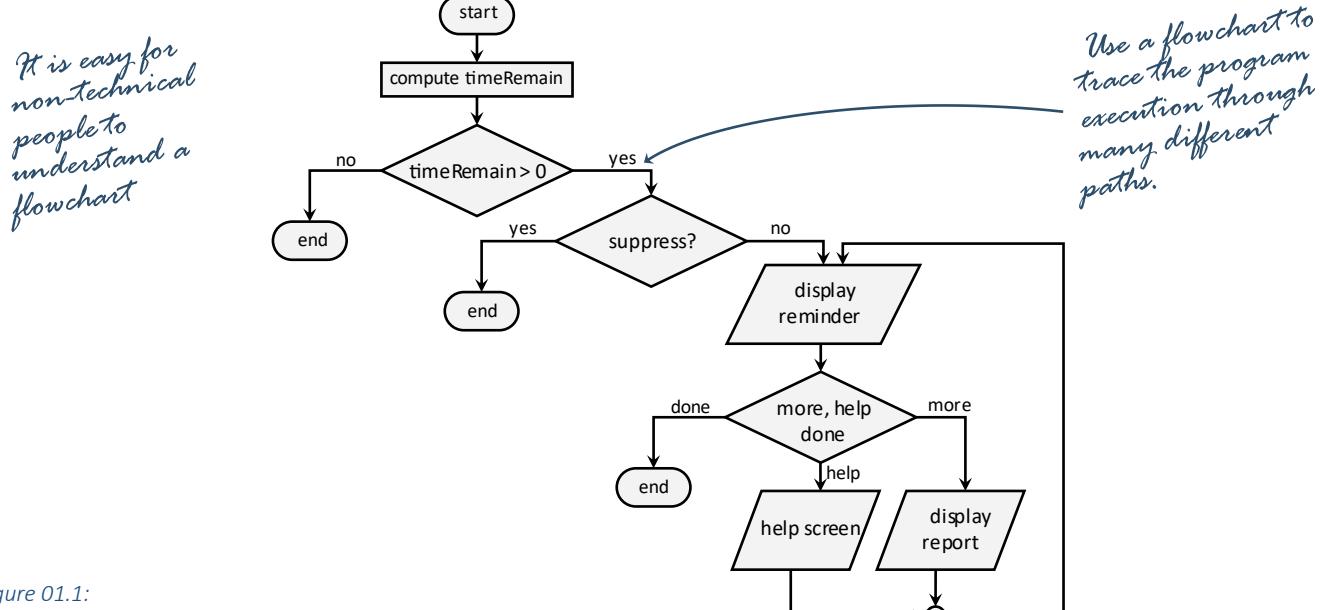


Figure 01.1:
Flowchart

Flowcharts have the following properties:

Property	Description
Use	Design and illustrate program flow
Viewpoint	Process: Flowcharts represent algorithms
Strength	Easy to understand, easy to create
Weakness	Can get messy, difficult to modify

A commonly used analogy used to explain how computer programs work is a cook following recipe. In this analogy, the cook reads one instruction in the recipe, completes the task, and then moves on to the next one until the meal is prepared. The computer fulfills the role of the cook here and the computer program (also known as the code) is the recipe. The computer keeps track of the current instruction with an “instruction pointer” (IP). When one instruction is completed or executed, the IP moves to the next one in the list. This continues until a special “end of program” instruction is encountered.

The problem with this analogy is that computer programs are almost never linear. Frequently the program contains subtasks requiring the IP to jump to a different part of the program and then return to the old location when finished. Programs are also filled with decision points, conditionally jumping to other parts of the program depending on the task at hand. These “jump” instructions are very common in code, comprising almost 20% of all instructions. As you can imagine, all these jumps make the program difficult to understand. We call the various paths that the IP can take through the code “program flow.”

A key part of designing quality software is managing program flow. The programmer wants the IP to take the best path through the program and not end up in an unexpected location. Fortunately, we have excellent tools at our disposal enabling us to visualize program flow in existing programs and to represent program flow ideas as we design new programs. The most commonly used tool to accomplish this is the flowchart. There are several advantages and disadvantages to using a flowchart.

An Instruction Pointer (IP) indicates which instruction or command is currently being executed in a program

Program flow is the various paths that an IP can follow when executing a program

Advantages	Disadvantages
Easy to visualize program flow	Can get messy
No special tools are required	Alterations can be difficult
Understandable by novices	Not all flowcharts can be translated to code
Facilitates finding defects	

Flowchart Elements

A flowchart is a graph used to represent program flow. In this graph, the IP begins in the “start circle,” following arrows to other chart elements, until the “end” circle is encountered. When a decision point is encountered, the IP follows the branch corresponding to the result of the decision. To see how this works, consider this flowchart representing whether a program should display a bill-pay reminder to the user:

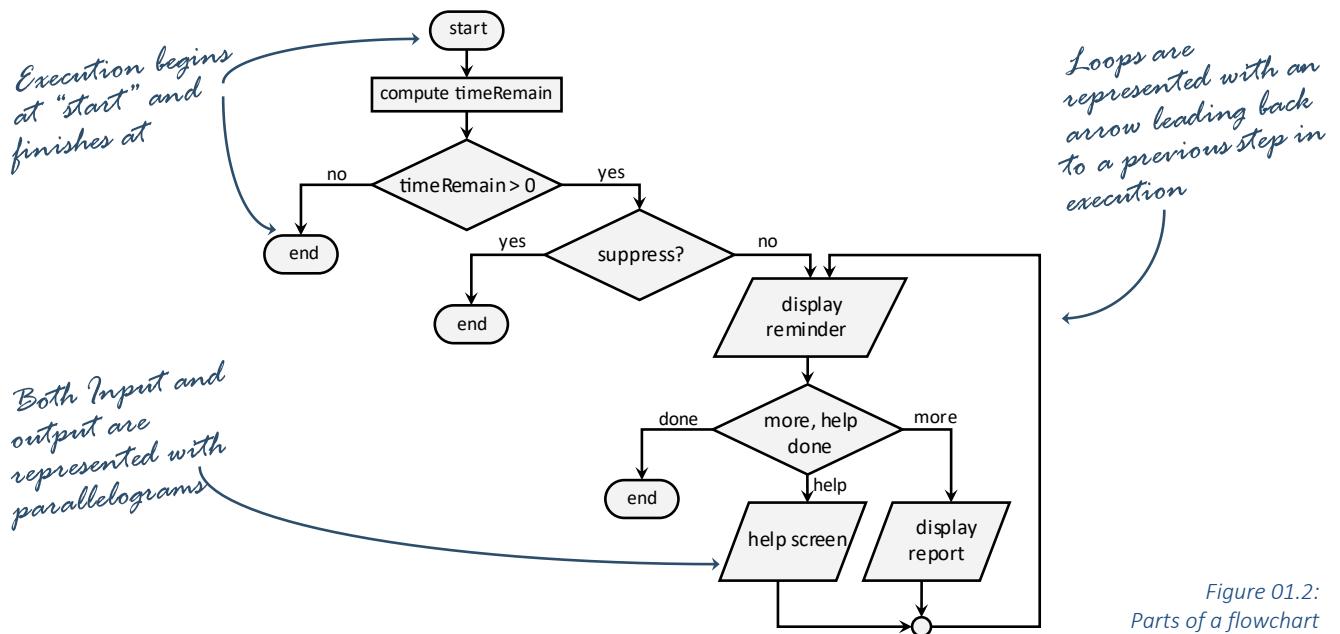


Figure 01.2:
Parts of a flowchart

One reads a flowchart by following the path that the IP will travel through the program, starting at “start” and finishing at one of the “end” circles. It should be readily apparent from just glancing at this program that the IP can take many paths depending on the state of the program and user input. It is the purpose of the flowchart to illustrate these many paths in the clearest way possible.

Symbol	Meaning
Terminator	The beginning and end of program execution
→	Arrows indicate the direction of program flow
Decision	Program execution can go multiple ways
Input & Output	Input and/or output
Processing	Something is computed

The flowchart consists of several symbols: the terminators, arrows, input/output, processing, and decisions. Each of these will be explored in detail on the next several pages.

Terminators

Rule 01.1 Flow starts at the “start” and ends at the “end”

All flowcharts begin with a “start” symbol and end with an “end.” Both are represented with a circle or an oval. You will often see flowcharts with a double outline around the start and end symbols to make them easier to spot in a complicated graph.

Rule 01.2 Start terminators have no inflow arrow and exactly one outflow arrow

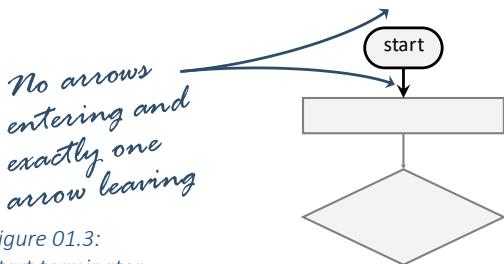


Figure 01.3:
Start terminator

Though flowcharts usually have a single start, there can be exceptions. Sometimes a program can have multiple entry points depending on how the program or subprogram is launched. In these cases, it is useful to put a note in the start symbol indicating the entry point. Start terminators do not have arrows leading into them and have a single arrow leading away. In other words, there is one and only one way for the IP to flow out of a start symbol.

Rule 01.3 End terminators can have many inflow arrows but no outflow arrows

Every flowchart has at least one end terminator, though it is common to have many. Some flowchart designers prefer a single end symbol where all possible termination points gather. There is some elegance to this approach. Unfortunately, however, this often results in overly complicated or messy flowcharts as dozens of possible branches converge on a single end. Therefore, many designers choose to use multiple end symbols. The choice is up to you. Every end terminator needs to have at least one arrow entering it and can have no arrows leaving it.

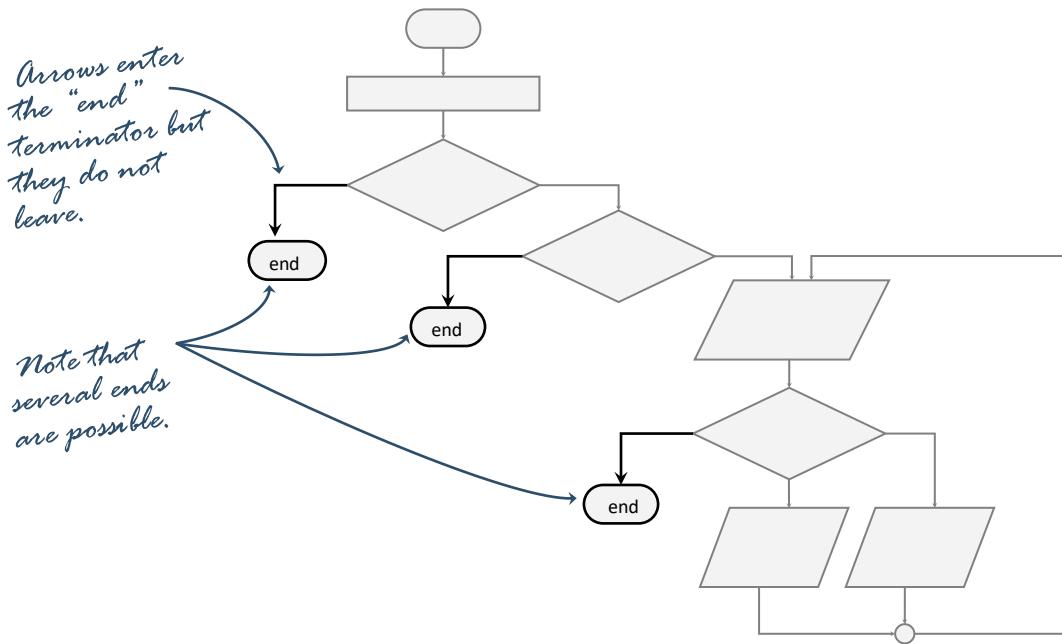


Figure 01.4:
End terminator

Arrows

You represent how the IP moves from one instruction to another with an arrow. All arrows (sometimes called flow arrows) are one-way, moving from one symbol to another. For the most part, arrows are not labeled. They indicate that the IP moves from one instruction to another, but do not indicate that data are flowing between parts of the program (we use a data flow diagram for that. See Chapter 11 Tool: Data Flow Diagram). The only exception to this rule is with decisions where the various choice options are labeled.

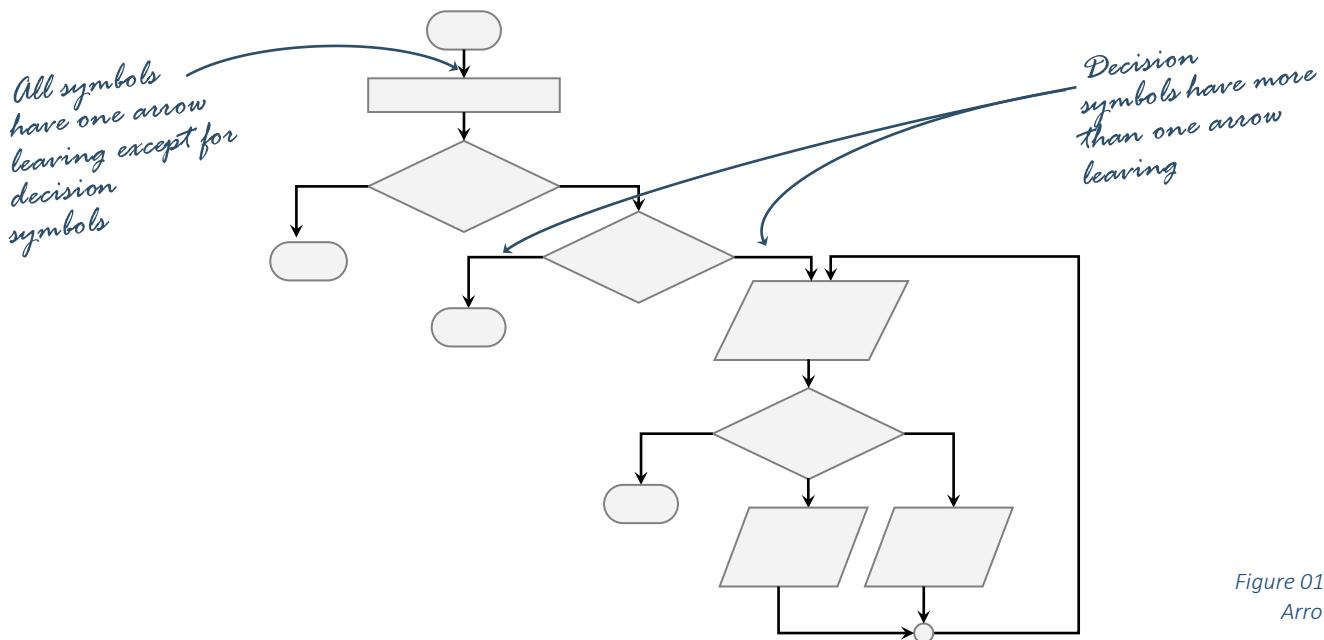


Figure 01.5:
Arrows

As you can imagine the arrows on a flowchart algorithm can get quite complicated. If the designer is not careful, then the flowchart can become a jumbled mess that is impossible to understand. There are a few things that can be done to help.

Best Practice 01.1 Make the flowchart more understandable by making your arrows point in the same general direction

It is generally a good idea to try to make the arrows point in the same general direction, such as top-to-bottom or left-to right. Put some thought into how your algorithm works. Try to make the main flow of control travel in a straight line. This makes it easier for the reader to follow the designer's intention.

Sometimes the algorithm needs the IP to move back to a previous instruction. This happens most often in loops. To represent this, the arrow moves circles back to a previous instruction against the dominant flow of the algorithm. This is called a "backtrack." In this case, you still use one-way arrows.

Only use a backtrack arrow when the algorithm calls for a loop. Do not use it when you run out of room on your page or when two flowchart elements do not line up nicely. In situations like that, it is better to redraw the flowchart rather than keep it cluttered and misleading.

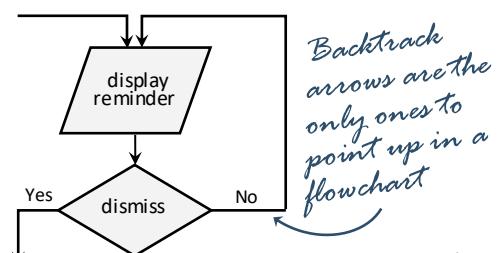


Figure 01.6:
Decision arrows

Sometimes, when a flowchart is very large or complicated, it is difficult to keep the arrows from crossing each other. There are occasionally times when arrows need to connect symbols which are far from each other on the flowchart. This can make things very messy and difficult to understand. To address this issue, we can use a connector.

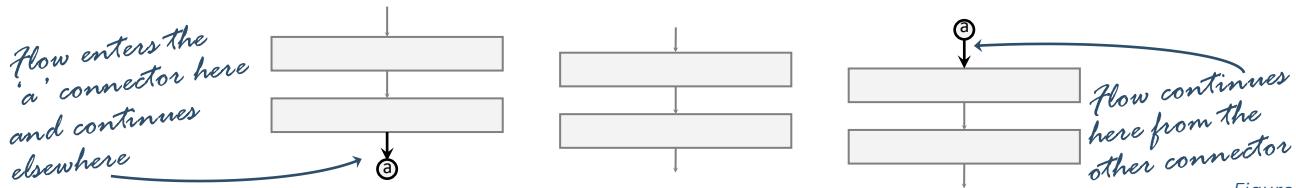


Figure 01.7:
Arrow connectors

An arrow connector is a pair of labeled circles indicating that two halves of an arrow are connected. As a rule, connectors should be avoided. It is better to more carefully lay out the flowchart or to break one large one into two. When this is not possible, connectors can make flowcharts more understandable by shortening arrows, avoiding crossing arrows, or by reducing the number of arrows entering a single location in the flowchart.

Connector serve another purpose. Oftentimes, there is more than one way for control to reach a given statement. Consider an IF statement that has two conditions: the TRUE condition and the FALSE condition. After the conditions are executed, then control reconnects and flows through the program.

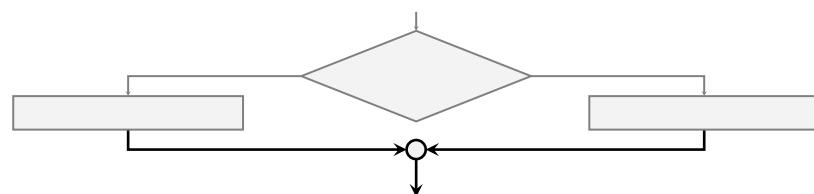


Figure 01.8:
Arrow connectors
with IF statements

Notice now multiple arrows can enter a connector, but only one arrow can leave. It is very important to put the arrowhead on connectors such as these, so it is unambiguous how control flows. This same general pattern applies for loops.

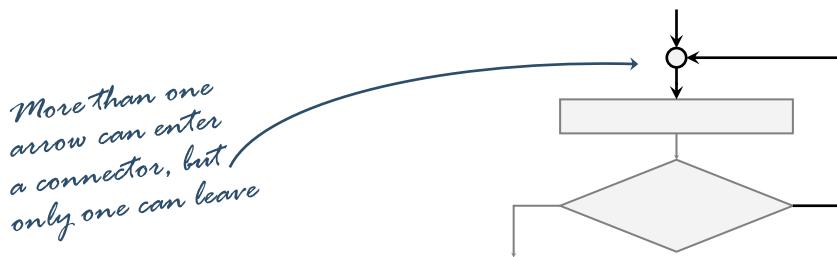


Figure 01.9:
Arrow connectors
with loops

Some designers do not use connectors when representing IF statements and loops. Instead, they just have multiple arrows enter a single symbol. This is a normal and accepted practice. Other designers think that the connector adds clarity to the design. Which will you choose? It is up to you.

Decisions

The strength of flowcharts as a way to represent algorithm designs comes from how decisions can be made understandable to the casual observer. Decisions are made with diamonds with the interior containing a question, condition, or set of options.

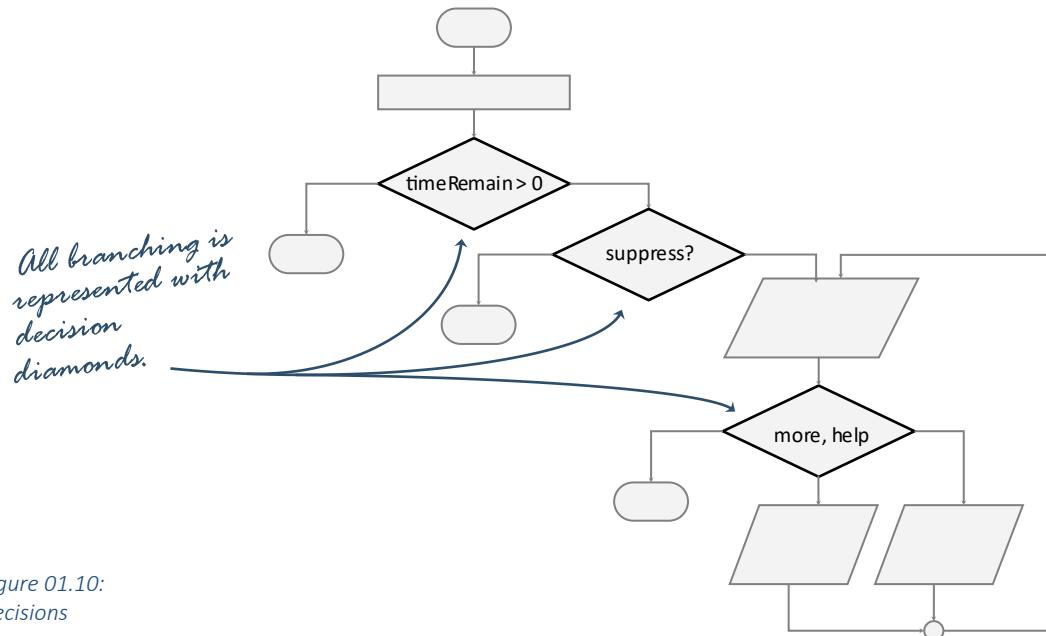


Figure 01.10:
Decisions

Rule 01.4 Decision diamonds are the only symbol with more than one outflow arrow

Unlike every other symbol in a flowchart, decisions have more than one arrow coming out. In the simplest case, a decision has two outcomes: true and false. Here, an arrow would enter the diamond from the top and two arrows would come out each side. One side will be labeled “true” and the other “false.”

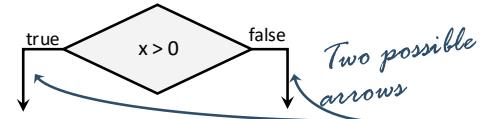


Figure 01.11:
Binary decision

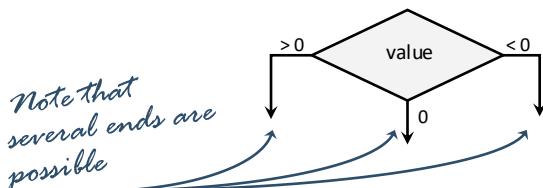


Figure 01.12:
Multi-way decision

It can be the case that there are more than two possible outcomes from a decision. In this case, many arrows can come out of a decision diamond. When this happens, care needs to be taken to leave room for the arrow labels or the flowchart could become unreadable.

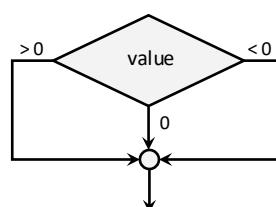


Figure 01.13:
Flow reconnected
after a decision
with a connector

When many branches from a decision need to reconnect into a single flow, the designer has two options. First, he or she can simply have two arrows enter the next symbol in the flowchart. This is probably the most common solution. Other designers may wish to have use a connector: a small circle indicating that multiple arrows converge. You can see an example of this in Figure 01.8 where the two options at the bottom of the flowchart reconnect at a connector before an arrow takes the IP back up in the flowchart.

Input and Output

Input and output (I/O) instructions are represented with parallelograms. Usually the designer will put a note or two in the parallelogram indicating what kind of data are being inputted or what is displayed to the user. Every I/O symbol has a single arrow leaving it. Program flow does not stop in an I/O symbol, and there is always exactly one direction for program flow to go once the input/output task is complete.

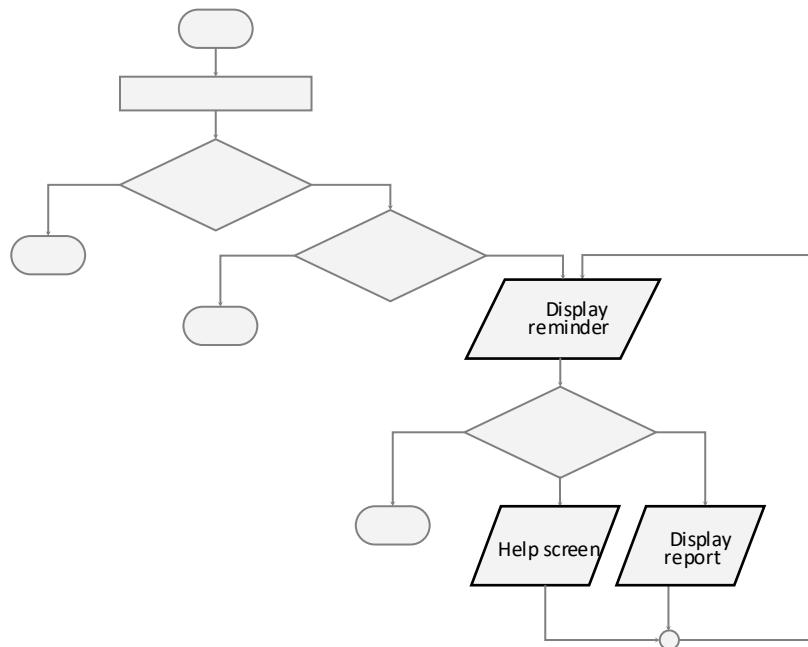
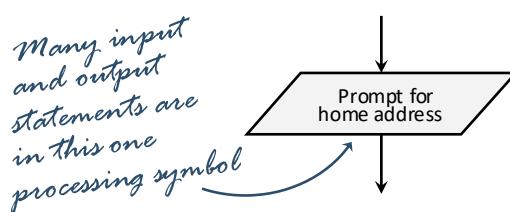


Figure 01.14:
Input and Output

There is usually no need to use different I/O symbols to indicate where data came from or where it is going. We also do not have to specify the data structure or variable holding the data. While it is certainly possible to represent these types of details with a flowchart, it is not the kind of thing that a flowchart is designed to convey. We can use data flow diagrams (Chapter 11 Tool: Data Flow Diagram) and pseudocode (Chapter 02 Tool: Pseudocode) for that.



Designers frequently combine input and output instructions into a single flowchart symbol. For example, a prompt usually includes both an output statement with a question and an input statement to accept a result. These can easily be incorporated into a single I/O symbol. One can also use a single I/O symbol to represent very complex output involving hundreds of lines of code.

Figure 01.15:
Complex I/O in a single
flowchart element

Processing

The processing symbol is used to represent any type of work that the program is performing. This could be simple tasks such as adding numbers or large tasks such as sorting numbers. The size and complexity of the task is at the programmer's prerogative. The processing symbol is a rectangle in a flowchart.

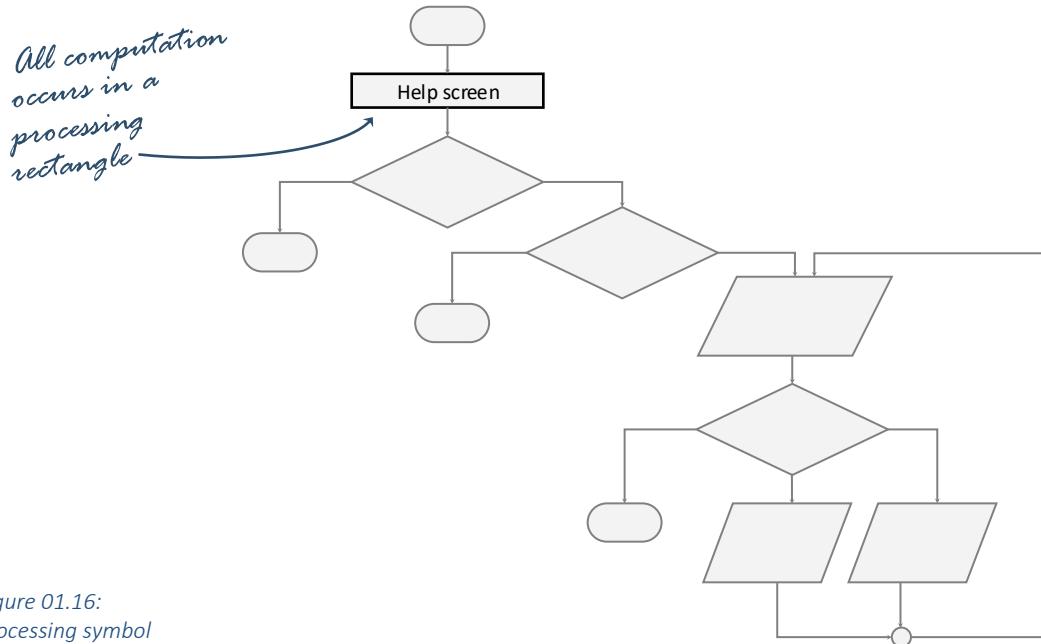


Figure 01.16:
Processing symbol

When a flowchart gets too large or complicated to fit on a single sheet, then it is often useful to break it up into many smaller flowcharts. This can be accomplished using the processing symbol. First, you cut out the part of the flowchart you would like to represent separately. We will call this "Flowchart #2" for now. This flowchart will have a start and stop symbol just like all others. Then you create a new processing symbol to represent this new flowchart. Label that processing symbol with "Flowchart #2." The following two smaller flowcharts (Figure 01.15) represent the same algorithm as the larger one above (Figure 01.14).

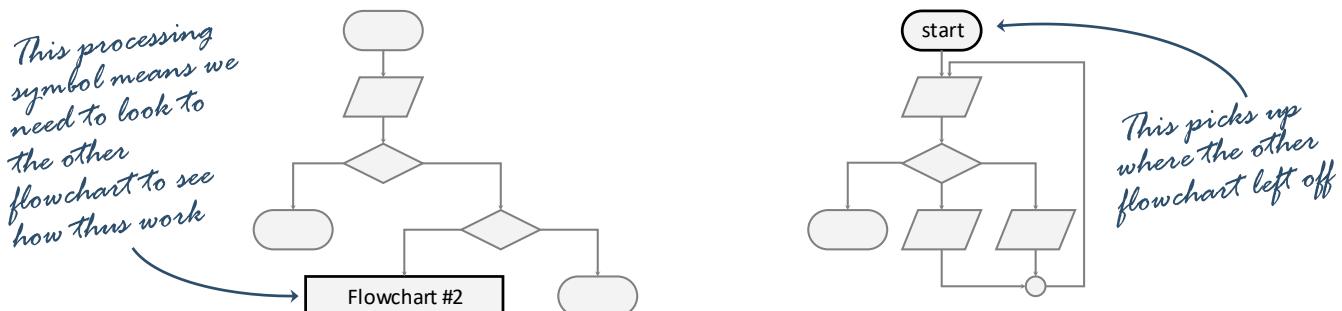


Figure 01.17:
Linking two flowcharts
with a processing symbol

Alternatives

The first attempt to diagram process flow can be accredited to Frank and Lillian Gilbreth in 1921. Their “flow process chart” is similar in function to the flowchart but with different symbols and conventions. John Van Neumann used flowcharts with his early explorations into computer programs. These tools are readily recognizable as flowcharts, though the symbols and other details may be slightly different. Today, there are several types of graphs used by programmers: flow process chart, swimlane flowchart, workflow diagram, activity diagram, and decision tree diagram.

Flow Process Chart

Developed by Frank and Lillian Gilbreth to describe business processes, the flow process chart arranged all the symbols in a vertical line. This made it possible to easily add comments and details about the processes being represented but made it more difficult to understand the structure of the program.

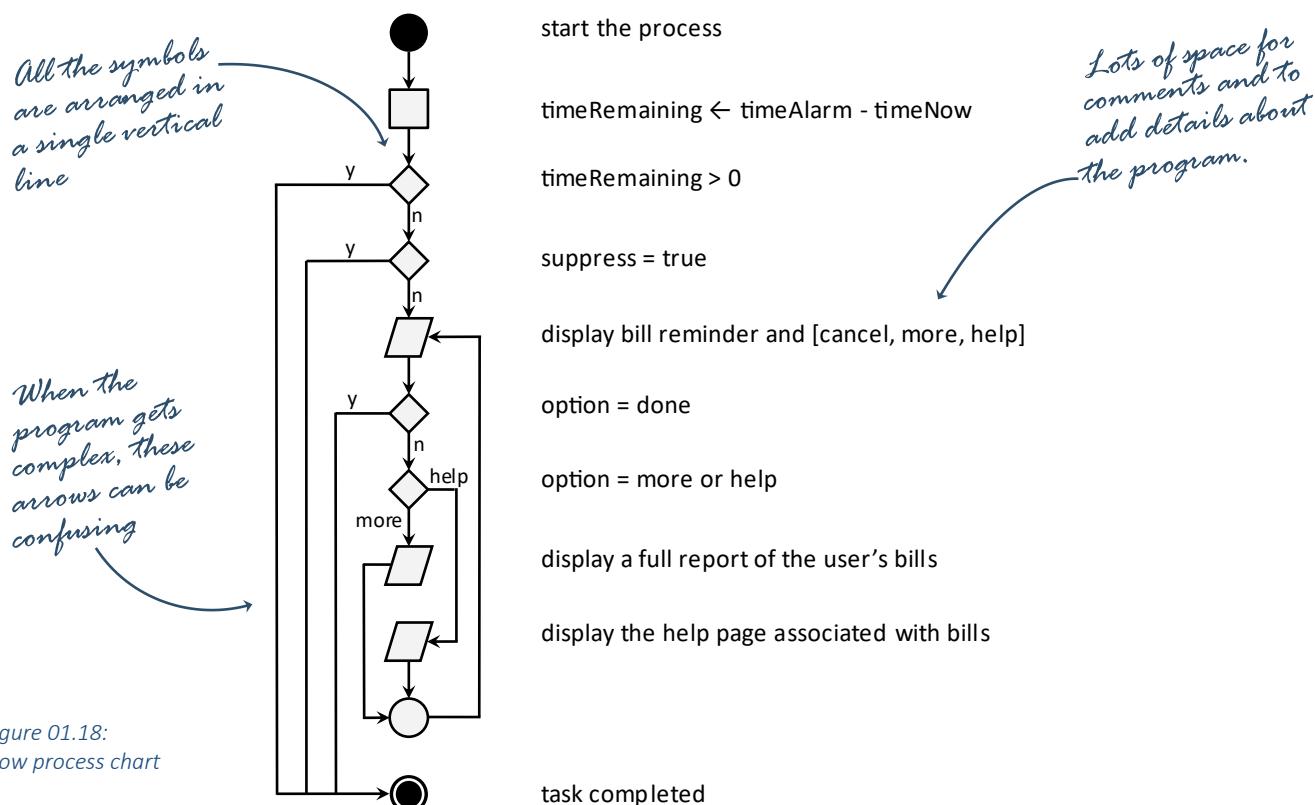


Figure 01.18:
Flow process chart

When working with assembly language (such as with embedded systems), a flow process chart may be a good alternative to the flowchart. It facilitates working with linear languages and gives plenty of room for annotations. In those cases, though, it may be useful to replace the flow process chart symbols with flowchart symbols so a wider audience can understand them.

Advantages	Disadvantages
Easy to add new instructions Great for coding in assembly Lots of room for comments	Difficult to follow when there are lots of branches Not often used anymore

Swimlane Flowchart

Though the original author of the swimlane flowchart (or just “swimlane charts”) is unknown, it first was developed in the 1940s as a way to represent processes that cross organizational boundaries. Modern swimlane flowcharts use the same symbols as flowcharts but with vertical columns representing structural borders.

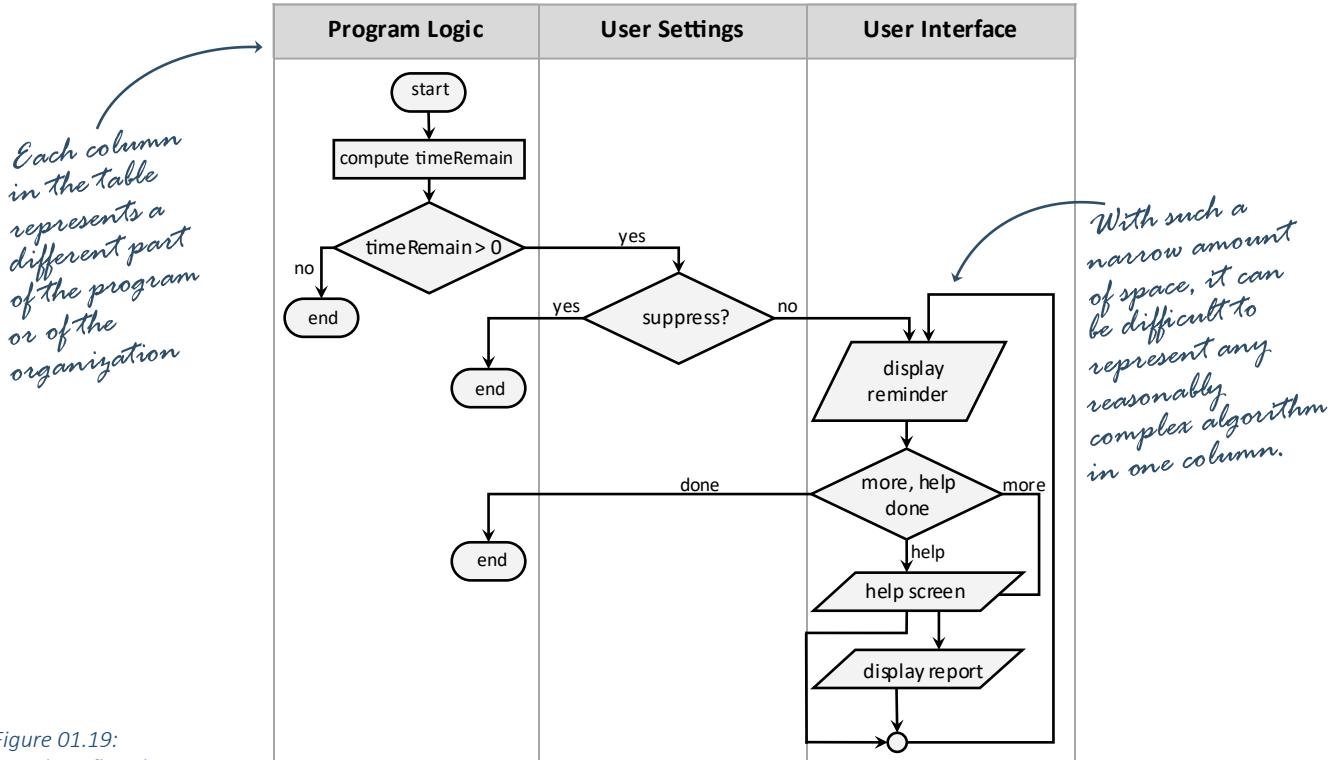


Figure 01.19:
Swimlane flowchart

Swimlane flowcharts are commonly used in business settings where multiple organizational boundaries exist. In the computational context, there are seldom equivalent needs. There are exceptions, of course. For example, some programs have processes which operate in different levels of security or clearance. These can be represented by overlaying swimlanes onto flowcharts.

Advantages	Disadvantages
Uses standard symbols	One swimlane tends to dominate the chart
Easily understood	Only works with a small number of swimlanes
	It can be difficult to fit a complex algorithm in one lane

Activity Diagram

The Unified Modeling Language (UML) is a suite of tools used to diagram various parts of a program. We will learn about many UML tools in later chapters. One such UML tool is an activity diagram. Because UML is such a commonly used standard, activity diagrams are often utilized in the software design process.

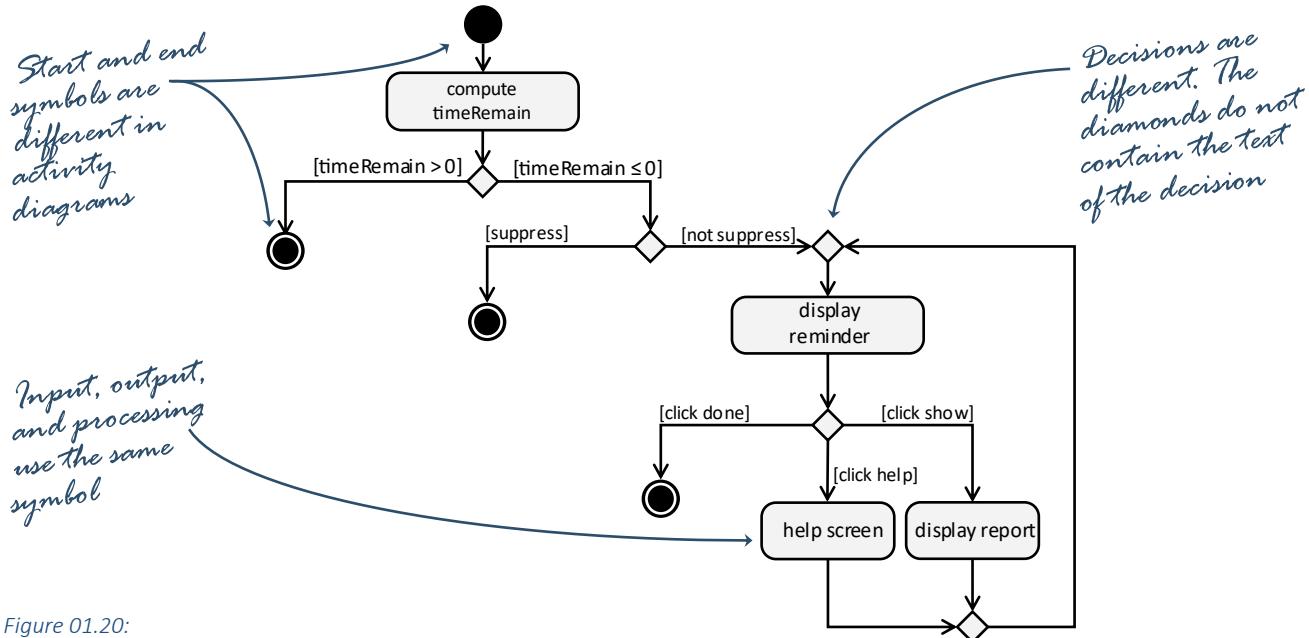
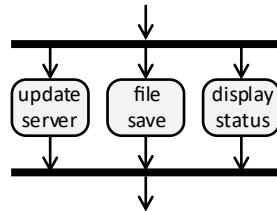


Figure 01.20:
UML Activity Diagram

UML activity diagrams are essentially flowcharts and use many of the same symbols. The main differences are slightly different start/stop symbols, the absence of I/O symbols, and support for more programming constructs. These symbols are:

Symbol	Name & Description
●	Start: The same as a flowchart terminator
○	End: The same as a flowchart terminator
→	Connector: The same as a flowchart arrow
◇	Decision: Same as flowcharts but no text in the diamond
□	Activity: Combination of a processing and I/O symbol
→→	Joint/Synchronization: More than one arrow combining
→→	Fork: Two or more activities going on at the same time

The most important difference between a flowchart and an activity diagram is that a single arrow can split into two without a decision. This enables two concurrent activities. An example would be multi-process or parallel processing. For example, a program may send a transaction to the server, save a file, and display a message to the user at the same time. They are all started at the same time, are carried out in parallel, and then the program proceeds once they are all complete.



All three of these processes go on concurrently

Figure 01.21:
Forks and joins

Decision Tree

A decision tree is a graph used to represent the many outcomes of a large or complex decision. As its name implies, a decision tree can be thought of as a physical tree where the trunk represents the beginning of the decision-making process. At each decision point, branches veer off representing the consequences of choosing that alternative. The process continues until the end of a branch (called a “leaf”) is reached. A decision tree can have just two leaves or hundreds, depending on the complexity of the decision being represented. For example, consider the following decision tree diagram demonstrating how to choose what type of account a user needs.

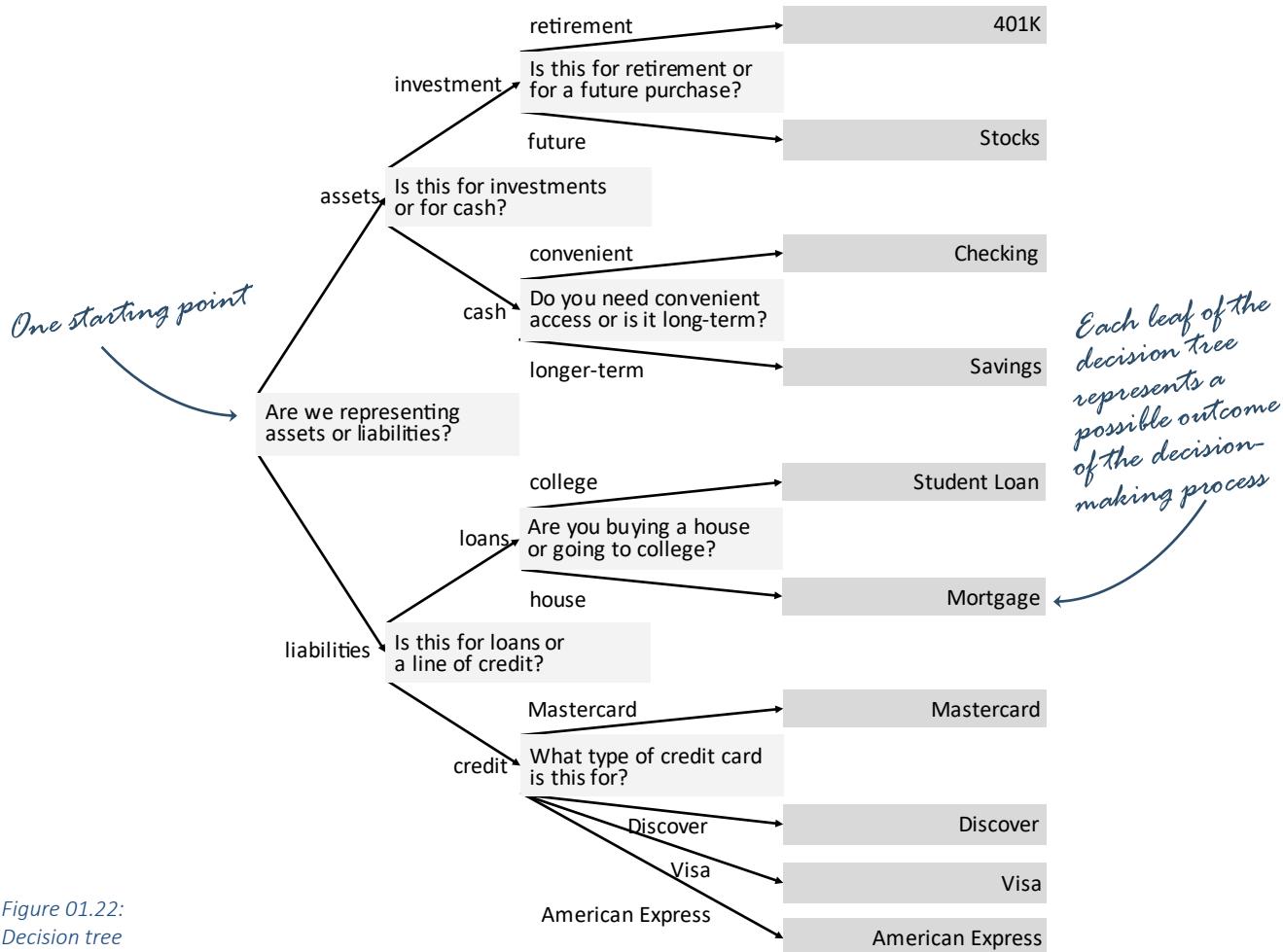


Figure 01.22:
Decision tree

Decision trees cannot represent complete algorithms; they can only represent decisions. In other words, there are many ideas represented in flowcharts that cannot be represented with a decision tree diagram. The opposite is not true. Every single decision tree can easily be represented with a flowchart. We thus say that decision tree diagrams are subsets of flowcharts.

Advantages	Disadvantages
Illustrates complex decisions Easily understood by novices	Only useful for some problems Converging decision can be messy

Using Flowcharts

There are two common uses of a flowchart: to capture design ideas and to communicate design ideas to the other stakeholders on a development team.

Designing with Flowcharts

Flowcharts are best used in the design process when complex decisions are made by the software. They can also be used to represent and untangle complex programming logic. In both cases, the same process is used: 1) draft your initial ideas, 2) refine the design, and 3) validate the flowchart.

Step 1: Draft

Most programming problems begin with a problem description. This could be an assignment from a teacher, instructions from a customer, or a formal specification. In any case, we always start with a problem description.

From your problem description, look for input, output, and processing elements. These are usually easy to translate into flowchart symbols. It is often helpful to cross out these parts of the problem description once they are represented in the flowchart. The tricky part is to find decisions and loops. Often these are hidden or implied. When one is found, add it to the flowchart and cross out that part of the problem description. Continue this process until most or all the problem description is crossed out.

Sometimes the problem description is complete enough that this yields a somewhat working draft of a flowchart. Sadly, this is the exception rather than the rule. Instead, it is often necessary to rewrite the problem description in more detail and begin the process anew. Two or three sentences from the project sponsor can thus become a multi-paragraph problem description when it is fleshed out enough to begin drafting a solution.

It is often necessary to rewrite, expand, and elaborate on problem definitions so enough detail can be had to write code

If this process results in a fully-connected flowchart, then you are ready for Step 2. If not, then you might need to carefully reread the problem description to see how the various fragments connect and relate to each other. This will probably initiate another round of rewriting and refining the problem description.

Step 2: Refinement

Once an initial draft of the flowchart is finished, we need to go through a refining process before we can write code. This step is often skipped by new programmers

or impatient students! This step is not only crucial for creating high-quality software, it also saves a lot of time. Mistakes are more easily found in the refinement stage of design than they are in coding. They are also much easier to fix.

Mistakes are more easily found in the refinement stage of design than they are in coding

One refinement activity is to use standard symbols. This may seem like a waste of time; our goal is to produce working software, not “correct” flowcharts. However, nonstandard symbols are a good indicator of poor design. It could mean we don’t fully understand the algorithm or that our approach will not quite work. To combat

this, we need to more carefully look at the questionable parts and perhaps break them into smaller, more understandable design elements.

Another refinement activity is to keep the flowchart to one page. Clearly many programs are far too complex to fit on one page unless we use unreadably small symbols. Instead, break large flowcharts into smaller, more manageable ones. This subdivision process gives us the freedom to focus on small parts of the big design and momentarily forget about the complexities of other parts of the program.

Have one dominant direction for the arrows of your flowchart. In other words, have most of your arrows go from left-to-right or from top-to-bottom. Why does this matter? Mostly, it is because flowcharts are easier to understand if control flows in

one direction. The easier a flowchart is to understand, the more readily defects and improvement opportunities can be found. The process of “straightening out” a flowchart serves to highlight ways to fix your design.

The process of “straightening out” a flowchart serves to highlight ways to fix and refine your design

The final part of the refinement process is to align shapes, avoid crossing lines, and evenly space out the various elements. This seemingly mechanical process forces the designer to reconsider all aspects of the design. In other words, a few minutes of “tidying things up” can pay big dividends as you remember parts of the problem definition that were not quite satisfied.

Step 3: Validation

Validation is the process of making sure that the algorithm design will meet the problem description. This must be done eventually, either by the user of the software (hopefully not!) or by the programmer testing his or her code. However, the earlier a problem is found, the easier it is to fix. This is certainly true with flowcharts.

Flowcharts are particularly effective at facilitating algorithm validation. Begin at the start terminator and follow the arrows to the end. At each step of the way, make the sure that the program is performing what the problem description specifies. When you get to a decision, make sure to follow each branch all the way to the end terminator. For complex flowcharts with many branches, it may take several times to fully explore every branch. This walkthrough process is much easier with flowcharts than with other program drafting tools. It is even easier than with completed source code.

The algorithm walkthrough process is much easier with flowcharts than with other programming drafting tools

Using all three of these design steps – drafting, refining, and validating – can greatly reduce the code-writing and debugging time of any nontrivial software development task.

Communicating with Flowcharts

It is seldom the case that a development team consists of a single software engineer working behind a closed office door. Software production is a team effort involving a variety of stakeholders representing many disciplines. Each of these stakeholders needs to understand the project enough to make informed decisions so they can contribute in meaningful ways. This is particularly challenging when some of these stakeholders are non-technical.

It is seldom the case that a development team consists of a single programmer working behind a closed office door

For example, consider the financial software example we have been using throughout this chapter. While the programmer and the quality assurance engineers are certainly important members of the development team, they by themselves cannot hope to complete this project. Input will also be required from marketing (those tasked with identifying the value proposition of the software so people will want to buy it), user assistance (those writing the documentation to help users when they need help understanding something), usability engineers (those creating the user interface), and domain experts (in this case, the financial experts who know how banks, taxes, and currency exchanges work). If the programmer shows source code to these non-technical stakeholders, they will not be able to help with the development process. This is where flowcharts come in.



*Figure 01.23:
Using a flowchart
to communicate
design ideas*

In the above example, the developer needs to know that the logic used to determine whether a given item in a checking account ledger should count for a tax credit. To do this, the developer displays the flowchart on a screen and walks through the various paths. The domain expert, an accountant in this case, is then able to certify that the algorithm is correct or make necessary corrections.

Examples

Example 01.1: Flowchart from Code

This example will demonstrate how to create a flowchart to match a simple program.

Problem:

Create a flowchart corresponding to the following C code.

```
C

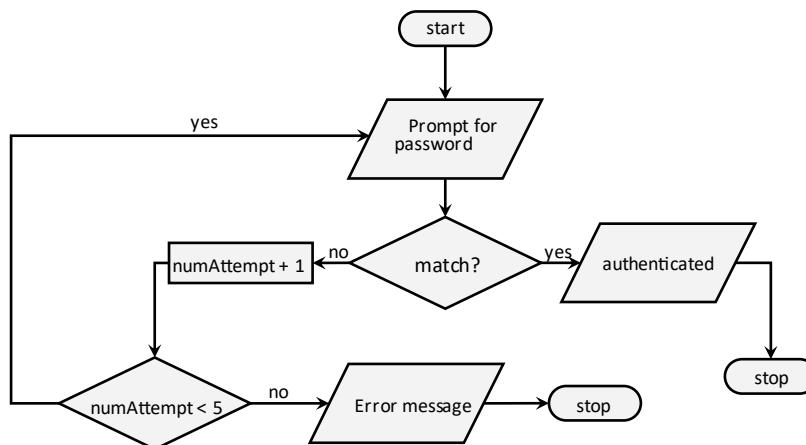
int main() {
    const char * password = "rosebud";
    char userInput[256];
    int numAttempts = 0;

    do {
        printf("password: ");
        getline(userInput, 256);

        if (strcmp(password, userInput) == 0) {
            printf("You are authenticated\n");
            return 0;
        }

        numAttempts++;
    } while (numAttempts < 5);
    printf("You are out of tries\n");
    return 1;
}
```

Solution:



Example 01.2: Flowchart from Code

This example will demonstrate how to create a flowchart from a slightly more complex program.

Problem:

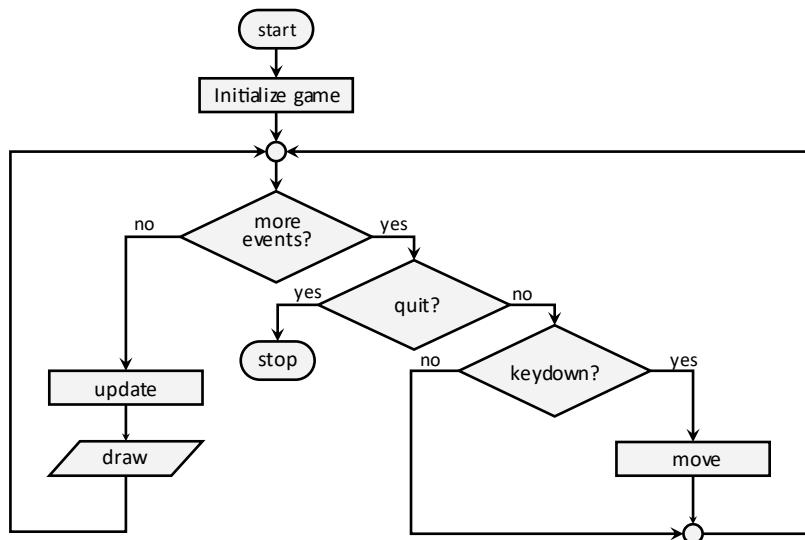
Create a flowchart corresponding to the following Python code.

```
Python
display = pygame.display.set_mode((640, 480))
pygame.display.set_caption('Example 01.2')

clock = pygame.time.Clock()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            exit()
        if event.type == pygame.KEYDOWN:
            move(event)
    updateGame()
    drawBoard()
    drawCharacters()
    drawScore()
    pygame.display.update()
    clock.tick(30)
```

Solution:

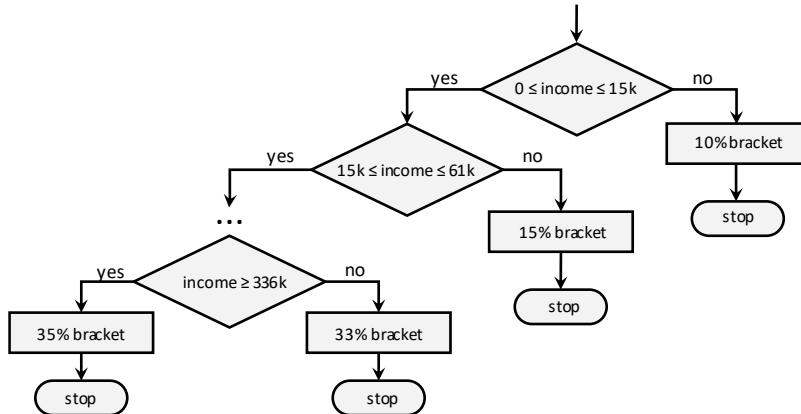


Example 01.3: JavaScript from Flowchart

This example will demonstrate how to create a JavaScript program corresponding to a complex flowchart.

Problem:

Create the JavaScript code corresponding to the following flowchart:



Solution:

JavaScript

```
function computeTax (income) {
    var tax;

    if (income >= 0 && income < 15100) {
        tax = income * 0.10;
    }
    else if (income >= 15100 && income < 61300) {
        tax = 1510 + 0.15 * (income - 15100);
    }
    else if (income >= 61300 && income < 123700) {
        tax = 8440 + 0.25 * (income - 61300);
    }
    else if (income >= 123700 && income < 188450) {
        tax = 24040 + 0.28 * (income - 123700);
    }
    else if (income >= 188450 && income < 336550) {
        tax = 42170 + 0.33 * (income - 188450);
    }
    else if (income >= 336550) {
        tax = 91043 + 0.35 * (income - 336550);
    }
    return tax;
}
```

Example 01.4: Flowchart for Monopoly

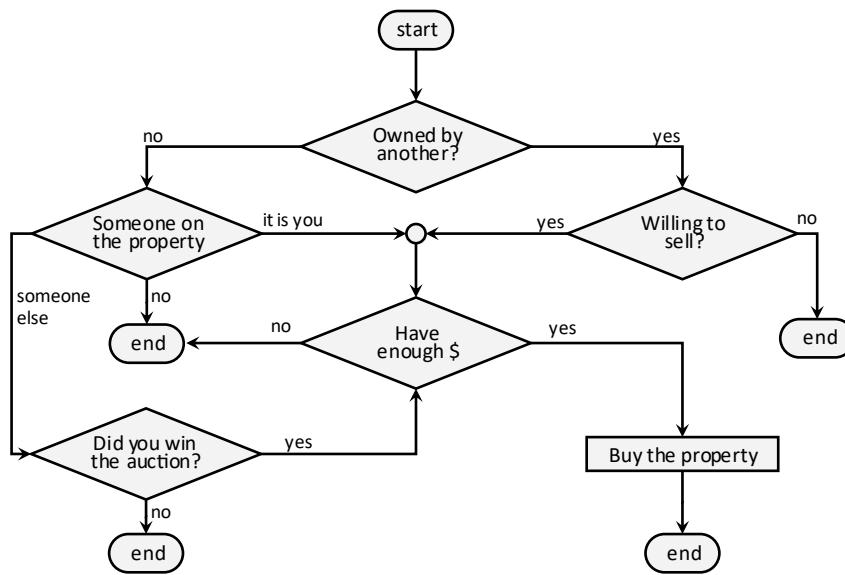
This example will demonstrate how to create a flowchart from a real-world complex business problem.

Problem:

Create a flowchart describing if a player can purchase Pennsylvania Avenue in Monopoly. The following rules apply:

1. If the property is unowned and you land on it, you have the option to purchase it for the list price.
2. If the property is unowned, a player lands on it, and the player decides not to purchase the property, the property is auctioned. Bidding may start at any price and any player (including the one who landed on the property) can make a bid.
3. A player can purchase a property from another player at any time provided they both agree on the price.
4. The full price of the property must be paid at the time of purchase. If the player cannot come up with sufficient funds, then the property cannot be purchased.

Solution:



Exercises

Exercise 01.1: Parts of a Flowchart

Draw the symbols corresponding to the parts of a flowchart.

Description	Drawing
Input symbol asking the user for his/her age	
Something connecting one processing instruction to another	
Start of the program	
Computing the age in days from the age in years	
Output symbol displaying a graph of the user's monthly budget	
A decision representing whether a user is over 65	
End of the program	
The symbol leading out of a decision corresponding to the case where the user is over 65	

Exercises 01.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Flowcharts track the movement of the IP through the program	
Flowcharts are difficult for non-technical people to understand	
More than one arrow can enter a processor	
More than one arrow can leave a processor	
Any flowchart can be represented with a decision tree	

Exercises 01.3: Authenticate

Create a flowchart for a program to compute whether a user is authenticated. The logic is the following:

Prompt the user for a username and a password. If the username is “administrator” and the password is “passw0rd,” then present the message “greetings.” Otherwise, present the message “you are not authenticated.”

Exercises 01.4: Loop

Create a flowchart for a program to sum the values from 1 to 10.

Problems

Problem 01.1: Ship Upgrade

Create a flowchart for a program to compute whether a given ship in a video game should be upgraded to the next level. The logic is the following:

A ship advances to level three if the following conditions are met: the ship is currently at level two, the ship has accumulated 200 points, and the ship has been at level two for at least one minute.

Problem 01.2: List Display

Create a flowchart representing a program to display the contents of a given to-do list. The logic for this program is the following:

Advance through all the list items in the program. If a list item is of type "to-do," then display that item to the user. Otherwise, skip that item and continue through the rest of the list.

Problem 01.3: Login

Create a flowchart matching the following problem definition:

Prompt the user to see if he/she has a username. If not, then prompt them for a username and password and store both in our authentication dictionary. Next, prompt the user for his/her username and password. If there is a username and password in the authentication dictionary, then display "you are authenticated." Otherwise, display "Invalid credentials" and prompt them again.

When finished, implement this algorithm in the programming language of your choice.

Problem 01.4: Point Tabulation

Create a flowchart matching the following problem definition:

This program has a list of points, each point has a value and a Boolean indicating whether it is to be included in the score. Iterate through the list, adding it to the total if the Boolean value is true. When the end of the list is reached, then display the total.

When finished, implement this algorithm in the programming language of your choice.

Problem 01.5: Leap Year Decision

Create a flowchart representing a program to compute whether a given year is a leap year. The logic for a leap year is the following:

According to the Gregorian calendar, which is the civil calendar in use today, years evenly divisible by 4 are leap years, with the exception of centurial years that are not evenly divisible by 400. Therefore, the years 1700, 1800, 1900 and 2100 are not leap years, but 1600, 2000, and 2400 are leap years.

When finished, implement this algorithm in the programming language of your choice.

Problem 01.6: Linear Search

Create a flowchart representing a linear search. The logic for the search is the following:

A linear search is a method used to determine whether a given value is in an array. There are three inputs: the array containing elements which may or may not match the searched value, the value to be searched, and the number of elements in the array. The algorithm begins by comparing the search value against the first element in the array. If the first element in the array matches the search value, then terminate the search and display a “found” message. If it is not a match, then proceed to the second item. Continue until either the element is found or the end of the array is reached. If this is the case, then display a “not found” message and end the program.

When finished, implement this algorithm in the programming language of your choice.

Problem 01.7: Authentication from a File

Create a flowchart representing a program to authenticate a user. The algorithm for the program is the following:

The program read a collection of usernames and passwords from a JSON file. The program will then prompt the user for his/her username and his/her password. If the provided username/password pair have a corresponding entry in the JSON file, then indicate that he/she is authenticated. Otherwise, indicate that they are denied access.

When finished, implement this algorithm in the programming language of your choice.

Problem 01.8: Guessing Game

Create a flowchart representing a guessing game. The logic for the game is the following:

The program will prompt the user for a positive integer. The program will then randomly generate a number between 1 and that number. The user will then be prompted to guess the number. If the number is too high or too low, a message to that effect will be displayed. That number will then be stored in a list and the user will be prompted again. If the user guesses the number, then the number of guesses and all the attempts will be displayed to the user.

When finished, implement this algorithm in the programming language of your choice.

Challenges

Challenge 01.1: Child Tax Credit

Research on the internet how the “child tax credit” works with the United States Internal Revenue Service (IRS). Specifically, we are looking for the 2017 rules.

Create a flowchart representing a program computing a user’s child tax credit.

Create also a program in the language of your choice matching your flowchart.

Challenge 01.2: Advanced Search

Create a flowchart for the following advanced search algorithm:

Consider a person looking up the word "glitch" in a dictionary. At first, the word could be in any of the 688 pages in the dictionary. The user cuts the dictionary in half (to page 344) and notices the word "magnet" at the top of the page. This word sorts after "glitch." Because we can assume that the dictionary is sorted, we can deduce that the word "glitch" must be between pages 1 and 343. The user then cuts the remaining pages in half (to page 171) and finds the word "drift" on that page. Since "drift" sorts before "glitch," we can rule out pages 1 through 171. Thus, the possible pages must be 172 through 343. The user then cuts to page 257 and finds the word "guidance." Since "guidance" sorts after "glitch," we can rule out pages 257 through 343. This leaves pages 172 through 256. This process continues until the page containing "glitch" is located.

Your program must do the following:

- Prompt the user for a file containing a collection of sorted words.
- Read the file.
- Prompt the user for a word to search for.
- Perform the advanced search described above to locate the word.
- Display a message indicating whether the word exists in the list.

Challenge 01.3: Monopoly

Create a flowchart representing a program that will ask the user a series of questions and determine whether the player can place a hotel on Pennsylvania Avenue for the game Monopoly. For the purposes of this problem, we are not including the logic to purchase a given property. The rules of the game related to building a hotel are:

- Each house on Pennsylvania Avenue, Pacific Avenue, and North Carolina Avenue costs \$200.
- The properties in a color group must be developed evenly, i.e., each house that is built must go on a property in the group with the fewest number of houses.
- To purchase a hotel for Pennsylvania Avenue, Pacific Avenue, or North Carolina Avenue, one must first have four houses and then pay the bank an additional \$200.
- There can be no more than four houses and no more than one hotel per property. A property cannot have both hotels and houses.
- A hotel may be built on a color group only after all properties in the group have four houses. A player purchases a hotel by paying the price of an additional house, and returning the four houses on that property to the Bank in exchange for a hotel.
- When the Bank has no houses to sell, players must wait for some player to return or sell his/her houses to the Bank before building. If there are a limited number of houses and hotels available and two or more players wish to buy more than the Bank has, the houses or hotels must be sold at auction to the highest bidder.
- If a player wishes to buy a house/hotel for a property, it is not necessary to wait for their turn. The player can buy houses/hotels even if it is not their turn. The player does not need to be on the property they wish to put the house/hotel on.

Pseudocode

Chapter 02

Pseudocode is a vehicle used by the programmer to iteratively add structure to design ideas. This facilitates translation of design ideas into working code.

Unlike all the other design tools available to programmers (such as flowcharts, structure charts, data flow diagrams, UML class diagrams, and UML component diagrams), pseudocode is not a chart, diagram, or graph. Instead, it is text. In fact, the name “pseudocode” tells you all you need to know: “pseudo” means “fake” and “code” means, well, code! Pseudocode is an algorithm design language that cannot be compiled, but is useful as a vehicle with which we translate design ideas into the syntax of our target programming language.

Pseudocode is language independent. This means one should be able to write pseudocode without having to think about the specific style, syntax, or capabilities of any one programming language. It also means a single pseudocode design should be readily translatable into a variety of programming languages. Thus, anytime a programmer would like to present an algorithm that is usable in any programming language, then pseudocode is an obvious choice.

Pseudocode is language independent

Indents are used
for functions,
loops and IF
statements

Pseudocode
keywords
are used

Statements
are ordered
top-to-bottom

Each statement
is on its own line

```
Pseudocode
clearScreen( frame, color )
    IF not frame.initialized()
        RETURN
        yMax ← frame.height()
        xMax ← frame.width()
        FOR y ← 0 ... yMax
            FOR x ← 0 ... xMax
                frame[y][x] ← color
    END
```

Figure 02.1:
Pseudocode

Pseudocode has the following properties:

Property	Description
Use	Design algorithms
Viewpoint	Process: Pseudocode represents algorithms
Strength	Easy to create and modify, translates easily to code
Weakness	Only programmers can read pseudocode

Pseudocode Elements

Pseudocode has several rules, but these rules are far more lax than those for compiled programming languages. Remember, the goal here is to represent design ideas. In other words, pseudocode is a means to an end, not an end in itself. We can therefore bend the rules of pseudocode to meet the needs of the problem at hand.

Rule 02.1 Instructions are ordered top to bottom

With flowcharts, symbols are ordered according to the arrows that connect them. This can make it challenging to convert a flowchart into a programming language. Pseudocode is read top-to-bottom like most programming languages.

Rule 02.2 Each instruction is on its own line

For the sake of clarity, we avoid putting multiple instructions on a single line. Some programming languages allow this (the most notorious being C++). That being said, there is no guarantee that a single pseudocode instruction corresponds to a single programming language statement.

Rule 02.3 Indentations are used for loops, IF statements, and function membership

Many programming languages use curly-braces {} or parentheses () to indicate that a collection of statements are in a block – the most common use being to group statements into the body of an IF statement, loop, or a function. This is not necessary with pseudocode; simply indent the code rather than use curly-braces or parentheses.

Rule 02.4 Use actual variable names and equations

Rule 02.4 is perhaps the most difficult for new programmers to understand – how much detail should be included in a pseudocode program and what can be left out? Rule 02.4 addresses part of that question. Use actual variable names, call functions by name, include the actual logic of Boolean expressions, and include equations that will appear in the final production code. If you find yourself describing a process generally, then you are probably violating rule 02.4 and more detail is needed.

Rule 02.5 Use pseudocode keywords whenever possible

So, what are these pseudocode keywords? They map to the seven things computers can do: perform math, store data, receive input, send output, make decisions, repeat actions, and call functions. All the complex tasks that computers perform every day can be reduced to combinations of these simple things. It is remarkable if you think about it, but such is the magic of programming. The most commonly used pseudocode keywords are the following:

Keyword	Description
GET	Receive input from the user
READ	Receive input from a file, a server, or another program
PUT	Send output to the user
WRITE	Send output to a file, a server, or another program
IF	Make a binary decision
SWITCH/CASE	Make a multi-way decision
FOR	A counting loop
WHILE	All other loop applications

Perform Math

Though many of us look at our computers today as communication devices (for texting, browsing the web, and sending e-mail), the historical use for computers centered on making calculations. As a result, all computer languages have extensive number-crunching capabilities. The syntax, utility, and capabilities of these tools vary greatly from language to language.

All your favorite algebra and trig notations can be freely used

Pseudocode
$vt + \frac{1}{2}at^2$
$\sqrt{(dx^2 + dy^2)}$
e^{inx}
$s \cos \theta$

It is a goal of pseudocode to be language independent. We therefore do not have to worry about the specific mathematical library associated with each language. We only try to express ourselves using whatever mathematical notation is most convenient. In other words, our programming language or the associated libraries may force us to use strange formats or overly complex equations. We can ignore these when writing pseudocode and just use the notation we learned in our algebra, trigonometry, linear algebra, or calculus class.

Figure 02.2:
Mathematical expressions
in pseudocode

There are a few rules associated with mathematical expressions. First, rule 02.4 is worth repeating:

Rule 02.4 Use actual variable names and equations

This rule was mentioned previously, but is repeated here because it is particularly relevant when performing math. Your initial pseudocode draft may say, “use the distance formula,” but your final pseudocode should have the actual equation and the variables used to compute the value.

Rule 02.6 Make sure your intent is clear when using mathematical symbols

Do not use an obscure mathematical symbol or notation if your audience will not understand it. This is only important, of course, if you are collaborating with other programmers.

Rule 02.7 Honor the order of operations

If you need to add parentheses to make your intent clearer, then do so! If you do not honor the order of operations or any other mathematical conventions, then you risk miscommunicating your intent.

Rule 02.8 Avoid impossible constructs

Finally, avoid mathematical operations that you know your programming language does not support. If that is the case, then write a function to capture that operation.

Store Data

We represent the process of assigning values to variables in pseudocode with the arrow operator (\leftarrow). There are three parts to this design: the variable on the left, the arrow itself, and the expression or value on the right.

*The result goes
on the left side
of the arrow*



*The value or
expression goes
on the right side*

Figure 02.3:
Store data with the
arrow operator \leftarrow

Rule 02.9 Variable declarations are unnecessary

Unlike most programming languages, we don't need to declare the variable before using it. Declaring variables is among the housekeeping details that we can ignore when working with pseudocode.

Rule 02.10 There is no data type associated with the variable

Data types represent another housekeeping detail that pseudocode generally avoids. We may need to determine and/or specify a variable data type when we translate the pseudocode to a programming language.

Rule 02.11 The left-hand side of the statement is the value

The result of an assignment goes on the left side of an arrow and the value goes on the right side by convention. Note that we can put pretty much anything on the right side of an equation, such as a literal like the number 42 or the text "The Answer." It could be a mathematical equation (more on that later). It could even be something more abstract such as "the current interest rate."

Variations

Because pseudocode is a design language rather than a programming language, you often see variations in style or usage. Some common variations include using SET as a keyword, using the equals sign, and using the colon equals sign.

*One often sees
the SET
keyword used here*



*The assignment
operator and
Pascal-style :=
are commonly
used*

Figure 02.4:
Other ways to indicate
data are stored

You can use whatever variation you like. Note that if you use the equals sign for assignment values, then you need to use something else when you test for equivalence.

Receive Input

Input can enter a program from a variety of sources: from the keyboard, from the network, from a file, from the operating system, or even from another program. As a general rule, we break this down to just two categories in pseudocode: user input and machine input. We use the **GET** keyword when referring to user input and **READ** for everything else.

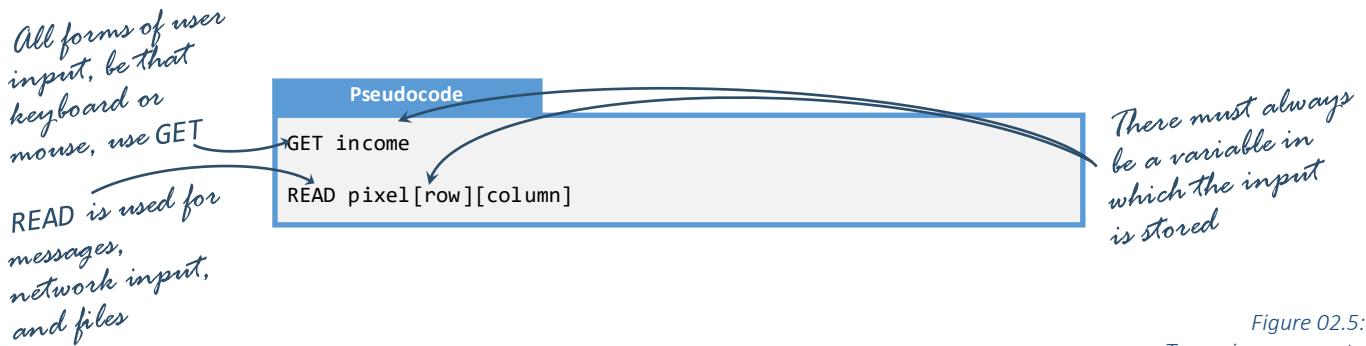


Figure 02.5:
Two primary ways to receive data

Rule 02.12 Specify the variable which will be receiving the input

Whenever the **GET** or **READ** keyword is used in pseudocode, make sure you also include the name of the variable which will be receiving the input so it is abundantly clear where the data will be stored.

Sometimes it is necessary to specify extra details to make your intent known. In this case, it is appropriate to add keywords or to make the description more verbose. As long as your intent is clear and it helps to produce the final working code, then the modifications are appropriate.

Variations

For example, consider the case when we want to read a line of text from a file. If the code simply says **READ fullName**, then many important details are lost.

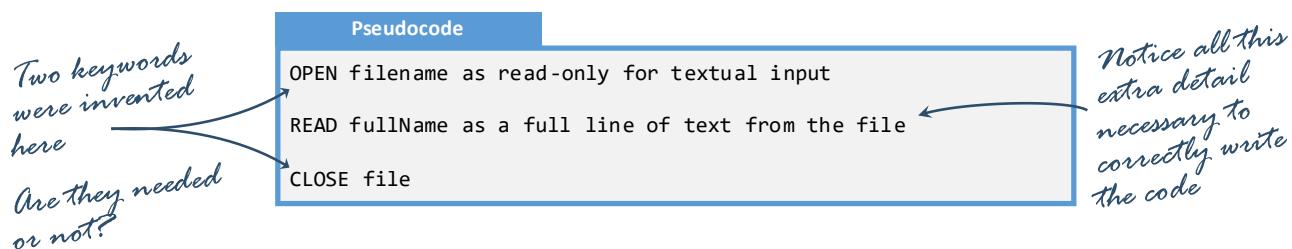


Figure 02.6:
Reading from a file

The question here is whether the **OPEN** and **CLOSE** keywords are necessary? If you are new to the language or you need to specify important details about how the file is opened, then you could argue that they are necessary. If these are trivial details that you feel any reasonable programmer would understand, then you might argue that these keywords are unnecessary.

Send Output

Just as the myriad of different options for accepting input are represented with a small number of pseudocode keywords, the same is true for sending output. Most pseudocode writers restrict themselves to two keywords: **PUT** when sending data to a human and **WRITE** for all other circumstances.

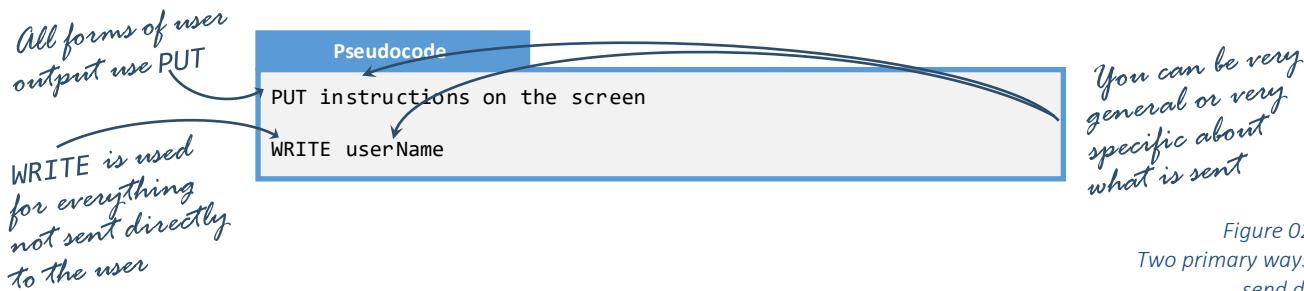


Figure 02.7:
Two primary ways to
send data

Note that the first line of output in the example above might correspond to dozens of lines of code. It is often useful to be very generic about the exact format and nature of the output. However, it might be the case that we need to be extremely detailed because the output is nontrivial. In those cases, then much more detail is needed. How much detail to you include in your pseudocode? It depends! It depends on how confident you are, as the programmer, in generating the corresponding code. It depends on whether the output code is trivial or complex. It depends on whether the format of the code is specified in another document (such as a user interface specification).

Variations

Many pseudocode authors find it useful to use other send keywords. The most common **PROMPT** (the combination of **PUT** and **GET**).

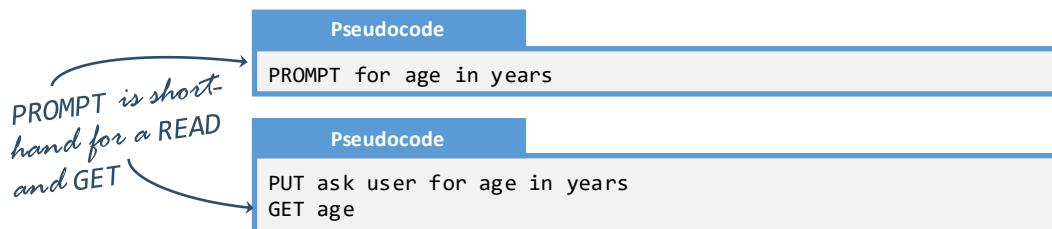


Figure 02.8:
The PROMPT keyword

It is often necessary to be more specific about the exact destination of output. The most common examples are sending data to a network location or sending data to a printer. If it increases the clarity of your pseudocode, then it may be necessary to create a new keyword.

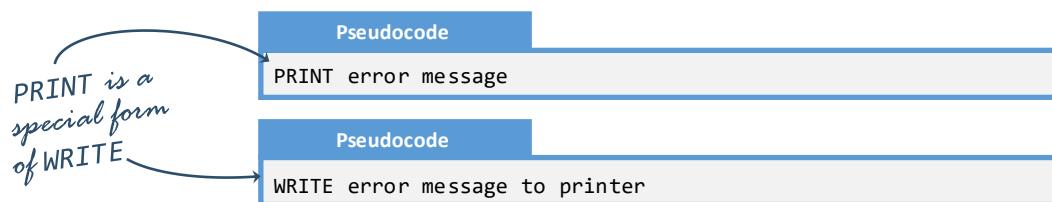


Figure 02.9:
The PRINT keyword

Make Decisions

The two main ways to make decisions are code is with **IF** statements and **SWITCH/CASE** statements. Both are represented in pseudocode.

IF Statement

In most cases, decisions are represented with the **IF** keyword in pseudocode. The format here is the same as we find in most programming languages: there is a Boolean expression after the **IF** keyword, there is a collection of statements corresponding to the true condition, and there is an optional else condition as well denoted with the **ELSE** keyword. Most pseudocode authors chose to omit the parentheses around the Boolean expression because they are unnecessary to indicate the intent of the code. Others use them because they are required in their programming language of choice. Which do you choose? It is up to you.

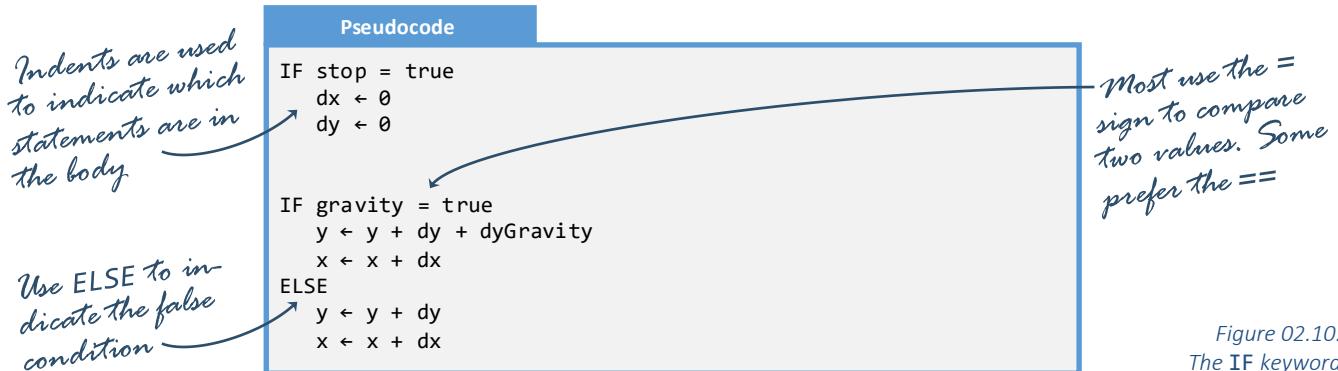


Figure 02.10:
The IF keyword

Switch / Case Statement

When there are more than one or two options in a decision, we can use the **IF/ELSE-IF** combination as we would in any other programming language. Most languages also provide a **SWITCH/CASE** statement for these circumstances.

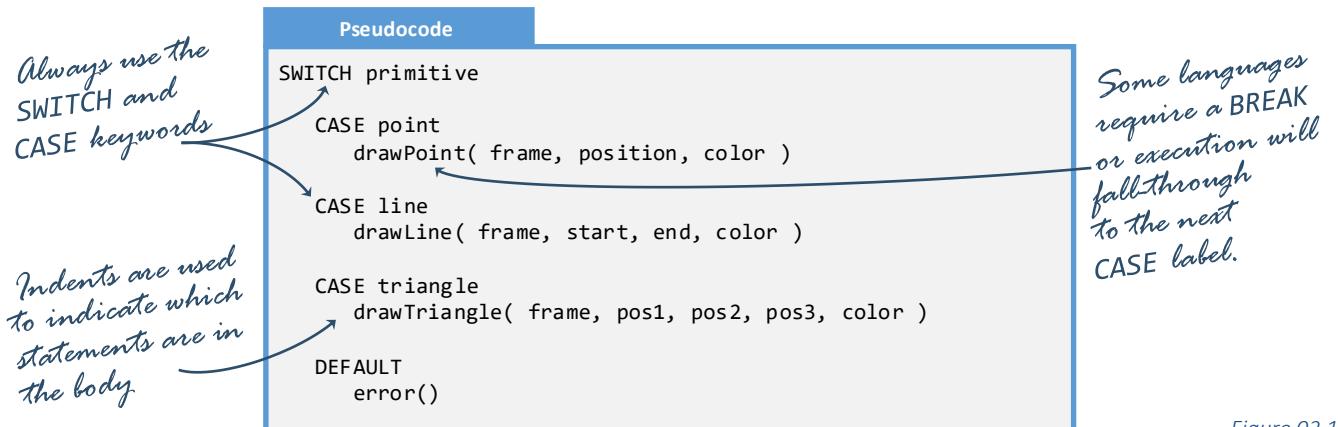


Figure 02.11:
The SWITCH/CASE keywords

Note that many languages provide restrictions on the use of **SWITCH/CASE** statements. C++, for example, requires the **SWITCH** expression to be an integer whereas JavaScript has no such limitation. Python does not even have a **SWITCH/CASE**. Therefore, care must be taken when using this construct or your design might not translate well to your language of choice.

Repeat Actions

Repetitive control structures, also known as loops, are represented in pseudocode with the **FOR** and **WHILE** keywords. We need to specify enough detail in this description so that the ambiguity is removed and any reasonable programmer can translate the pseudocode into a programming language. As long as this is accomplished, then the pseudocode is “good enough.”

WHILE loop

The **WHILE** loop has three parts: the **WHILE** keyword, the Boolean expression, and the body of the loop. As long as the Boolean expression evaluates to true, then the loop continues. As with **IF** statements, we can express the Boolean expression in any convenient way.

Indents are used to indicate which statements are in the body

Pseudocode

```
WHILE not done
    done ← processUserInput( events )
    clearScreen( frame )
    generateFrame( frame )
    sendFrame( frame )
```

Continue in the loop until the Boolean expression evaluates to false

Figure 02.12:
The **WHILE** keyword

Rule 02.13 Specify the Boolean expression in the **WHILE** loop

Every **WHILE** loop contains a Boolean expression indicating when the loop continues and when it terminates. This expression needs to be clearly stated in the pseudocode.

FOR Loop

FOR loops are for counting and, as such, can be more complicated than **WHILE** loops. Each **FOR** loop needs to indicate three things: where to start, when to end, and how to increment. Sometimes we can gloss over these details; we don’t care how we loop through the array just so long as every element is reached once.

Clearly indicate the loop control variable

Pseudocode

```
FOR y ← 0 ... yMax
    FOR x ← 0 ... xMax
        frame[y][x] ← color
    FOR each ship in ships
        ship.draw()
```

Counting loops indicate the start and stop
Other loops just need to indicate the container

Figure 02.13:
The **FOR** keyword

Rule 02.14 Specify the loop control variable in the **FOR** loop

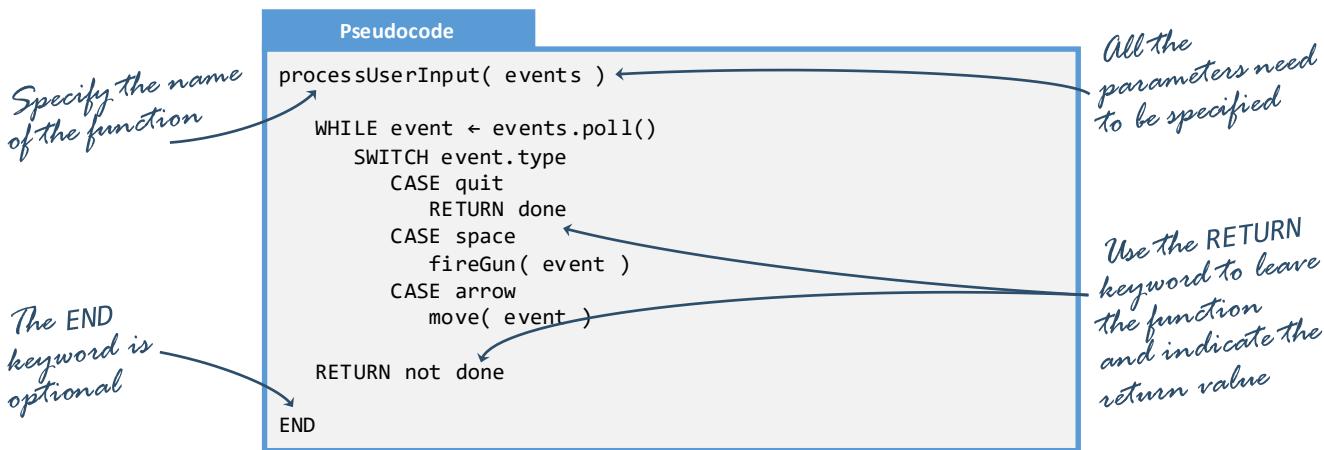
Each **FOR** loop needs to indicate the loop control variable. This could be an index, a counter, or an iterator. Regardless of the style of **FOR** loop we are representing, enough detail needs to be included that any reasonable programmer can convert the pseudocode into working code.

Functions

There are two parts to using functions in pseudocode: defining a function and calling a function.

Function Definition

Functions are represented with the minimal amount of syntax. The most important detail is that the actual function name and each parameter needs to be specified.



Note that we typically do not define classes or data structures with pseudocode. Those are represented using UML Class Diagrams and other design tools.

Figure 02.14:
Function definition

Calling a Function

A function is called in pseudocode exactly the same as it is called in a programming language: the name of the function is specified as are all of the parameters.

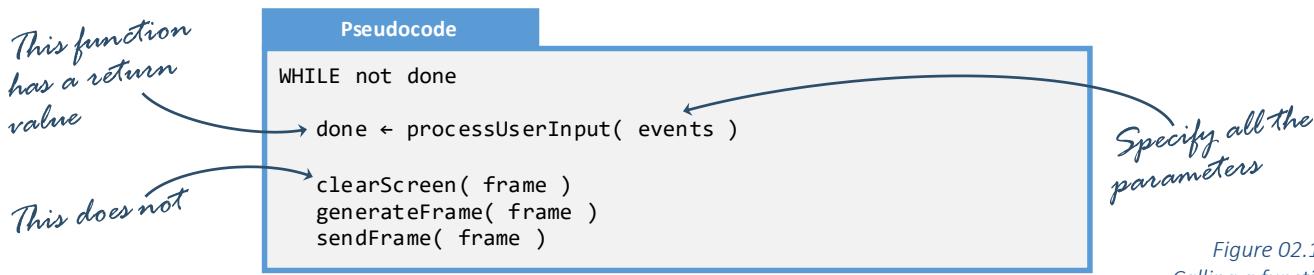


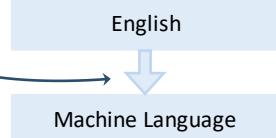
Figure 02.15:
Calling a function

Designing with Pseudocode

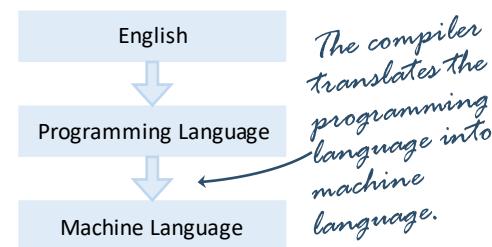
To understand the role of pseudocode in the code design process, it is useful to take a step back and look at the bigger picture. It is the job of the designer to take a high-level idea of how a function or program is to work, and produce from it working code.

In the early days of programming, this was a one-step process. The programmer would produce machine language on punch cards which would be executed directly by the computer.

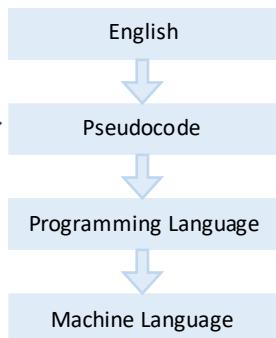
The programmer translates ideas from English into code.



A little later, programming languages were invented. This made it possible for the programmer to work at a higher level of abstraction leaving routine and unimportant details to the compiler. For example, the programmer can work with named variables rather than with memory locations. The programmer can use an IF statement rather than conditional jumps. The programmer can write a FOR loop or a function rather than coding all those things by hand. This was a huge boon for productivity and code quality. It also enabled the programmer to solve larger and more complex problems because she was not required to manage quite so many details. The compiler can handle them!

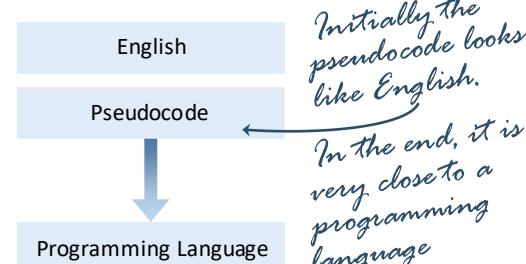


Pseudocode is a vehicle to help us move design ideas to a programming language



This approach worked well for many programmers and for many problems. Occasionally, however, the jump from the client's needs to the programming language was too great. Also, programming languages forced the designer to resolve a multitude of nit-pick details rather than address big design decisions. This is where pseudocode (and other design tools) enters the picture. Pseudocode is sufficiently abstract and high-level that the programmer can paint with broad brush strokes, putting off details until later. It relieves the programmer from the burden of worrying about the syntax of the language, data types, and other "bookkeeping" details that every programming language requires. Since pseudocode does not compile, all of these details can be ignored as the overall design is worked out. Once this is done, then the programmer can then go about translating the pseudocode into the syntax of the programming language of their choice.

Pseudocode is a vehicle transporting the designer's high-level ideas into programming constructs that can be readily converted into code. The key word here is "vehicle" because pseudocode helps with the movement of design across this English / programming language gulf. Initially ideas are expressed in English with little structure or detail. Then, through an iterative process, more details are added and more programming structure is introduced. With each iteration, pseudocode transports the designer's ideas from their English beginnings into constructs that look like code. Eventually all the English is removed and nothing but pseudocode keywords remain. At this point, the programmer is ready to write code.



Examples

Example 02.1: Bubble Sort

This example will demonstrate how to use pseudocode to discover an algorithmic solution to a complex problem.

Problem

Write the pseudocode to represent the Bubble Sort. The definition is the following:

The Bubble Sort is a comparison sorting algorithm that repeatedly swap adjacent elements if they are in wrong order, finding the largest item, then the second largest, and so on until the array is sorted.

Solution

We will start drafting the solution mostly in English and slowly work in details and pseudocode keywords. We are done when all the pseudocode rules are honored.

The first draft is mostly in English

Pseudocode
Repeatedly swap the adjacent elements if they are in wrong order, finding the largest item, then the second largest, and so on until the array is sorted .

Work out some details, such as "until the array is sorted"

The second draft is more detailed but still mostly English

Pseudocode
First, find the largest item in the array and put it at the end. Then find the next largest and put it at next spot. Continue until the list is sorted. Do this by starting at one side of the array and moving to the other. With each spot, swap the adjacent elements if they are in wrong order .

The third draft begins to introduce some structure

Pseudocode
For each spot (left to right), find the largest item and put it there. Find the largest by moving through the rest of the array, comparing each item with the one in the adjacent slot. If it is bigger than the target item, then swap.

Now many pseudocode keywords are added

Pseudocode
FOR iSpot ← end to beginning
 FOR iCheck ← beginning to iSpot
 IF array[iSpot] < array[iCheck]
 swap spot iSpot and iCheck

Notice how there are still some details left in English

We know we are done when all the pseudocode rules are obeyed

Pseudocode
FOR iSpot ← numItems ... 0
 FOR iCheck ← 0 ... iSpot - 1
 IF array[iSpot] < array[iCheck]
 swap(array[iSpot], array[iCheck])

Example 02.2: Compute Tax

This example will demonstrate how to use pseudocode to work through a confusing problem definition.

Problem

Write the pseudocode to compute an individual's tax burden:

Income range	Tax is
\$0 – \$15,100	10% of amount over \$0
\$15,100 – \$61,300	\$1,510 plus 15% of amount over \$15,100
\$61,300 – \$123,700	\$8,440 plus 25% of amount over \$61,300
\$123,700 – \$188,450	\$24,040 plus 28% of amount over \$123,700
\$188,450 – \$336,550	\$42,170 plus 28% of amount over \$188,450
\$336,550 – no limit	\$91,043 plus 35% of amount over \$336,550

Solution

We will start with an English draft of the second row of the table, convert that to pseudocode, and then work through the rest of the table.

The first draft is an English version of the 2nd row of the table.

Some parts are easy to translate to pseudocode

The whole 2nd row of the tax table is now pseudocode

With the hard part done, we can now apply the same pattern to the entire table

Pseudocode
If income is between \$15,100 and \$61,300 then the tax is \$1,510 plus 15% of the amount over \$15,100.

Pseudocode
IF $\$15,100 \leq \text{income} \leq \$61,300$
 $\text{tax} \leftarrow \$1,510 + 0.15 \times (\text{income} - \$15,100)$

Pseudocode
IF $\$15,100 \leq \text{income} \leq \$61,300$
 $\text{tax} \leftarrow \$1,510 + 0.15 \times (\text{income} - \$15,100)$

Pseudocode
computeTax (income)

IF $\$0 \leq \text{income} < \$15,100$
 $\text{tax} \leftarrow \text{income} \times 0.10$
IF $\$15,100 \leq \text{income} < \$61,300$
 $\text{tax} \leftarrow \$1,510 + 0.15 \times (\text{income} - \$15,100)$
IF $\$61,300 \leq \text{income} < \$123,700$
 $\text{tax} \leftarrow \$8,440 + 0.25 \times (\text{income} - \$61,300)$
IF $\$123,700 \leq \text{income} < \$188,450$
 $\text{tax} \leftarrow \$24,040 + 0.28 \times (\text{income} - \$123,700)$
IF $\$188,450 \leq \text{income} < \$336,550$
 $\text{tax} \leftarrow \$42,170 + 0.33 \times (\text{income} - \$188,450)$
IF $\$336,550 \leq \text{income}$
 $\text{tax} \leftarrow \$91,043 + 0.35 \times (\text{income} - \$336,550)$

RETURN tax
END

This bit is still clearly English

Make sure that all the pseudocode rules are honored

Exercises

Exercise 02.1: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Pseudocode can be compiled like any other programming language.	
Pseudocode is a high-level language.	
Pseudocode is drawn like any other graph, chart, or diagram.	
There is no structure or form to pseudocode.	
Pseudocode can easily be put in the comments of production code.	
Pseudocode has the same structure as a programming language.	

Exercise 02.2: Pseudocode Keywords

From memory, list the seven classes of pseudocode keywords. For each class, list the keywords themselves with a short example.

Class	Keywords

Exercise 02.3: Pseudocode Statements

Write the pseudocode corresponding to the following descriptions:

Description	Pseudocode
Display "Hello world"	
Prompt the user for his/her age	
Initialize a variable to zero	
Convert feet to meters	
Display "Mr." if male and "Mrs." if female	
Count from 1 to 10.	
Call the function "show" with the value 42.	

Exercise 02.4: Pseudocode Errors

Identify all the errors in the following pseudocode. If possible, list the rule number and description corresponding to each error.

Pseudocode

```
float computeTax (float tax rate)

while (!done)
{
    PROMPT for hourlyWage
    PROMPT for hoursWorked

    IF hourlyWage < 0 or hoursWorked < 0
        PUT error message on the screen
    ELSE
        done ← true
}

SET pay ← half pay for hours over forty plus regular pay

Determine if the user is eligible for a bonus

RETURN pay
END
```

Problems

Problem 02.1: Currency Conversion

Write the pseudocode corresponding to the following code:

Java

```
public static void main( String[ ] args )
{
    // prompt for the money in dollars
    double moneyDollars = getDollars();

    // perform the conversion
    double moneyEuros = 0.873 * moneyDollars;

    // display the output
    System.out.printf("€%.2f\n", moneyEuros)

    return 0;
}
```

Problem 02.2: Display Negative Balances

Write the pseudocode corresponding to the following code:

VB

```
Sub displayNegative(ByVal ParamArray balances() As Double)
    ' Loop through the array looking for negative values
    For i As Integer = 0 To UBound(balances, 1)
        If balances(i) < 0.00 Then
            Console.WriteLine("ERROR: " & balances(i))
        End If
    Next i
End Sub
```

Problem 02.3: Leap Year Decision

Write the pseudocode representing a program to compute whether a given year is a leap year. The logic for a leap year is the following:

According to the Gregorian calendar, which is the civil calendar in use today, years evenly divisible by 4 are leap years, with the exception of centurial years that are not evenly divisible by 400. Therefore, the years 1700, 1800, 1900 and 2100 are not leap years, but 1600, 2000, and 2400 are leap years.

If you have completed Problem 01.1, compare your pseudocode with your flowchart. If not, create also a program in the language of your choice matching your pseudocode.

Problem 02.4: Linear Search

Write the pseudocode representing a linear search. The logic for the search is the following:

A linear search is a method used to determine whether a given value is in an array. There are three inputs: the array containing elements which may or may not match the searched value, the value to be searched, and the number of elements in the array. The algorithm begins by comparing the search value against the first element in the array. If the first element in the array matches the search value, then terminate the search and display a “found” message. If it is not a match, then proceed to the second item. Continue until either the element is found or the end of the array is reached. If this is the case, then display a “not found” message and end the program.

If you have completed Problem 01.2, compare your pseudocode with your flowchart. If not, also create a program in the language of your choice matching your pseudocode.

Problem 02.5: Ship Upgrade

Write the pseudocode to represent a program to compute whether a given ship in a video game should be upgraded to the next level. The logic is the following:

A ship advances to level three if the following conditions are met: the ship is currently at level two, the ship has accumulated 200 points, and the ship has been at level two for at least one minute.

Problem 02.6: List Display

Write the pseudocode to represent a program to display the contents of a given to-do list. The logic for this program is the following:

Advance through all the list items in the program. If a list item is of type "to-do," then display that item to the user. Otherwise, skip that item and continue through the rest of the list.

Problem 02.7: Login

Write the pseudocode to represent a program matching the following problem definition:

Prompt the user to see if he/she has a username. If not, then prompt them for a username and password and store both in our authentication dictionary. Next, prompt the user for his/her username and password. If there is a username and password in the authentication dictionary, then display "you are authenticated." Otherwise, display "Invalid credentials" and prompt them again.

When finished, implement this algorithm in the programming language of your choice.

Problem 02.8: Point Tabulation

Write the pseudocode corresponding to the following flowchart:

This program has a list of points, each point has a value and a Boolean indicating whether it is to be included in the score. Iterate through the list, adding it to the total if the Boolean value is true. When the end of the list is reached, then display the total.

Also create a program in the language of your choice matching your pseudocode.

Problem 02.9: Authentication from a File

Write the pseudocode to represent a program to authenticate a user. The algorithm for the program is the following:

The program read a collection of usernames and passwords from a JSON file. The program will then prompt the user for his/her username and his/her password. If the provided username/password pair have a corresponding entry in the JSON file, then indicate that he/she is authenticated. Otherwise, indicate that they are denied access.

Also create a program in the language of your choice matching your pseudocode.

Problem 02.10: Guessing Game

Write the pseudocode to represent a guessing game program. The logic for the game is the following:

The program will prompt the user for a positive integer. The program will then randomly generate a number between 1 and that number. The user will then be prompted to guess the number. If the number is too high or too low, a message to that effect will be displayed. That number will then be stored in a list and the user will be prompted again. If the user guesses the number, then the number of guesses and all the attempts will be displayed to the user.

Also create a program in the language of your choice matching your pseudocode.

Challenges

Challenge 02.1: Advanced Search

Write the pseudocode for the following advanced search algorithm:

Consider a person looking up the word "glitch" in a dictionary. At first, the word could be in any of the 688 pages in the dictionary. The user cuts the dictionary in half (to page 344) and notices the word "magnet" at the top of the page. This word sorts after "glitch." Because we can assume that the dictionary is sorted, we can deduce that the word "glitch" must be between pages 1 and 343. The user then cuts the remaining pages in half (to page 171) and finds the word "drift" on that page. Since "drift" sorts before "glitch," we can rule out pages 1 through 171. Thus, the possible pages must be 172 through 343. The user then cuts to page 257 and finds the word "guidance." Since "guidance" sorts after "glitch," we can rule out pages 257 through 343. This leaves pages 172 through 256. This process continues until the page containing "glitch" is located.

Your program must do the following:

- Prompt the user for a file containing a collection of words
- Read the file
- Prompt the user for a word to search for
- Perform the advanced search described above to locate the word
- Display a message indicating whether the word exists in the list

Challenge 02.2: Child Tax Credit

Research on the internet how the "child tax credit" works with the United States Internal Revenue Service (IRS). Specifically, we are looking for the 2017 rules. You may use your flowchart from Challenge 01.1 if you have done this previously.

Write the pseudocode representing a program computing a user's child tax credit.

If you have completed Challenge 01.1, compare your pseudocode with your flowchart. If not, create also a program in the language of your choice matching your pseudocode.

Efficiency

Chapter 03

Algorithmic efficiency, called Big-O, is a measurement of how program execution time is related to input size.

Algorithmic efficiency is how algorithm execution time is related to input size

One of the most important measures of the quality of a program is performance, or how long it takes to produce the desired output. It may seem that there is an easy way to make this measurement: just use a stopwatch! If only it were that easy! There are so many factors affecting execution time that have nothing to do with the quality of the underlying algorithm (such as performance of the computer, the other processes running, the state of the operating system, and others). Clearly, we need a better metric. Algorithmic efficiency, called Big-O, is a measurement of how program execution time is related to input size. In other words, if we were to graph input size to execution time, what would be the shape of the resulting graph?

Perhaps this is best explained by example. If it takes a blink of an eye (0.1 seconds if you are very quick!) to look up a record in a database, then it will take 2 seconds to perform 20 lookups. If we were to perform 40 lookups, then it should take 4 seconds. Since the cost is directly proportional to the size, we call this relationship linear. Thus, if the number of lookups is “ n ,” then the cost of finding an element is $O(n)$ (pronounced “Big-O of N”) which means roughly “ n times some constant.” The most common performance characteristics of a function are the following:

Notation	Name	Description
$O(1)$	Constant	Performance is unrelated to the amount of input into the algorithm
$O(\log n)$	Logarithmic	Performance is a log of the amount of input into the algorithm
$O(n)$	Linear	Performance is directly related to amount of input into the algorithm
$O(n \log n)$	N-Log-N	Performance is linear times logarithmic
$O(n^2)$	N-Squared	Performance is the square of the amount of input into the algorithm
$O(2^n)$	2-to-the-N	Performance is really bad

When working with an algorithm that takes a large or variable amount of input, then algorithmic efficiency needs to be a central consideration. As a general rule, it is worthwhile to go to extreme lengths to reduce the Big-O cost of an algorithm (such as moving from $O(n^2)$ to $O(n \log n)$).

To get an idea of how these performance characteristics relate to each other, consider a dataset containing 1,000 elements. In this dataset, we can select between a constant time lookup algorithm $O(1)$, a logarithmic algorithm $O(\log n)$, a linear algorithm $O(n)$, and so on. In each case, the time per lookup takes the same (0.1 seconds or a blink of an eye), but the number of lookups required to find the needed element changes according to our selected algorithm. Based on this, how long would it take to complete each?

Notation	Time	Performance Implications
$O(1)$	0.1 second	Faster than most people would notice. Essentially instantaneous
$O(\log n)$	One second	A noticeable lag
$O(n)$	Minute and a half	Takes considerable patience. Many users will give up and think it will not finish
$O(n \log n)$	14 minutes	The user will leave the computer and take a walk
$O(n^2)$	A day	The user lost all hope
$O(2^n)$	2 trillion times the age of the universe	

Clearly the constant algorithm $O(1)$ is much faster than $O(2^n)$. Next, consider what would happen if our database lookup took us twice as long to perform (0.2 seconds). How will this affect performance?

Notation	Time	Performance Implications
$O(1)$	0.2 seconds	Faster than most people would notice. Essentially instantaneous
$O(\log n)$	Two seconds	A noticeable lag
$O(n)$	Three minutes	Takes considerable patience. Many users will give up and think it will not finish
$O(n \log n)$	28 minutes	The user will leave the computer and take a walk
$O(n^2)$	Two days	The user lost all hope
$O(2^n)$	4 trillion times the age of the universe	

Doubling the cost of a database lookup would result in a noticeable change; everything would take twice as long. However, if we used an algorithm that is $O(n^2)$ as opposed to $O(n \log n)$, then the performance implications would be dramatic (56 hours for $O(n^2)$ as opposed to 28 minutes for $O(n \log n)$). Thus, the exponent of the performance characteristic (meaning $O(1)$ vs. $O(n)$ vs. $O(n^2)$) is much more important than the duration of a single event (0.2 seconds per operation vs. 0.1 seconds).

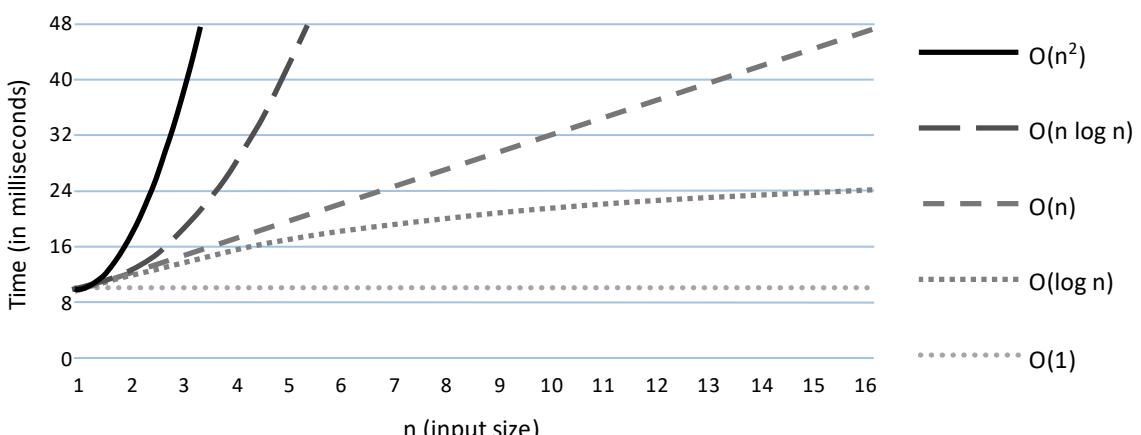


Figure 03.1:
Efficiency of various
algorithms

Levels of Performance

While it is always worthwhile to increase the overall efficiency of an algorithm, the algorithmic efficiency is a much more important consideration when working with large or variable size input.

O(1) Efficiency

O(1) is also known as constant performance. This means that the execution time of an algorithm is unrelated to the size of the input. To put it in more understandable terms, doubling the size of the input should have no impact on the algorithm execution time. O(1) is the gold standard of algorithmic efficiency; it is the goal we should all strive for.

Doubling the size of the input has no impact on execution time

We can graph the performance of an algorithm by putting the cost on the vertical axis and the input size on the horizontal axis. The cost can be measured in time, number of iterations, or a variety of similar metrics. We call this graph a performance curve. The performance curve of an O(1) algorithm is the following:

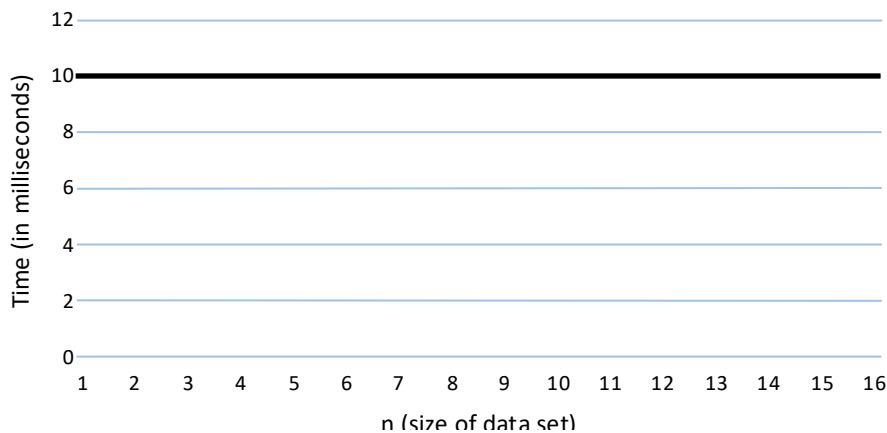


Figure 03.2:
O(1) efficiency

An O(1) performance curve is a flat horizontal line

In the above graph, we can see that there is a constant cost to executing this function: 10ms. This cost is unchanged regardless of the size of the input; $n=1$ takes exactly as long to execute as $n=16$. Thus, the defining characteristic of constant performance is a horizontal graph. Constant performance can be described with the following equation:

$$\text{cost} = c_1$$

Here, c_1 refers to some constant which is not influenced by the size of the input.

Recognizing O(1)

Any algorithm whose execution time is independent of input size is O(1). This means that one can determine whether an algorithm is O(1) by carefully tracking how input parameters are used. For example, consider the following code.

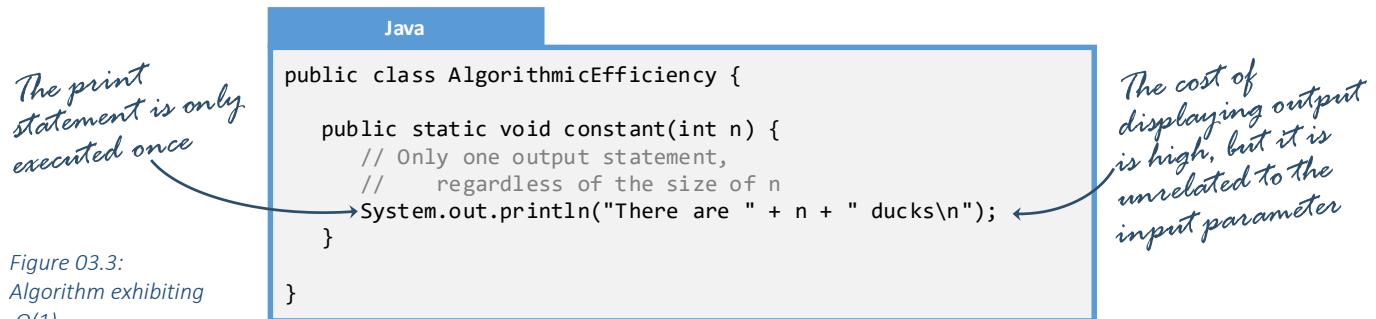


Figure 03.3:
Algorithm exhibiting
O(1)

In this example, the cost displaying text is rather high. However, the cost is the same regardless of the amount of the value of n . This is the defining characteristic of O(1) algorithms.

Tips for recognizing O(1) algorithms include the following:

- **Constant Input.** If an algorithm takes no input or if the input does not directly affect execution time, then it is O(1).
- **No Loops.** If an algorithm has no loops, then it is a good indication that it is O(1). Note that this is not a hard-and-fast rule; sometimes a loop can be hidden in a function that is called or a library that is used.
- **Only Using Constant-Length Loops.** The existence of a loop does not automatically disqualify an algorithm from being constant-cost. The important thing is that the number of executions of the loop needs to be independent of the input.

O(log n) Efficiency

$O(\log n)$ is also known as logarithmic performance. This means that the execution time is a logarithmic function of the input size. To put it in more understandable terms, doubling the size of the input should slow execution by one time unit. For very small input sizes, there is a noticeable performance change as the input n increases. However, when n becomes relatively large, changes in input size have no perceivable impact on performance.

Doubling the size of the input should slow execution by 1 time unit

The performance curve of an $O(\log n)$ algorithm is the following:

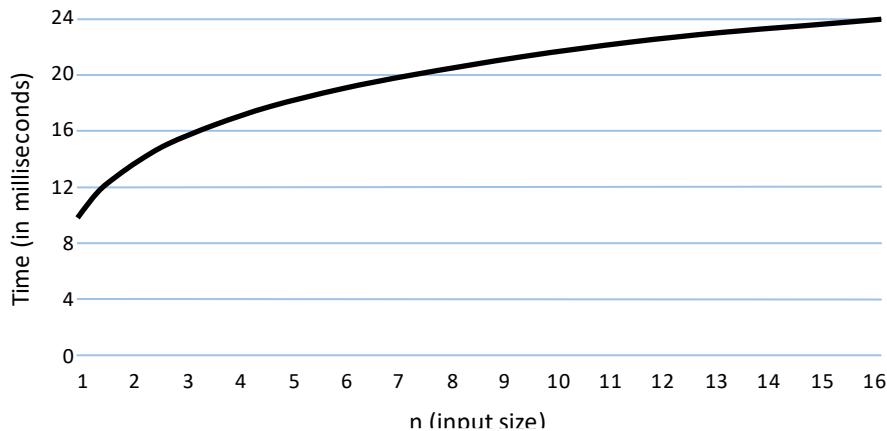


Figure 03.4:
 $O(\log n)$

An $O(\log n)$ performance curve starts diagonal and curves down to a nearly horizontal line

The defining characteristic of a logarithmic performance graph is that the slope of the curve gently decreases until it becomes almost horizontal for large values of n . The graph curves down until it becomes nearly flat. Logarithmic performance can be described with the following equation:

$$\text{cost} = c_1 \log_2 n + c_2$$

Here, c_1 refers to some constant which is not influenced by the size of the input, n is the size of the input, and c_2 is some other constant.

Logarithmic performance (and indeed everything but constant) does relate execution time to dataset size. However, the rate of increase slows down as the data set gets larger. Many people find it difficult to internalize logarithmic growth. Consider this equation:

$$2^n = m \xrightarrow{\text{yields}} \log_2 m = n$$

Thus, exponents and logarithms are inverses of each other. Since $2^{32} = 4,000,000,000$, then $\log_2 4,000,000,000 = 32$. This means it will only take 32 iterations through the logarithmic algorithm to process an input of 4 billion. Logarithmic algorithms are very efficient, even for extremely large datasets.

Recognizing O(log n)

All logarithmic algorithms skip through half of the input with every iteration of a loop.

The figure shows a Java code snippet within a blue-bordered box. The code defines a class `AlgorithmicEfficiency` with a static method `logarithmic`. The method prints "Duck!\n" and "Goose!\n" repeatedly, with the number of iterations determined by the input `n`. A handwritten note on the left side of the box states: "We output Duck O(Log n) times, and Goose O(1) times". Another handwritten note on the right side says: "The sequence of values is: 1, 2, 4, 8, 16, ...". Arrows from the notes point to the relevant parts of the code: one arrow points to the loop counter `i *= 2`, and another points to the value `1, 2, 4, 8, 16, ...`.

```
Java
public class AlgorithmicEfficiency {
    public static void logarithmic(int n) {
        // We do not count by 1's, but
        // double our counter with every iteration
        for (int i = 1; i <= n; i *= 2) {
            System.out.println("Duck!\n");
            System.out.println("Goose!\n");
        }
    }
}
```

Figure 03.5:
Algorithm exhibiting
 $O(\log n)$

Notice the index `i` does not increment by 1s, but instead doubles with every iteration. The sequence is 1, 2, 4, 8, 16, 32, 64, 128, ... instead of 1, 2, 3, 4, 5, 6, 7, 8. The same would be true if we counted down: 100, 50, 25, 12, 6, 3, 1. In both cases, if one were to pass a value of `n` that is twice as large, we would only go through the loop one additional iteration. This is the distinguishing characteristic of logarithmic algorithms.

Tips for recognizing $O(\log n)$ algorithms include the following:

- **Loop.** There must be a loop controlled by the input in some way. Note that this loop could be hidden in recursion or it could be in a function that our code calls.
- **Subdivision.** Every iteration of the loop must cut the size of the input by a fraction. In almost all cases, that fraction is a half resulting in a $\log_2 n$ equation. If the loop cuts the size by a third, then the resulting algorithm is still logarithmic but the equation is $\log_3 n$.

O(n) Efficiency

O(n) is also known as linear performance. This means that the execution time is directly related to input size. To put it in more understandable terms, doubling the size of the input should double execution time. We usually see linear performance when every element in an input buffer is visited a constant number of times. For this reason, it is among the most common characterizations of an algorithm. The performance curve of an O(n) algorithm is the following:

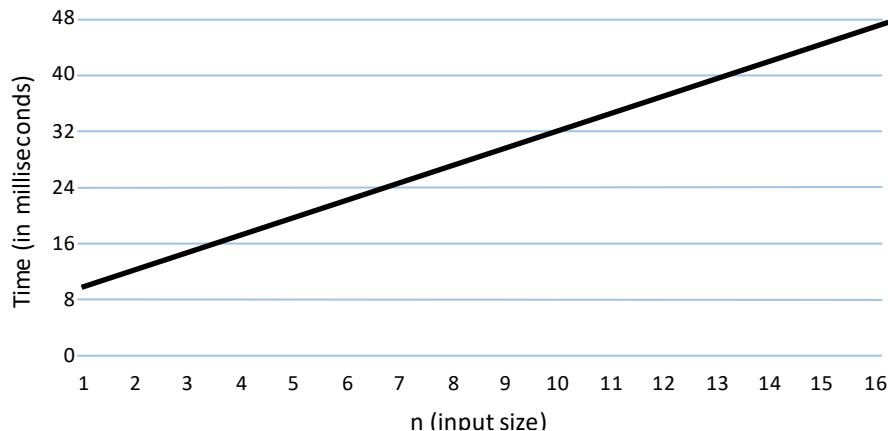


Figure 03.6:
O(n), linear

An O(n) performance curve is a straight diagonal line

The defining characteristic of linear performance graphs is that the slope remains constant for all values of n . The steepness of the slope is a function of how much work is done on each element of the input buffer. Many linear performance curves do not hit zero when the input size is zero. This is because the algorithm performs a fixed amount of work regardless of the size of the input buffer.

Linear performance can be described with the following equation:

$$\text{cost} = c_1 n + c_2$$

Here, c_1 refers to some constant which is not influenced by the size of the input, n is the size of the input, and c_2 is some other constant. It is not uncommon for an algorithm to contain both linear and logarithmic components. In this case, the performance equation might be:

$$\text{cost} = c_1 n + c_2 \log_2 n + c_3$$

For small n values, it might be that the logarithmic component of the equation dominates the value of cost. However, as n gets larger, then the logarithmic component becomes less and less important. To illustrate this point, consider the case where $c_1=10$ and $c_2=1,000$. Here, the logarithmic part of the algorithm is a hundred times slower than the linear part of the equation. When n is small (say $n=8$) then this is the most important part: $\text{cost} = 10 \times 8 + 1,000 \times 3$ (because $\log_2 8 = 3$). However, when n is large (say $n=1,000,000$), then things are different: $\text{cost} = 10 \times 1,000,000 + 1,000 \times 20$ (because $\log_2 1,000,000 = 20$). For this reason, only the highest order term (the term with the largest exponent) matters in Big-O calculations.

Only the highest order term matters
in Big-O calculations

Recognizing O(n)

Linear algorithms are characterized by a loop where every element in the collection is visited once. Note that sometimes a library or a feature of the programming language may obscure this loop, but the loop must still exist.

Consider the following algorithm to loop through all the values between 1 and n:

The number of Ducks is directly related to the value of n

```
Java
public class AlgorithmicEfficiency {
    public static void linear(int n) {
        // We count by 1's. If the user passes in n,
        // we count from 1 to n
        for (int i = 1; i <= n; i++) {
            System.out.println("Duck!\n");
            System.out.println("Goose!\n");
        }
    }
}
```

The sequence of values is:
1, 2, 3, 4, 5...

Figure 03.7:
Algorithm exhibiting $O(n)$

Observe how we will display exactly n instances of “Duck”. If we double the value of n , one would expect twice as many “Duck”s. This is the defining characteristic of linear algorithms. Note that if we did not count by 1s but rather by 2s, the same thing would be true. If $n = 100$, we would get 50 “Duck”s and if $n = 200$, we would get 100. In other words, we would still double the number of “Duck”s if the input parameter is twice the size.

Tips for recognizing $O(n)$ algorithms include the following:

- **Loop.** There must be a loop controlled by the input in some way. Note that this loop could be hidden in recursion or it could be in a function that our code calls.
- **Every element visited.** In most cases, linear algorithms visit every element in the input buffer. However, there are exceptions. An algorithm that visits every other element would still be $O(n)$ but the equation would be cost = $\frac{1}{2}n$.

O($n \log n$) Efficiency

$O(n \log n)$ is commonly called “N-Log-N.” This is the same thing as saying “for each element in the input, perform a logarithmic operation.” Thus, $O(n \log n)$ is the same thing as $O(n) \times O(\log n)$. The performance curve for N-Log-N begins with an arc looking much like $O(\log n)$.

However, when n gets larger, the $O(n)$ component of the equation dominates and the curve becomes nearly straight.

Doubling the size of the input should slow execution by double, plus one time unit

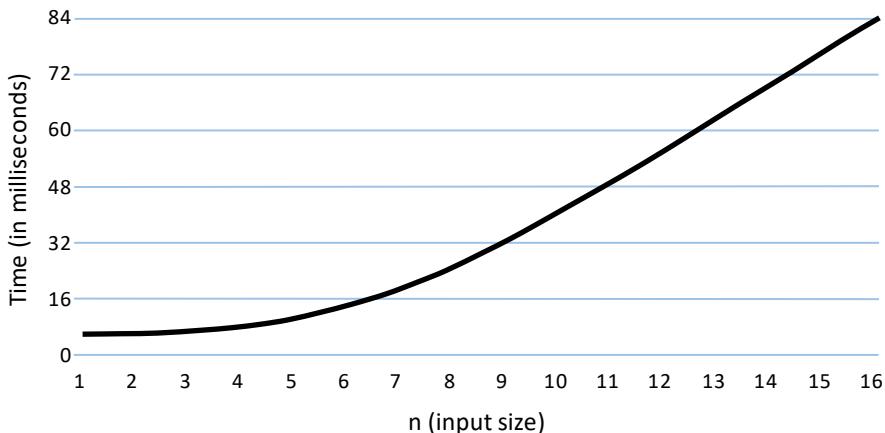


Figure 03.8:
 $O(n \log n)$

An $O(n \log n)$ performance curve is an upwardly facing arch that becomes nearly straight for large values of n

The defining characteristic of N-Log-N performance graphs is that the slope gradually increases until it approaches a constant slope. It curves up until it becomes nearly a straight diagonal line for large ns . If you only view the performance curve for large ns , then it will be easy to miss the curve at the bottom of the graph and mistake it for $O(n)$.

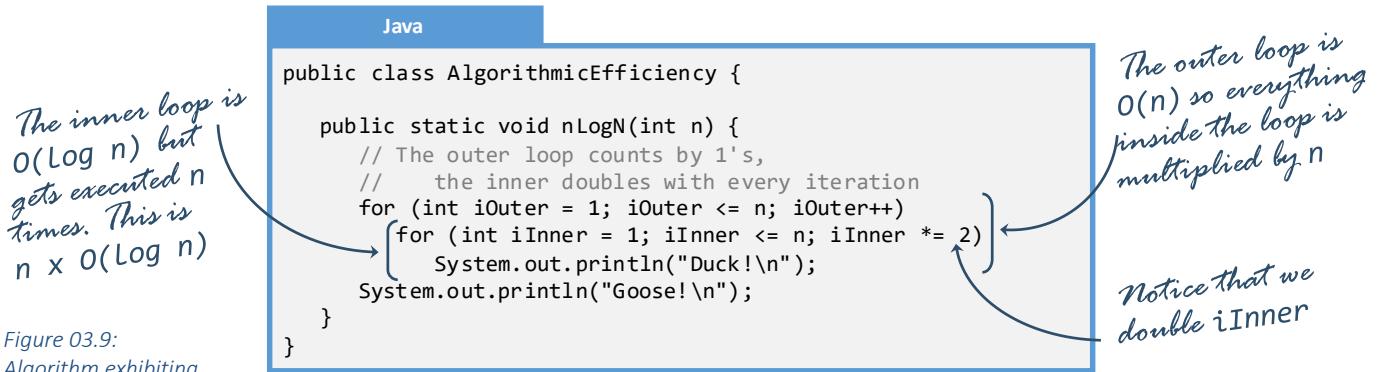
N-Log-N performance can be described with the following equation:

$$cost = c_1(n \log_2 n) + c_2$$

As before, c_1 refers to some constant which is not influenced by the size of the input and n is the size of the input. There can be any number of lower order terms after represented by c_2 . When n becomes large, however, only the highest order term has a significant impact on the cost equation.

Recognizing $O(n \log n)$

There are two ways that we can have $O(n \log n)$ efficiency: if a linear operation is done a logarithmic number of times, or if a logarithmic operation is done a linear number of times. Consider the following code:



This algorithm is quite a bit more complex than the previous ones. Note that the outer loop is $O(n)$ because it counts by 1's, visiting every number between 1 and n . The inner loop is $O(\log n)$ because it doubles with every iteration. If the algorithm first completed the outer loop and then the inner loop, it would be $O(n) + O(\log n)$ which would be $O(n)$. However, since the inner loop is within the outer loop, we complete an inner loop with every iteration of the outer loop. This means we multiply the two together. Thus, we get $O(n) \times O(\log n)$ which is $O(n \log n)$.

We would get the same results if we flipped the loops. If the outer loop was $O(\log n)$ and the inner loop was $O(n)$, we would get $O(\log n) \times O(n)$ which is still $O(n \log n)$. In other words, the commutative property of multiplication means that it does not matter which loop is “fast” and which is “slow.”

Tips for recognizing $O(n \log n)$ algorithms:

- **Two loops.** There must be two loops in the algorithm which are both controlled by the size of the input.
- **Nested loops.** One loop must be in the body of the other loop. If two loops are arranged sequentially (beside each other), then the algorithm is $O(n)$.
- **One loop must visit every element.** One of the two loops must visit every element (or a constant fraction of the set of every elements).
- **One loop must subdivide the input.** As with $O(\log n)$, there must be a subdivision of the input range with every iteration of the loop.

$O(n^2)$ Efficiency

$O(n^2)$ is commonly called “N-Squared.” This is the same thing as saying “for each element in the input, perform a linear operation.” Thus, $O(n^2)$ is the same thing as $O(n) \times O(n)$.

Doubling the size of the input should quadruple execution time

There are times when the cost of performing an action becomes much worse as n increases. There are examples in nature. For example, the wind resistance on a car increases as a square of the speed. If you go twice as fast, it takes four times the amount of force to displace the air. As a rule, programmers try to avoid algorithms that exhibit such a behavior.

The performance curve of an $O(n^2)$ algorithm is the following:

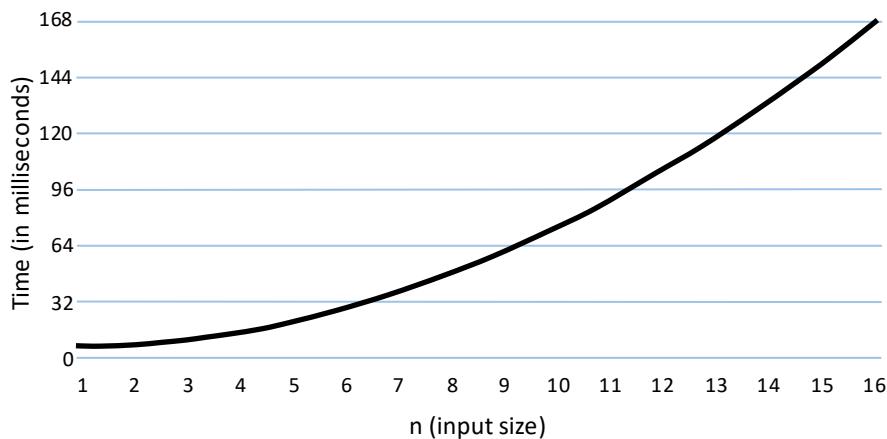


Figure 03.10:
 $O(n^2)$

An $O(n^2)$ performance curve arcs up into an ever-steeper parabola

The defining characteristic of N-Squared performance curves is that the slope increases throughout the graph. No matter how the graph is viewed for any choice of n , the curve has a parabolic shape.

N-Squared performance can be described with the following equation:

$$\text{cost} = c_1 n^2 + c_2$$

As with other algorithmic performance equations, the lower order terms are less important. To emphasize this point, a well-known sorting algorithm has a performance equation identical to Gauss’ equation: $\text{cost} = \frac{(n-1)n}{2}$. When you expand this equation, it becomes: $\text{cost} = \frac{1}{2}n^2 - \frac{n}{2}$. Observe how the n^2 component of the equation absolutely dominates the calculation when n reaches 100. By 10,000, the $\frac{1}{2}n$ part has almost no significance. It is for this reason that we only look at the highest-order polynomial when making Big-O determinations.

n	$\frac{1}{2}n^2$	$\frac{1}{2}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$
2	2	1	1
10	50	5	45
100	5,000	50	4,950
1,000	500,000	500	499,500
10,000	50,000,000	5,000	49,995,000

Recognizing $O(n^2)$

Whenever there is a loop within a loop, we have a prime candidate for $O(n^2)$ algorithm. The outer loop must be $O(n)$, usually meaning it visits every element in the input buffer. The inner loop must also be $O(n)$. To see an example of this code in action, consider the following code:

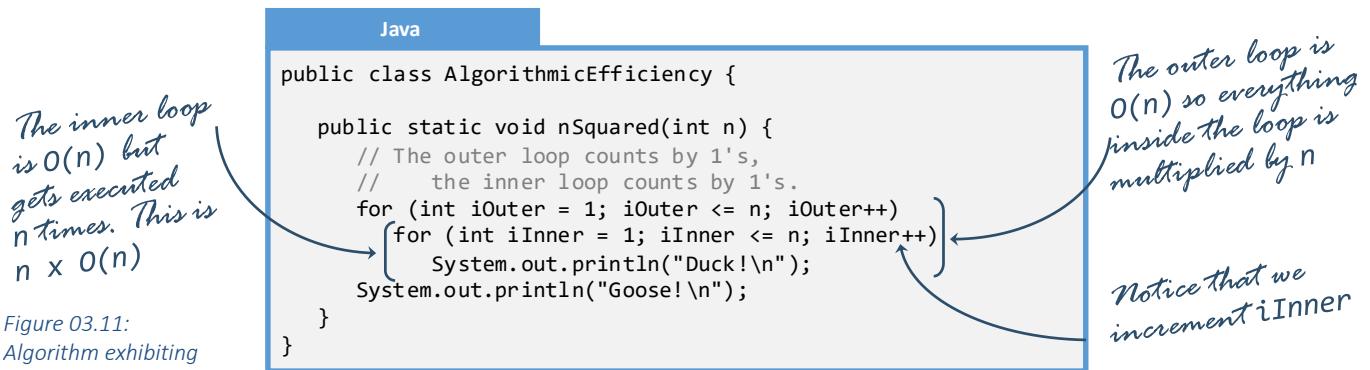


Figure 03.11:
Algorithm exhibiting
 $O(n^2)$

Notice that the outer loop characterized by the variable `iOuter` visits every number between 1 and n . If the function had no additional loops, we would call this a linear algorithm. However, a second loop also exists. This second loop also visits every element between 1 and n , making it $O(n)$ as well. Because the inner loop is within the outer loop, the total cost of this algorithm is $O(n) \times O(n)$ otherwise known as $O(n^2)$.

Tips for recognizing $O(n^2)$ algorithms include the following:

- **Two loops.** There must be two loops in the algorithm which are both controlled by the size of the input.
- **Nested loops.** One loop must be in the body of the other loop. If two loops are arranged sequentially (beside each other), then the algorithm is $O(n) + O(n)$, or $O(n)$.
- **Both loops must visit every element.** Each of the two loops must visit every element (or a constant fraction of the set of every elements).

O(2^n) Efficiency

There are many levels of algorithmic efficiency above O(n^2). Some of the more common ones include O($n^2 \log n$), O($n!$), and O(n^3). The performance of each of these is very bad for datasets larger than 10 or 20. In other words, if you calculate the algorithmic efficiency as worse than O(n^2), then there is probably some unnecessary computation that should be avoided. You've got a bug!

To illustrate how easy it can be to create an algorithm with truly horrid performance, consider the following code:

Ruby

```
def fibonacci(n)
  n <= 1 ? n : fibonacci( n - 1 ) + fibonacci( n - 2 )
end
```

Figure 03.12:
O(2^n) efficiency

This code has no loops so it should be O(1), right? However, notice that the function `fibonacci()` calls itself to compute the answer when $n > 1$. This is called recursion, a topic for a later chapter (Chapter 16 Strategy: Recursion). If one were to call this function with `fibonacci(6)`, then that would be the same thing as `fibonacci(5) + fibonacci(4)`. Since both 5 and 4 are greater than 1, we would need to call the function again. The full expansion is the following where we use `f()` instead of `fibonacci()` for brevity.

1. $f(6)$
2. $f(5) + f(4)$
3. $f(4) + f(3) + f(3) + f(2)$
4. $f(3) + f(2) + f(2) + f(1) + f(2) + f(1) + f(1) + f(0)$
5. $f(2) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1) + f(1) + f(0) + f(1) + f(0)$
6. $f(1) + f(0) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1) + f(1) + f(0) + f(1) + f(0)$
7. $1 + 0 + 1 + 0 + 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 + 0 = 8$

So how many times do we call the function? It turns out, it is O(2^n). If you convert this into a binary tree, you can see it more clearly:

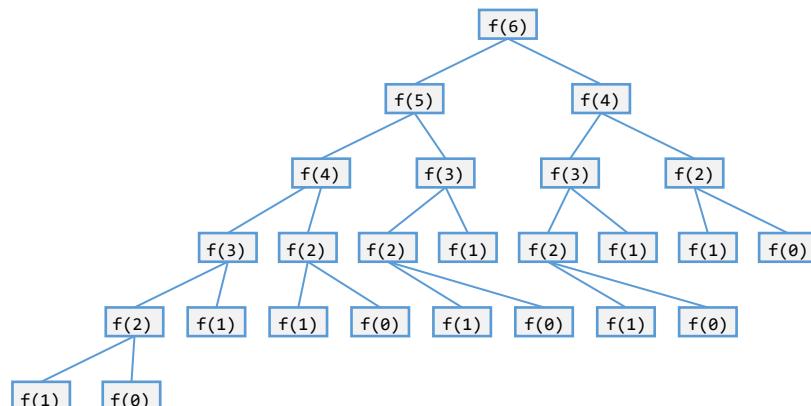


Figure 03.13:
O(2^n) tree

From this graph, we can see that the height of the tree is the input size and, since it is a binary tree, the number of nodes is roughly 2^n . Now, several of the branches are not completely filled out, but it is close enough to count as O(2^n). In other words, the equation for the number of nodes is $\text{num} = 2^n - \text{something}$, where *something* is a lower-order polynomial. We only take into account the highest order polynomial when computing Big-O algorithmic efficiency.

Instrumentation

We should make algorithmic efficiency determinations with every algorithm we create. This is especially true when the size of the input can be large. However, how can one be sure that our calculations are correct? In other words, it would be great if we could flip to the back of the book and verify that our predicted Big-O value is a true reflection of our algorithm. It turns out, we can do this through instrumentation.

Instrumentation is the process of adding counters to an existing algorithm which does not change the functionality of the task being measured. Instead, instrumentation measures how the task was performed.

For example, consider a function that inverts the order of elements in an array. Our algorithmic efficiency calculations suggest that this should be accomplished in $O(n)$. To confirm this, we add a counter inside the function to tabulate how many times the body of the loop is executed. If the algorithm is in fact linear, then one would expect that doubling the size of the input would result in double the value for our counter. To be sure, we will instrument our code. This is done in three steps:

1. **Add instrumentation code.** Add special code to count the number of times a key part of the algorithm is executed.
2. **Generate instrumentation reports.** Have the code create a table where each row corresponds to a single run and the columns track the input size versus the execution time.
3. **Interpret the results.** Compare the results against the characteristics of the various levels of algorithmic performance.

Back to our scenario, our generated instrumentation report looks like the following:

Output	
n	c
8	28
16	120
32	496
64	2016

Figure 03.14:
Instrumentation output

This is different than we expect. Our counter does not double when the input size doubles, but rather quadruples. This looks like $O(n^2)$. Just to be sure, we gather the four pieces of data and graph them.

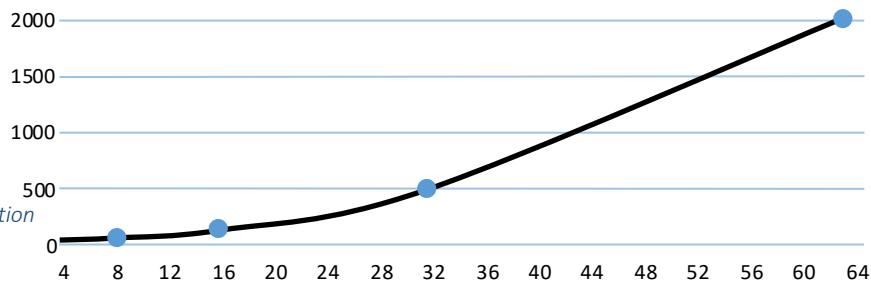


Figure 03.15:
Graphing instrumentation
output

As we suspected, the resulting curve has the parabolic shape of an $O(n^2)$ curve.

Adding Instrumentation Code

Whenever adding instrumentation code or any other debug code designed to gain insight into how an algorithm works, an important goal is to avoid impacting the outcome of the program. In other words, the instrumentation code should only count how often a given line of code is executed. It should not change anything else.

For example, consider a function reversing the elements in an array:

Figure 03.16:
Code before
instrumentation

```
C++
void reverse(int array[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int tmp = array[n - 1];
        for (int j = n - 1; j > i; j--)
            array[j] = array[j - 1];
        array[i] = tmp;
    }
}
```

How many times will this line of code get executed?

The counter is added here, in the body of the inner-most loop.

Figure 03.17:
Simple instrumentation

```
C++
void reverse(int array[], int n)
{
    int cIter = 0;

    for (int i = 0; i < n; i++)
    {
        int tmp = array[n - 1];
        for (int j = n - 1; j > i; j--)
        {
            array[j] = array[j - 1];
            cIter++;
        }
        array[i] = tmp;
    }
    cout << "n=" << n << ", cIter=" << cIter << endl;
}
```

The counter is added here, in the body of the inner-most loop.

Make sure the cIter variable is initialized to zero.

Display the results when the algorithm is finished

Figure 03.18:
Simple instrumentation
output

A quick inspection of this code reveals that **cIter** is only used three times: it is declared and initialized, it is incremented, and the results are displayed to the user. We therefore achieve our goal of not impacting the rest of the program. When we execute the code, we also realize that something unexpected happened:

Output
n=10, cIter=45
n=20, cIter=190

Notice that doubling the size of the input roughly quadruples the value of **cIter**. In other words, this looks more like an $O(n^2)$ than the $O(n)$ algorithm that was expected.

Even though we visually inspected our instrumentation code, it would be better if we could offer a guarantee that it has no impact on the underlying code. One way to accomplish this is to surround the code in a macro that is removed in the shipping code. This is easy to accomplish in C++:

The diagram shows a C++ code block with annotations. On the left, a callout points to the first section of code with the text: "If INSTRUMENT is defined, then we use these macros". On the right, another callout points to the second section of code with the text: "If INSTRUMENT is not defined, then this code is compiled into nothing". The code itself is as follows:

```

C++

```

#define INSTRUMENT
int inst_counter = 0;
#define inst_increment() inst_counter++
#define inst_reset() inst_counter = 0
#define inst_display(n) cout << "n=" << n << ", c=" \
 << inst_counter << endl;

#else // !INSTRUMENT
#define inst_increment()
#define inst_reset()
#define inst_display(n)
#endif

```


```

Figure 03.19:
Instrumentation macros

This is one of those rare times when using a global variable is justified. The global variable does not manifest itself in shipping code (because the **INSTRUMENT** macro is not defined in code compiled for the customer) and it is only accessed through these three macros.

Now, we can add and remove our instrumentation code much more easily:

The diagram shows a C++ code block for a `reverse` function with annotations. A callout on the left points to the `inst_reset()` call with the text: "Rather than accessing cIter directly, we use a macro.". Another callout on the right points to the `inst_display(n)` call with the text: "Here we access the counter with inst_display(n)". A third callout on the right points to the `inst_reset()` call with the text: "Here we use inst_reset() rather than int cIter = 0;". The code is as follows:

```

C++

```

void reverse(int array[], int n)
{
 inst_reset();

 for (int i = 0; i < n; i++)
 {
 int tmp = array[n - 1];
 for (int j = n - 1; j > i; j--)
 {
 array[j] = array[j - 1];
 inst_increment();
 }
 array[i] = tmp;
 }
 inst_display(n);
}

```


```

Figure 03.20:
Adding instrumentation macros to code

With these three macros defined, it is now very easy to add and remove instrumentation code.

Generating Instrumentation Reports

By instrumenting our code, we can obtain a measure of how many times a given statement is executed. One measurement, however, is not enough to determine algorithmic efficiency. Instead, we need to know the shape of the curve when we plot input size against iterations. This is accomplished with a two-column table. The first column is the number of input elements, otherwise known as n . The second column is the count of iterations, known as c . To generate this table, we need to modify our instrumentation macros:

The diagram shows a block of C++ code with handwritten annotations. The code defines macros for instrumentation. It includes a conditional block based on the `INSTRUMENT` macro. The annotations are:

- The first macro is called once*: Points to the `#ifdef INSTRUMENT` line.
- The second is called with every experiment*: Points to the `#else // !INSTRUMENT` line.
- As before, when INSTRUMENT is not defined, then everything gets compiled to nothing*: Points to the `#endif` line.

```
C++
#ifndef INSTRUMENT
int inst_counter = 0;
#define inst_increment()
#define inst_reset()
#define inst_display_header()
#define inst_display_row(n)

#else // !INSTRUMENT
#define inst_increment()
#define inst_reset()
#define inst_display_header()
#define inst_display_row(n)
#endif
```

Figure 03.21:
Macros to generate instrumentation reports

When the time comes to instrument our code, we put `inst_display_row()` in the location that we would like to verify. The others get placed around the function so they get called several times:

```
C++
inst_display_header();
inst_reset();
inst_display_row(8);
reverse(array, 8);

inst_reset();
inst_display_row(16);
reverse(array, 16);

inst_reset();
inst_display_row(32);
reverse(array, 32);

inst_reset();
inst_display_row(64);
reverse(array, 64);
```

Figure 03.22:
Creating instrumentation reports

Though we can often validate our Big-O calculation with just one or two measurements, it is often better to have several rows of data in our table.

Output	
n	c
8	28
16	120
32	496
64	2016

Figure 03.23:
Instrumentation output

Interpreting Instrumentation Data

Once the code is instrumented and an output table has been generated, the final step of the process is to interpret the results. This can be done one of three ways: comparing two rows, graphing the results, or creating an equation.

Comparing Two Rows

Each level of algorithmic efficiency is characterized by what happens when the input size is doubled.

Efficiency	Description
$O(1)$	Doubling the size of the input has no impact on execution time
$O(\log n)$	Doubling the size of the input slows execution by one time unit
$O(n)$	Doubling the size of the input doubles execution time
$O(n \log n)$	Doubling the size of the input slows execution by double plus one
$O(n^2)$	Doubling the size of the input quadruples execution time
$O(2^n)$	Increasing size by one doubles execution time

From our data table, we will notice that $n=64$ is roughly quadruple the value of $n=32$ so $O(n^2)$ seems like a likely value.

Graphing the Results

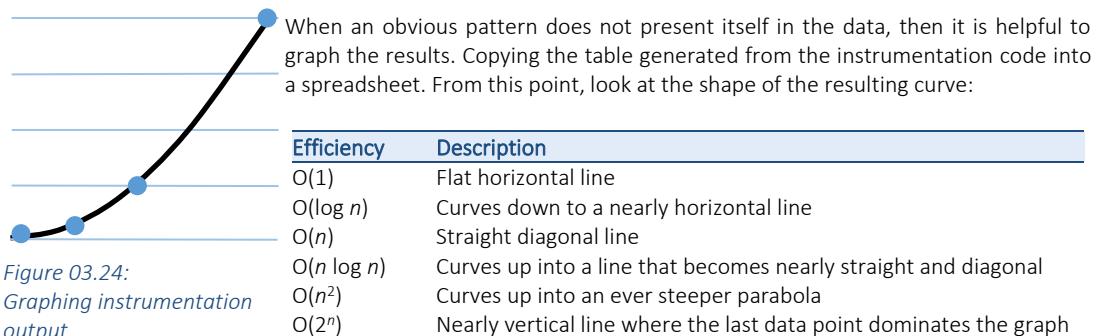


Figure 03.24:
Graphing instrumentation
output

Graphing the four values yields a parabolic curve, supporting an $O(n^2)$ conclusion.

Create an Equation

The final approach is to attempt to create an equation that matches the generated data. Start with an educated guess as to which curve you will be fitting. If you are wrong with this guess, then the curve will not fit and you can try again. For example, say that we are trying to fit an $O(n^2)$ curve. Therefore, our equation will be something like $\text{cost} = c_1n^2 + c_2n + c_3$.

Next, try to identify the highest order constant. In the case of our $O(n^2)$ example, the highest order constant would be c_1 . Taking the data from the last item in the table, we try $2016 = c_164^2$ which yields $c_1 = 0.5$. Next try the lower-order values with the second item on the table: $496 = \frac{1}{2}32^2 + c_232$ which yields $c_2 = -0.5$.

From here, we will then validate our equation by plugging the numbers in for the third item in the table. $\text{cost} = \frac{1}{2}n^2 - \frac{1}{2}n = 136$. This is very close to the recorded value of 120 so we can be confident that this is an $O(n^2)$ algorithm.

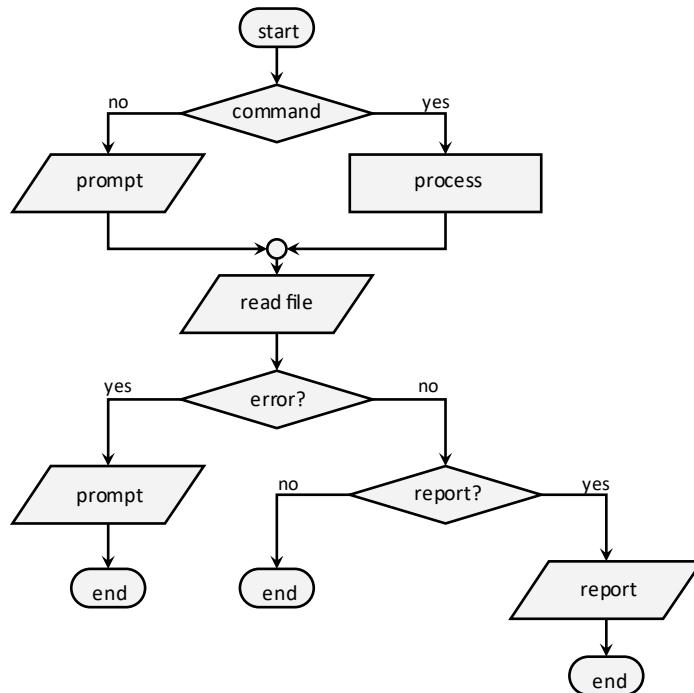
Examples

Example 03.1: O(1) Algorithmic Efficiency

This example will demonstrate how to recognize O(1) or constant algorithmic efficiency.

Problem

Identify the algorithmic efficiency for the following flowchart:



Solution

This flowchart does not have any backtracking, meaning there are no loops. Though execution can follow several different paths, each path has a specific duration. Without loops, this is O(1).

There is a possibility that the algorithm might be O(n) or slower. The read file operation may depend on the size of the input file and therefore might be O(n). There is also no telling what the report operation might be. However, given just the algorithm as presented in this flowchart, we are forced to conclude that it is O(1).

Example 03.2: $O(n)$ Algorithmic Efficiency

This example will demonstrate how to recognize $O(n)$ or linear algorithmic efficiency.

Problem

Identify the algorithmic efficiency for the following pseudocode:

```
Pseudocode
addName(newName, names)
    range[] ← {0, names.size}
    WHILE range[0] ≤ range[1]
        mid ← (range[0] + range[1])/2
        IF names[mid] = newName
            RETURN
        IF names[mid] > newName
            range[1] ← mid - 1
        ELSE
            range[0] ← mid + 1
    FOR i ← names.size ... mid
        names[i] ← name[i - 1]
    names[i] ← newName
```

Solution

The first thing to notice here is that there are two loops. These loops are executed in sequence, meaning that one finishes before the second begins. Since they are not nested, they are not $O(n^2)$ and need to be analyzed independently.

The first loop is a binary search with the range of possible values decreasing by half with every iteration. This fits all the criteria for $O(\log n)$.

The second loop passes through every element in the names array from the back up to the mid index. On average, this mid index will be in the middle of the array (assuming that the place where the `newName` will fit in the array is equally likely to be anywhere in `names`). Thus, we expect to visit $n/2$ elements. This fits all the criteria for $O(n)$.

The overall cost of this algorithm is therefore $O(\log n) + O(n)$ since these two loops are completed in sequence. With algorithmic efficiency calculations, the highest-order exponent dominates the performance equation. Therefore, this is an $O(n)$ algorithm.

Example 03.3: $O(n^2)$

This example will demonstrate how to recognize $O(n^2)$ or N-Squared algorithmic efficiency.

Problem

Identify the algorithmic efficiency for the following code:

```
C++  
void escape(string & text)  
{  
    size_t i;  
  
    while (string::npos != (i = s.find("<")))  
        s.replace(i, 1, string("&lt;"));  
}
```

Note that this code converts all less-than symbols `<` into HTML-encoded `<` symbols.

Solution

With the existence of a single loop, it looks like this might be $O(n)$. However, how many times do we execute this loop? It depends on the number of less-than symbols in the code. In the best-case scenario, there are none so we will not go through the loop at all. In the worst-case scenario, the string contains nothing but less-than symbols. In that case, we are $O(n)$ where n is the length of the string.

In the body of the loop, we call `string::replace`. In this case, the text we are replacing is one character in size (the second parameter to the function) and the new text is four characters in size. This means that everything to the left of the index i needs to be shifted. Shifting for strings is an $O(n)$ operation where n is the size of the string to the right of the index i .

Since we are performing an $O(n)$ operation $O(n)$ times, this is $O(n^2)$.

Example 03.4: $O(n \log n)$ Algorithmic Efficiency

This final example will demonstrate how to recognize $O(n \log n)$ algorithmic efficiency.

Problem

A function has been instrumented and produced the following output table.

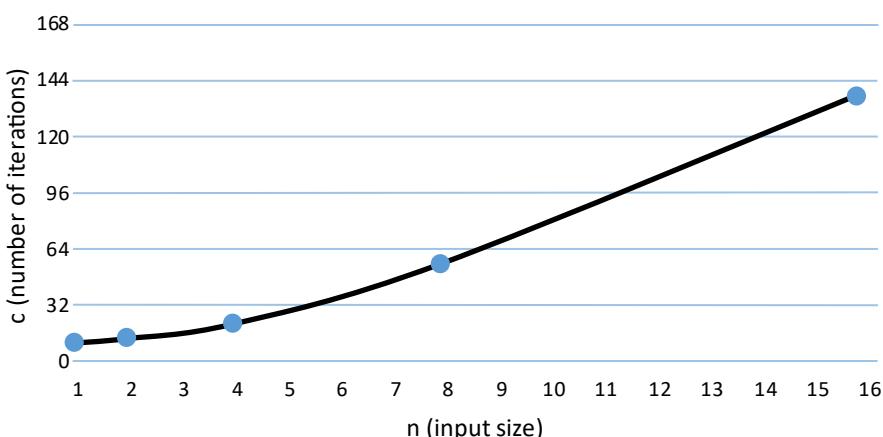
Output	
n	c
1	8
2	12
4	24
8	56
16	136

Based on this data, what is the algorithmic efficiency of the code?

Solution

First, two rows will be compared. From this, we can see that we are roughly doubling the time when the input size is doubled. It could be $O(n)$ or $O(n \log n)$.

Next, we will graph the results. Again, the graph looks linear but, with a small n , there is a definite curve. This leads us to believe it is $O(n \log n)$.



Finally, we will attempt to create an equation to solve the problem. Note that $O(n)$ does fit the curve as n is larger. However, it is off by quite a bit at the bottom. Trying $O(n \log n)$ and the curve almost perfectly fits.

$$\text{cost} = 2(n \log_2 n) + 8$$

Exercises

Exercise 03.1: Performance Level Names

From memory, list the common name for each of the performance levels.

Level	Name
$O(1)$	
$O(\log n)$	
$O(n)$	
$O(n \log n)$	
$O(n^2)$	

Exercise 03.2: Level Characteristics

From memory, characterize what would happen if the input size is doubled.

Level	If input size is doubled...
$O(1)$	
$O(\log n)$	
$O(n)$	
$O(n \log n)$	
$O(n^2)$	

Exercise 03.3: Level Curves

From memory, draw the rough shape of the graph representing the relationship between input size and execution time.

Level	Graph shape
$O(1)$	
$O(\log n)$	
$O(n)$	
$O(n \log n)$	
$O(n^2)$	

Exercise 03.4: Fact or Fiction

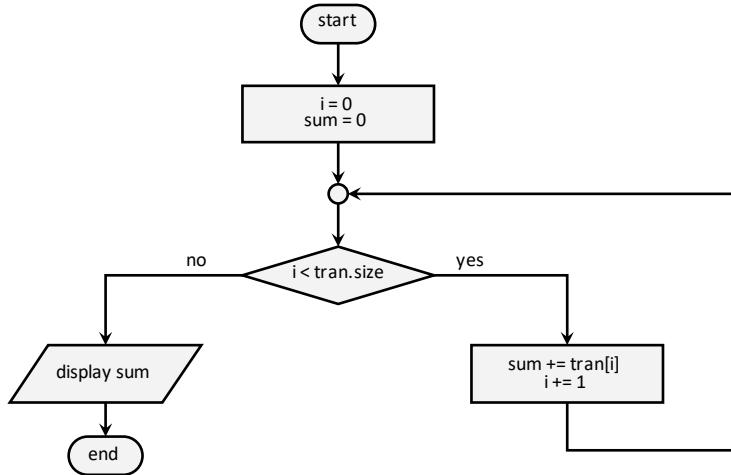
For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Instrumentation code should not interfere with the algorithm it is measuring	
Only one instrumentation data-point is enough to make algorithmic efficiency determinations	
Instrumentation code is often nothing more than a simple counter	
It is often possible to determine algorithmic efficiency by graphing instrumentation data	

Problems

Problem 03.1: Efficiency from Flowchart

Characterize the algorithmic efficiency of the following flowchart:



Problem 03.2: Efficiency from Pseudocode

Characterize the algorithmic efficiency of the following pseudocode:

Pseudocode

```
vector::resize(cNew)
arrayNew ← new(cNew)
FOR i ← 0 ... numElements
    arrayNew[i] ← array[i]
delete(array)
array ← arrayNew
numCapacity ← cNew
```

Problem 03.3: Efficiency from Python Program

Characterize the algorithmic efficiency of the following algorithm:

Python

```
def translate(text):
    keys   = [">", "<", "&", "\n", ","]
    values = ["&gt;", "&lt;", "&amp;", "&NewLine;", "&comma;"]
    translation = zip(keys, values)

    for (key, value) in translation:
        if key == text:
            return value
```

Problem 03.4: Efficiency from Pseudocode

Characterize the algorithmic efficiency of the following algorithm:

Pseudocode

```
duplicatesExist(array, n)
FOR i ← 0 ... n
    FOR j ← 0 ... n
        IF i ≠ j and array[i] = array[j]
            RETURN true
RETURN false
```

Problem 03.5: Efficiency from Pseudocode

Characterize the algorithmic efficiency of the following algorithm:

Pseudocode

```
findLargest(array, n)
largest ← array[0]
FOR i ← 1 ... n
    IF array[i] > largest
        largest ← array[i]
RETURN largest
```

Problem 03.6: Instrument a C++ Program

The following instrumentation macros exist:

```
C++  
#ifdef INSTRUMENT  
int inst_counter = 0;  
#define inst_increment() inst_counter++  
#define inst_reset() inst_counter = 0  
#define inst_display_header() cout << "n\tc\n"  
#define inst_display_row(n) cout << n << "\t" \  
                           << inst_counter << endl  
  
#else // !INSTRUMENT  
#define inst_increment()  
#define inst_reset()  
#define inst_display_header()  
#define inst_display_row(n)  
#endif
```

Instrument the following function:

```
C++  
void wordWrap(char * text)  
{  
    int nColumn = 0;  
    char * pLastSpace = NULL;  
    for (char * p = text; *p; p++)  
    {  
        switch (*p)  
        {  
            case '\n':  
                nColumn = 0;  
                break;  
            case ' ':  
                nColumn++;  
                pLastSpace = p;  
                break;  
            case '\t':  
                nColumn = (nColumn / SIZE_TAB + 1) * SIZE_TAB;  
                break;  
            default:  
                if (nColumn > SIZE_LINE && pLastSpace)  
                {  
                    *pLastSpace = '\n';  
                    nColumn = 0;  
                }  
                else  
                    nColumn++;  
        }  
    }  
}
```

Problem 03.7: Instrumentation Data

Characterize the algorithmic efficiency from the following data set using all three methods described in the chapter:

Output	
n	c
2	2
4	8
8	20
16	44
32	92

Problem 03.8: Instrumentation Data

Characterize the algorithmic efficiency from the following data set using all three methods described in the chapter:

Output	
n	c
1	5
2	11
4	35
8	131
16	515

Challenges

Challenge 03.1: Advanced Search

Consider the advanced search algorithm from Challenge 02.1:

Consider a person looking up the word "glitch" in a dictionary. At first, the word could be in any of the 688 pages in the dictionary. The user cuts the dictionary in half (to page 344) and notices the word "magnet" at the top of the page. This word sorts after "glitch." Because we can assume that the dictionary is sorted, we can deduce that the word "glitch" must be between pages 1 and 343. The user then cuts the remaining pages in half (to page 171) and finds the word "drift" on that page. Since "drift" sorts before "glitch," we can rule out pages 1 through 171. Thus, the possible pages must be 172 through 343. The user then cuts to page 257 and finds the word "guidance." Since "guidance" sorts after "glitch," we can rule out pages 257 through 343. This leaves pages 172 through 256. This process continues until the page containing "glitch" is located.

What is the algorithmic efficiency of this search algorithm?

Challenge 03.2: Bubble Sort

Compute the algorithmic efficiency of the bubble sort:

Pseudocode

```
bubbleSort(array, numItems)
FOR iSpot ← numItems ... 0
    FOR iCheck ← 0 ... iSpot - 1
        IF array[iSpot] < array[iCheck]
            temp ← array[iSpot]
            array[iSpot] ← array[iCheck]
            array[iCheck] ← temp
```

Challenge 03.3: Personal Project

Find the largest project you have yet completed in the programming language of your choice. For this project, identify three functions that do “meaningful work.” Carry out the following tasks for these functions:

1. Compute the algorithmic efficiency of each of the functions.
2. Instrument each function, collecting 5 data points.
3. Determine the algorithmic efficiency using the “compare two rows” method.
4. Determine the algorithmic efficiency using the “graph the results” method.
5. Determine the algorithmic efficiency using the “create an equation” method.

Maintainability

Maintainability is measure of how much time or effort is required to fix defects or make enhancements.

75%–90% of the cost of software occurs after the project is “finished”

“finished.” This is not the case with commercial software. By most estimates, 75%–90% of the cost of software occurs after the project is “finished.” What is the source of this cost?

Most industry experts point to four main factors influencing maintenance costs: corrective costs, adaptive costs, perfective costs, and product enhancements. The exact ratio of these costs varies from project to project of course, but the broad categories of costs have proven to be consistent across a broad spectrum of software systems.

Source	Description
Perfective Maintenance	Improvements and enhancements that do not change initial functionality. For example, performance improvements would be perfective. On average, this constitutes 5% of maintenance costs.
Corrective Maintenance	Fixing defects in the software that were discovered after it was initially deployed. This is usually about 17% of maintenance costs.
Adaptive Maintenance	Adjusting software so it can continue to function as before in a new or changing environment. This is usually 18% of costs.
Enhancements	Adding new functionality that was not part of the initial specification or design. This is usually 60% of costs.

There have been several attempts to objectively and mechanically compute the maintainability of code. Usually such metrics take into account the number of lines of code, the number and types of control flow constructs (called cyclomatic complexity), and the number and types of operations performed (called the Halstead complexity measure). While there may be a correlation between these metrics and how easy it is to perform maintenance, it is easy to find counterexamples: puzzling code often has a low maintainability score (and should therefore be more maintainable) whereas obvious code (which should have a low maintainability score) sometimes has a high score. In other words, these objective and mechanically computed maintainability metrics often produce misleading values.

The ultimate metric of maintainability is how long it takes to make an adjustment or fix a defect. Specifically, how long does it take to make the specific change that we are faced with right now! Since this cost is completely dependent on the nature of the change and who is asked to make it, we cannot use this metric as a generic tool for measuring maintainability. Instead, we need to uncover the factors that make changes difficult to make. Fortunately, this can be reduced to two simple concepts: understandability and malleability.

Characterizing Understandability

Studies have shown that the process of understanding unfamiliar software accounts for approximately 30% of maintenance costs. This is a major contributor to the perfective and corrective maintenance activities of a codebase. Clearly, we would like to minimize these costs. How, then, can we make our code more comprehensible?

Understandability is a measure of how easy it is for a programmer to form a valid mental model of a segment of code

this definition. The first is “how easy it is ... to form.” Understandability is about the process of gaining insight and internalizing facts. In other words, it is a learning process.

The second component of the definition is the phrase “a programmer.” This could be the person who wrote the code last week or someone who has never seen the code before. It could be a novice programmer or a seasoned professional. Clearly, the selection of who represents “a programmer” greatly influences the understandability metric. It is therefore critical to accurately identify who will be performing this software maintenance. If the individual who wrote the code will be asked to maintain it, that is one thing. If, on the other hand, maintenance will be turned over to another team, this must factor into the equation.

The final component of the definition is “a valid mental model.” The programmer in question needs to understand the code sufficiently to make a productive change. This does not necessarily mean that every single detail of the software needs to be understood. It just means that it needs to be understood well enough.

When making an understandability determination on a block of code, there are five levels or categories. These are:

Level	Definition
Obvious	There is no room for confusion or interpretation.
Straightforward	Everything is clearly stated; there is no subtlety.
Deducible	All required information is present.
Misleading	An invalid mental model is suggested.
Puzzling	Substantial effort is required to figure things out.

When characterizing the understandability of software, one must first know something about who will be maintaining the code

Our goal, of course, is to make all code obvious. In order to do this, we need to identify those aspects of our code which could make it merely deducible or worse!

Obvious Understandability

Software is classified as obvious if there is no room for confusion or interpretation. This definition is filled with subjectivity, of course. It states, in effect, that no programmer could possibly misunderstand the intent nor the details of a given algorithm. This is unachievable, of course! However, the impossibility of this task is the point: there is always room for improvement. Code should be written towards the unattainable goal of having zero room for confusion. What does obvious code look like? It has several properties: has unambiguous names, honors well-understood coding conventions, and has clear and helpful comments.

Names

Code is obvious if all the variable, function, class, and file names completely and concisely describe what they represent. This is often called *intent-revealing names*. While it is impossible to mathematically describe what qualifies as an obvious name,

it can be quickly determined with an easy test. If one were to ask another programmer unfamiliar with your particular codebase what the variable `balanceChecking` means, their response should exactly fit the usage of that variable in the code.

Provided nothing but a variable or function name, one should be able to completely describe its usage in the code

Designing code with obvious names is a two-step process. First, take the time to come up with names for coding entity that completely and concisely represent what they are meant to do. If this step is difficult, then we have the first indication that the code will have maintainability issues. Second, verify that every usage of that entity is consistent with the chosen name.

Coding Conventions

Every programming language has a set of best practices and an associated style. Similarly, every organization has a Standard Operating Procedure (SOP) for performing common tasks. Obvious code honors these coding conventions whenever possible.

Perhaps this is best explained by example. In Python, there is a standard way to visit every element in a list. In C, there is a standard loop used to traverse a C string. These coding patterns become familiar and routine to anyone who has written code in a given language for any length of time. If a large project has 100 instances of reading data from a file, then these 100 instances should open and close the file in exactly the same way. Consistency and repetition in code makes for more obvious and maintainable code.

Consistency and repetition in code makes for obvious and maintainable code

Comments

Programming languages are very good at explaining what needs to happen in a given algorithm, but they never can explain “why.” Comments, fundamentally, are there to answer “why” questions. If there is any information that another programmer would need to understand a given block of code, the comment should clearly, unambiguously, and concisely provide that information.

Comments answer “why” questions

One final thought. Uninformative comments are worse than unhelpful, they are distracting. Every piece of extraneous information steals attention from the critical information. Comments should never be added for their own sake; they should all fulfill a definite purpose.

Straightforward Understandability

Code is considered straightforward when everything is clearly stated and there is no subtlety. In many contexts, such as art, literature, and society, it is not a compliment to be called straightforward or accused of lacking subtlety. With software engineering, the opposite is the case.

Everything is clearly stated; there is no subtlety

We all love a nuanced joke. We welcome a story with a subtle plot development or a song with an understated melody. We appreciate a picture or a painting with delicate and almost imperceptible nudges directing the eye to a given feature of the image. These are all interesting because they cause us to think, to seek out details, and to take notice. Code is fundamentally different. The purpose here is not to be entertained, but rather to be understood. The antithesis of straightforwardness is cleverness. If any code construct is chosen for the purpose of being clever, unique, or just different, then it cannot be considered straightforward.

Code is straightforward rather than just deducible when every coding construct can be taken at face value. The following must be honored for code to be considered straightforward:

Property	Description
Variables	Every variable must represent exactly one concept.
Functions	Every function must perform exactly one task.
Side Effects	There can be no side effects.
Standard Usage	Every programming construct must be used in the standard or mainstream way.

There are very good reasons why one would sacrifice understandability in code. The most common reason is performance. In order to minimize the size of a data structure, in order to reduce the cost of traversing a list, or in order to streamline the initialization process, it is often necessary to sacrifice one of the straightforward principles. This may be a worthwhile trade-off. If you go down this path, you must realize that there will be a comprehensibility cost which will increase the overall maintenance costs of the software.

Variables:
Every variable
represents exactly
one concept

Figure 04.1:
Straightforward
understandability

```
JavaScript
*****
* UPDATE REGISTERS
* Update the CPU's registers
*****
function updateRegisters()
{
    // update the register
    var elementReg = document.getElementById("reg");
    elementReg.innerHTML = reg;

    // update the IP or Next Instruction
    var elementIP = document.getElementById("ip");
    elementIP.innerHTML = ip;
}
```

Functions:
The function
performs exactly
one task

Standard Usage:
Content is added
to the page in the
standard way

Deducible Understandability

Software is classified as deducible if it is possible with careful analysis for a programmer to figure out how it works. There are two key words in this definition: possible and carefully. The word "possible" does not mean "probably" or even "likely." It means that it is not beyond the scope of possibility. With so much room for confusion and opportunities for mistake, deducible software is often fragile. The word "carefully" implies that the programmer needs to slow down and focus. It implies attention to detail and caution. In other words, it is hard. When you are working on something that is merely deducible, be that your income tax or an overly complicated website, do you not get frustrated? Why, then, do we inflict this on ourselves and other programmers when we write code that is merely deducible?

On the surface, this may not seem to be a very good characterization of software. If it is barely deducible, should the programmer not strive to make it straightforward, or even obvious? Sadly, this is not always possible. Sometimes software is designed to model highly complex things, such as intricate business processes or sophisticated mathematical models. There is only so much a programmer can do in situations such as these.

The problem comes when the system the software is meant to model is not overly complicated but the code itself is. When the author of the code says things like "well, any reasonable programmer could figure this out" or "there is nothing incorrect or misrepresentative about this code," then they are probably settling for deducible when they could be better. In situations like these, it is instructive to ask yourself this question: "yes, the programmer could figure this out, but why are we making him/her work so hard?"

Yes, the programmer could figure this out, but why are we making him/her work so hard?

The following must be honored for code to be considered deducible:

Property	Description
Present	All information is present that is needed to deduce how the code behaves.
Documented	All tools and constructs are used in accordance of their documented behavior.
Truthful	Every name embodies to some degree what it represents.

Consider the following code. All the necessary information is present, the constructs are used in accordance with their documented behavior, and the names are representative. However, few would consider this code straightforward or obvious.

```
C
void strcpy(char *dest, const char *src)
{
    while(*dest++ = *src++) ;
}
```

Four things are happening here at once!!!

Figure 04.2:
Deducible
understandability

Misleading Understandability

Software classified as misleading when any aspect of the design suggests that it behaves differently than it does. In other words, there is at least one lie. The most common reasons why code would be considered misleading include outdated comments, variables that change their meaning, functions that perform different tasks than their names suggest, and language constructs used in nonstandard ways.

Comments

It is easy for comments to get out of date in code. Some seasoned (and perhaps jaded) programmers even state, “never trust comments, only trust code!” Why is this so common? The answer lies in how we write code.

VB

```
' Display values from 1..5
For counter As Integer = 2 To 10 Step 2
    Debug.WriteLine(counter.ToString & " ")
```

Figure 04.3:
Misleading comments

Programmers typically add comments as code is being authored. They think of commenting as part of the design process not part of the maintenance process. When it comes time to add functionality, fix a bug, or just tidy up some code, they do not think to

update the comments. As a result, the comments become out of sync with how the code works and become misleading.

The solution is very simple: when you read any block of code, check your understanding of the code with the explanation given in the comments. If they are out of sync (and you are sure you are right!), update the comments.

Variables

A horrible programming practice that is surprisingly common is to reuse a previously declared variable for a different purpose. Some justify because they think they are saving memory. It turns out, this is not true! Compilers are very good at reusing space. When a single variable is used for two different purposes, there is an implied relationship between the two uses. If this implication is not true, then the code is misleading.

C#

```
// yearly income
float income = 81743.10;

// monthly income
income /= 12.0;

// income tax
income = computeTax(income);
```

Figure 04.4:
Misleading variable

Functions

Functions should perform the action their names suggest. This is sometimes not the case when the programmer failed to take the time to properly name a function. Another common cause is when the function has an unintended side effect, changing system state in an unexpected way. This is such a widespread problem that an entire discipline of programming (called functional programming) was created to address this problem. Any time a function does more or less than the function name suggests, that function is misleading.

Figure 04.5:
Misleading use of language construct

Language Constructs

C++

```
// Do we mean to compare values or
// assign a new value?
if (value = getValue())
    cout << "Zero is not allowed!";
```

IF statements are designed for making decisions. Loops are designed for looping. Arrays are designed to hold data. You would be surprised how often these facts are not true in a program. Whenever a programming construct is used differently than intended, then the code is in danger of being misleading.

Puzzling Understandability

An algorithm is considered puzzling when substantial effort is required to comprehend even small blocks of code. The programmer is never sure if he or she has figured it out regardless of how much time or effort is spent studying it. You may wonder how this could be the case. Code, after all, is completely deterministic.

An algorithm is puzzling when it is hard to comprehend even small blocks of code

I remember as a young programmer wondering why English could not be as easy as programming. One could spend an entire lifetime working on a single English essay and not be sure it is right, but could spend 10 minutes on a program and be confident. This optimism quickly faded in the face of a large production codebase. I remember spending a couple hours studying a single function, still not sure I understood all the ramifications of a seemingly simple code change.

To emphasize this point: every year since 1984, there has been the “International Obfuscated C Code Contest.” One entry from David Burton is the following, which definitely qualifies as puzzling:

```
C  
0;main(1){for(;~1;(0=~(1=getchar())?-~0&printf("%02x ",1)*5:  
!0)||puts(""));}
```

Figure 04.6:
Obfuscated code,
presented by David Burton

Why would someone write code that is deliberately puzzling? All too often, the answer is: “I want to show other programmers how clever I am.” This attitude in any manifestation is likely to result in incomprehensible and therefore difficult to maintain code.

Aside from malice and pride, there are legitimate reasons why production code may be classified as puzzling. Here are a few:

- **Lack of time:** Blaise Pascal once said, “I would have written a shorter letter, but I did not have the time.” In other words, it takes effort to produce a condensed effort but is easy to write a verbose one. The same is true with writing code. We can hack together a working solution filled with unnecessary bits and misleading names. It takes effort, however, to reduce this to an elegant and understandable algorithm. In the pressures of the workplace, we often do not have the luxury of doing it “right.” This is called “technical debt,” a mass of unmaintainable code.
- **Lack of care:** Why make code more understandable when no one will ever look at it again? There is some truth to this. It is more important that the compiler understands the code than a human. As you may expect, there is a flaw to this logic. If the code is valuable to someone, then it will need to be maintained. Puzzling code is far more expensive to maintain than obvious code. This means the small cost today of making code more straightforward is avoided at the expense of maintenance later. Why would anyone choose that alternative? Instead, incur the small cost of making code maintainable to reduce the overall cost of the codebase.
- **Risk minimization:** At certain phases of the development cycle (such as when a product is about to be released to the customer), the development team becomes risk averse. They are unwilling to make a change to the codebase that is difficult to validate. This yields a collection of small changes which, while easy to verify, can change the intent of the code. After years and decades of this, a once obvious codebase can become merely deducible, then misleading, and eventually puzzling.

Characterizing Malleability

Malleability is a measure of how easy it is for a programmer to make changes to a system

Malleability is a measurement of how easy it is for a programmer to make changes to a system. There are three parts to this definition. The first is “a programmer.” It is important to recognize who exactly will be making the changes. If, for example, you

find it easy to think in the context of a given programming paradigm (such as functional programming, recursion, or polymorphism), then you may consider code written in that paradigm to be malleable. The programmer who is asked to make changes in this code years down the line might not agree! It is worthwhile to write code within the norms of the organization to which you belong so you can get everyone’s perspective as to what is difficult.

The second part of the definition is “make changes.” This can mean things as simple as a one-line bug fix or as complex as adding an entire subsystem. To get a grip on what “changes” means for your system, you need to think about how it is going to be used. It is always useful to have a 1-year, 5-year, and 10-year roadmap, even if it is just a vague vision of where the product is headed. Usually this roadmap is not something the programmer creates; instead, it is owned by the client, the marketing folks, or perhaps an executive. If you do not have a roadmap for your project, it is a good idea to ask about it.

We favor design alternatives that makes it easier to make predictable changes

The final part of the definition is “how easy.” This is a function of development effort, measurable in both time spent and expertise required. We thus favor design alternatives that lower the difficulty of making predictable changes. If we can engineer it so that non-programmers can make the change, so much the better!

All projects should have a roadmap, even if it is just a vague vision of where the project is headed

When making a malleability determination on a block of code, there are four levels or categories. These are:

Level	Definition
Configurable	Big changes can be made without altering any code, mostly through modification of configuration files.
Data-driven	Many behavior changes can be made without changing program logic.
Adjustable	Most updates do not alter the structure of the program and without altering much code.
Refactorable	Refactoring and redesigns are required for most nontrivial updates and changes.
Prohibitive	Changes are so difficult that it is easier to start over than to change the code.

Our goal, of course, is to make all code as malleable as possible. Unfortunately, this goal is often at odds with other design goals such as performance, initial development cost, and even comprehensibility. It is the charge of all engineers to make design trade-offs like these.

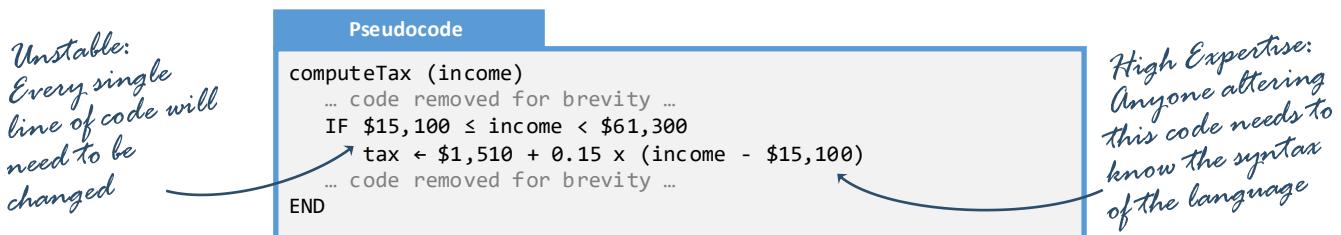
Configurable Malleability

Code is configurable when large alterations can be made without changing any code

Code can be considered configurable when large and substantial alterations can be made without changing any code. This is the gold standard for maintainability. Just about any project can be made configurable with enough time and effort.

Perhaps configurable malleability is best explained by example. The early 1978 game of *Space Invaders* shares no code with the 1979 *Galaxian*, even though the gameplay is very similar. One of the reasons is that the gameplay and the graphics of the game were “baked into” the codebase. Contrast this with the Unreal Engine by Epic Games in 1998. This single game engine hosted 16 distinct games by late 1999, the only difference between the games was the different configuration files to describe characters and levels. This made unreal engine more configurable than Space Invaders.

Consider a function designed to compute an individual’s tax burden given their income level. Nonmalleable software would bake the logic into the function:



Knowing that tax codes change every single year, more pliable software would make these alterations impact less code and require less expertise. Ideally it would require no change to code and be able to be accomplished by a nontechnical individual.

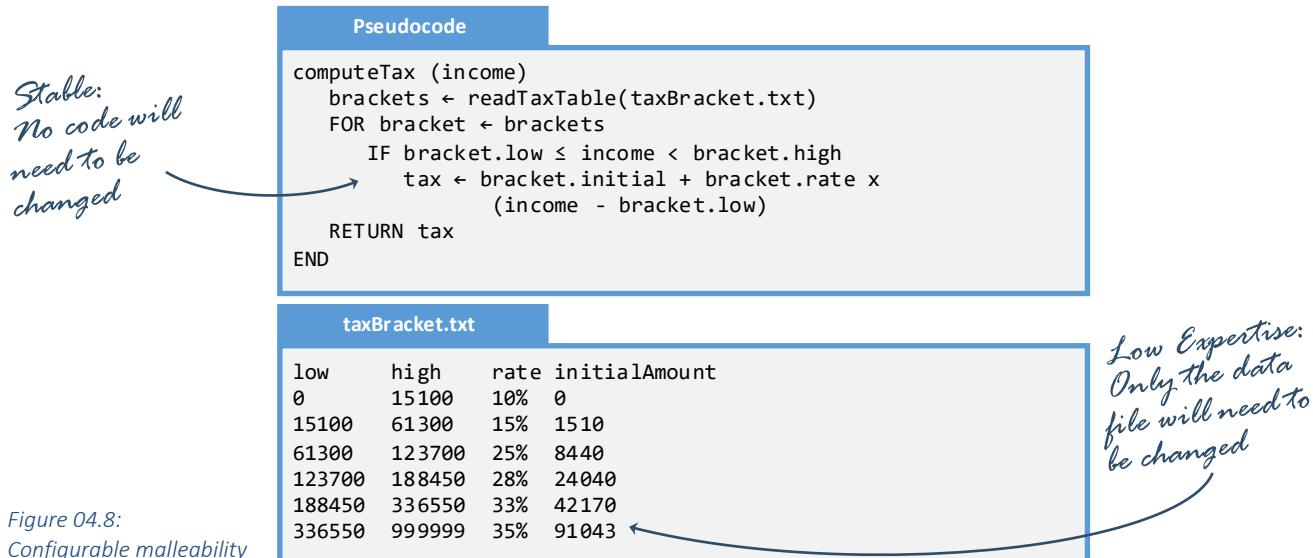


Figure 04.8:
Configurable malleability

Though it requires considerably more development effort and is often more difficult to test, configurable code has a significantly smaller maintenance cost.

Data-Driven Malleability

Code is considered data-driven when alterations can be made without changing the logic of the program. Most algorithms consist of two distinct components: the data and the logic. The data include all the constants in the algorithm: the numbers and the text. The program then uses these data to make decisions. The logic of the algorithm consists of the statements such as loops and decision statements. An algorithm is data-driven when pivotal parts of the program are separated from the program logic. The trick, of course, is to identify the pivotal parts...

Best Practice 04.1 Move algorithm data into constants

One of the easiest ways to make an algorithm data-driven is to move the data component of an algorithm into constants. Consider a cashier program designed to serve a small business. If the sales tax was 5.9%, then the programmer might be tempted to sprinkle the 0.059 constant in the code in a dozen or more places. The problem surfaces when the law changes and the sales tax jumps to 6.1%. A data-driven design would be to put the sales tax rate in a single global constant, making it easy to make the inevitable change. In the following example, the overtime rate (time-and-a-half overtime means you get paid 150% of your normal wage) and overtime threshold (40 hour workweek) are separated from the logic of the algorithm so changes can be made more easily.

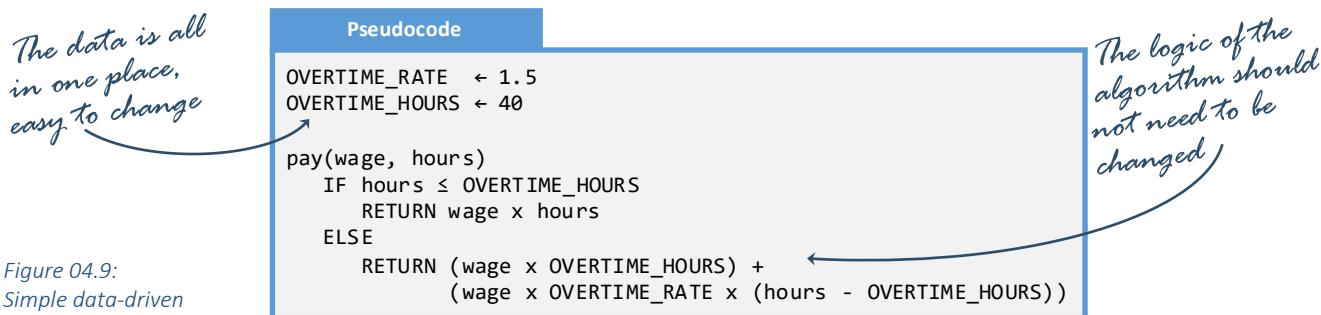


Figure 04.9:
Simple data-driven
malleability

The same data-driven technique can be applied to more complicated algorithms. The following code for our `computeTax()` program function is data-driven:

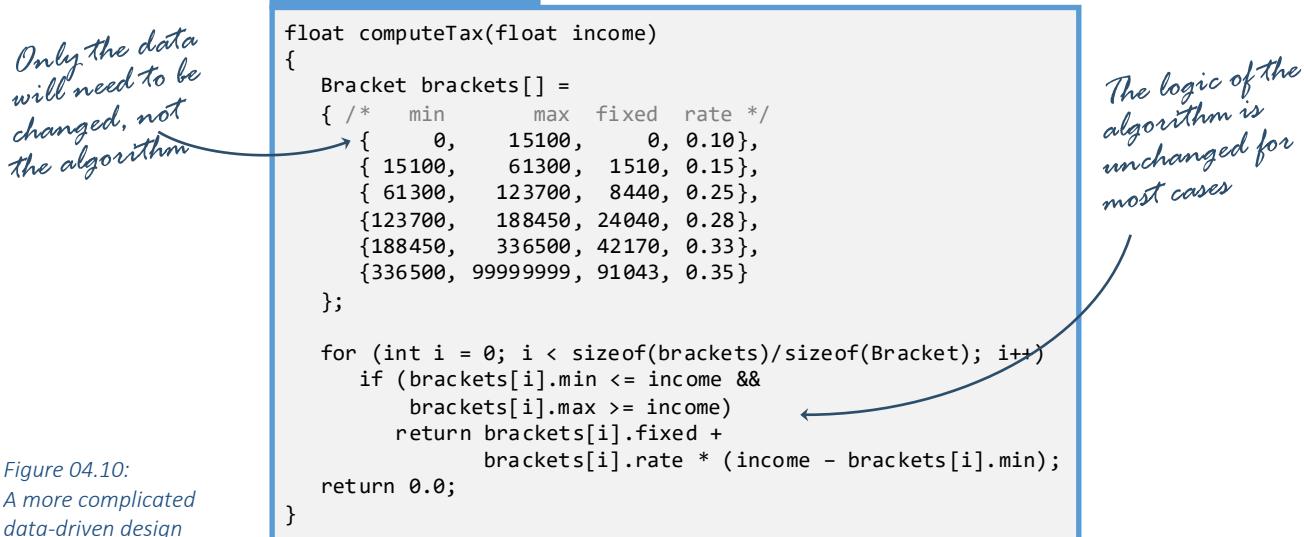


Figure 04.10:
A more complicated
data-driven design

Adjustable Malleability

Code is considered to exhibit adjustable malleability when most alterations can be made without altering the structure or overall design of the code. Conditionals may need to be added, Boolean expressions may need to be changed, statements may need to be removed, and whole blocks of code may need to be written. All these things fall within the realm of “adjustable.”

Adjustable code can be altered without changing the overall design or structure

It is easy to get data-driven, adjustable, and refactorable (discussed on the following page) malleability confused. In many ways, the difference is only a matter of degree. Data-driven code is code where most changes only involve changing the data rather than the logic of the program. If you find yourself adding, removing, or adjusting statements, then the code is not data-driven. Refactorable malleability requires changes to be made to the code before the new functionality is even started. Adjustable malleability, on the other hand, requires logical changes to the algorithm but these changes do not require refactoring.

To make code more adjustable, the programmer needs to anticipate changing requirements

To make code adjustable, the programmer needs to anticipating changing requirements, isolating code which depends on these requirements from that which is stable. There are a few things that can be done to help with this process.

Best Practice 04.2 Avoid duplicate code

If there is more than one copy of code in the codebase, then this is a good indication that there is a maintenance problem. Often this happens when an impatient programmer copy-pastes code. Why is this so bad?

Imagine the scenario where code that was copied contains a bug or the code needs improvement. The programmer making this change needs to be aware that other copies exist before making those changes. Of course, this is a problematic process, likely to result in something getting forgotten or done inconsistently. Rather than copy-pasting code, one should core out the duplicate code into a separate function.

Best Practice 04.3 Avoid overly complex Boolean expressions

Boolean expressions can be notoriously difficult to create and understand. This is especially true when there are more than three or four Boolean operators involved. In almost every case, they can be simplified to a more straightforward format. Better yet, break the large Boolean expressions into a collection of more manageable ones, each represented with a well labeled variable.

Best Practice 04.4 Keep it linear

Whenever possible, the path through the code should be simple to follow. Functions should have one entrance and one exit. The same should hold true with loops. Though many languages have `break` and `continue` statements allowing the programmer to use nonstandard program flow in loops, they should be avoided. Any overly-nested loops or conditionals, any `goto` statements, and any `breaks` or `continues` are almost always indications of convoluted code.

Refactorable Malleability

Refactoring is the process of restructuring or redesigning existing and functioning code. Usually the goal of refactoring is to improve the design while maintaining the existing functionality. In other words, refactored code will work the same as before, it will just be better. Examples of refactor tasks include moving code into or out of functions, eliminating duplicate code, enabling code to work with more than one data type, turning **IF/ELSE-IF** statements into **SWITCH/CASE** statements, and simplifying Boolean expressions.

Refactoring is restructuring existing code, so it behaves the same but works better

We refactor code when need to improve non-functional attributes: performance, security, stability, and of course maintainability. Novice programmers consider a task complete when the code works as expected. Experienced programmers know that an important step remains: refactoring.

Code exhibits refactorable malleability when anticipatable changes require refactoring. In other words, the code requires significant reworking and reengineering before the programmer can even start to make the required changes. If no refactoring is required, then the code is at least adjustable malleability. If the cost of refactoring is greater than the cost of rewriting the code from scratch, then the code can be labeled prohibitive.

If code must be refactored before any significant change is applied, then it exhibits refactorable malleability

Best Practice 04.5 Thoroughly test the code before refactoring it

When a change needs to made in code that exhibits refactorable malleability, it is necessary to first obtain a detailed understanding of how the code works. As a bare minimum, exercise the code with a wide variety of input. Hopefully the code works as expected without any bugs. When a bug is encountered, don't just fix it! It might be the case that another part of the code depends on this anomalous behavior. Take a few minutes to see if this is the case before fixing the bug. In truly horrible code, the refactoring process can take many hours as bug after bug is systematically tracked down. I remember one such refactor task that took two days. I kept a notebook documenting all the leads I had to track down. What should have been an easy task turned into a long and stressful process.

Best Practice 04.6 Only refactor code when there is no other option

Software engineers tend to be perfectionists. We like building a perfect world where everything is as it should be. Software engineers also tend to be a bit egotistical; we think our approach is better than all others. These two forces tend to steer programmers into refactoring, even when the situation does not require it. Be aware of this temptation and resist it!

Refactoring existing code is an expensive and risky proposition. When we do so, the testing investment is essentially thrown out, requiring everything to be retested. Of course, this investment is small on new code, especially code that has yet to be integrated into the larger project. However, legacy code that has endured several rounds of testing should be refactored only when necessary.

Best Practice 04.7 Design code to simplify refactoring or to eliminate the need to refactor

Code exhibiting refactorable malleability is poorly maintainable. It should be avoided. One can avoid this level of malleability by preferring straightforward solutions over clever ones, by avoiding or eliminating duplicate code, and by trying not to interweave code that does different types of things.

Prohibitive Malleability

Code is considered prohibitive when even small and trivial changes require substantial reengineering. In almost all cases, it is far easier and less risky to just start over. Perhaps this is best explained by example.

Microsoft Word was written in assembly by Charles Simonyi and Richard Brodie in the early 1980s. Shortly thereafter, the code was translated into the C programming language. A key part of the program was the `formatLine()` function which figured out how to put a single line of text on the screen. This was a single monolithic function of more than twenty thousand lines of code. Because `formatLine()` was converted from assembly, it consisted of hundreds of `goto` statements jumping forwards and backwards through the function. As you can imagine, it was extremely difficult to understand and very fragile; even the most innocuous changes often resulted in unexpected behavior. As the years went by, this single function became more and more of a liability. Features were put off because the cost of making changes to `formatLine()` was simply too expensive and fixing bugs was often very time consuming. Finally, in 1999 everyone on the team acknowledged that something had to be done. A group of five engineers was tasked with refactoring and rewriting `formatLine()` to make it more malleable. Three years later, they created a new component called line services which had the same functionality of `formatLine()` but was far more configurable and straightforward. It took two additional years to integrate this into Word, where it made it into the 2003 release. One single function with prohibitive malleability cost the company tens of millions of dollars.

It is easy to recognize when an algorithm has prohibitive malleability: it is a mess! It is the kind of code that makes your stomach turn and makes you want to shake your head. The question is: what do you do when you encounter it?

Best Practice 04.8 Rewrite only as a last resort

Rewriting is the process of starting from a clean slate. It involves identifying the original requirements, starting the design process anew, and writing the code from scratch. When rewriting, one may occasionally look at the original code as a point of reference. However, that code is seldom used. Refactoring, on the other hand, involves moving existing code, renaming existing components, and making a myriad of adjustments. Though the design may change substantially, the important thing is that elements of the original design and/or code remains.

It is seldom a good idea to just rewrite code you don't like. Often, wholesale rewrites create more problems than they solve. An incremental approach is far better than throwing everything out and starting over. Prefer refactoring over rewriting.

Best Practice 04.9 Manage technical debt carefully

We all periodically take shortcuts to finish a job rather than do it right. Sometimes the shortcut is good enough for the task at hand. More often than not, that shortcut comes back to haunt us and we need to redo our sloppy work. With programming, the accumulation of these shortcuts is called technical debt.

Technical debt is the accumulation of maintainability problems resulting from engineers pursuing easy solutions rather than doing things "right." Sometimes this can be justified, especially when a deadline is looming. If we do this too often, the codebase can become a maintenance nightmare. The project manager should carefully record those areas that are accumulating technical debt and schedule time to fix problem areas before they become a competitive disadvantage to the team.

Examples

Example 04.1: Obvious Understandability

This example will demonstrate how to recognize obvious understandability.

Problem

Characterize the level of understandability for the following code:

Pseudocode

```
handleDialogEvent(dialog)
SWITCH dialog.eventType
CASE INIT
    initializeDialog(dialog.dx, dialog.dy)
    setDialogValue(idScore, dialog.score)
    setDialogValue(idPoints, dialog.points)
    updatePreview(dialog.data)
CASE UPDATE
    updatePreview(dialog.data)
CASE OK
    saveValues(dialog)
CASE INPUT
    if not validScore(getDialogValue(idScore))
        ERROR invalid score
```

Solution

This function is at least deducible. All the information necessary to determine how the code behaves is present, from what we can tell the tools and constructs used in the expected way, and every function and variable name describes how it is used.

The function is at least straightforward. Every name represents one concept or performs one task, there do not appear to be any side effects, and everything is used in a standard or mainstream way.

The code appears to be obvious but based on this one sample it is difficult to be certain. Each function name appears to completely describe how it is used. This is certainly true of `handleDialogEvent()` itself. All the pseudocode conventions are honored. Finally, being pseudocode, there are no comments so one cannot make a determination in this regard.

Example 04.2: Deducible Understandability

This example will demonstrate how to recognize deducible understandability.

Problem

Characterize the level of understandability for the following code:

```
Perl
#!/usr/bin/perl

open (FILE, $ARGV[0]);
@c = <FILE>;
chomp @c;
close FILE;
$i = 0;
foreach $l (@c)
{
    $l =~ s/\s+$///;
    $l =~ s/\t/        /g;
    ++$i;
    $len = length $l;
    if ($len > 79)
    {
        print "$i --> $len\n\n";
    }
}
```

Solution

The joke goes that Perl is a write-only language; it cannot be read! Perhaps this is true. Even though this short program has only a half-dozen variables, none of the names are enough to get a clue as to what they do. With some work, we can figure out that “*c*” stores the contents of a file, “*l*” is one line of the file, “*i*” is the line number of the file, and “*len*” is the length of a line. With this solved, we can then see that this program is trying to determine how many lines of code are longer than 79 characters. However, the regular expressions are a bit tricky. Tabs are replaced with eight white spaces and spaces at the ends of lines appear to be stripped. This can be figured out but requires some detective work.

Overall, the code is not misleading and is certainly not intentionally puzzling. It is, however, a long way from being straightforward. It can be classified as deducible.

One could improve this to being straightforward with a few minor adjustments: more descriptive variable names and comments on a few key statements.

Example 04.3: Data-Driven Malleability

This example will demonstrate how to recognize data-driven malleability.

Problem

Characterize the level of malleability for the following code. This code is designed to HTML-encode characters in an input string.

Python

```
def translate(text):
    keys   = [>, <, &, "\n", , ]
    values = [&gt;, &lt;, &amp;, &NewLine;, &comma;]
    translation = zip(keys, values)

    for (key, value) in translation:
        if key == text:
            return value
```

Solution

This function is not configurable because you need to modify source code to make a change. If one were to change an encoding or add another to the list, then clearly some knowledge of Python will be required.

This function is data-driven because the data of the algorithm (the tags to be replaced and the inserted text) are separate from the logic of the algorithm. A programmer will not need to know how the FOR loop works in order to change this code. She will only need to realize that the **keys** and **values** lists are related by index. Thus, minimal expertise is required. The code is also stable because only the two arrays will need to be changed if the mapping is altered. No program logic (the loop and the IF statement in this case) will need to be changed.

To make this truly configurable, then the program would have to read the keys and the values from an external source such as a file or database. In most applications, the performance cost of reading the data from a file probably does not justify the malleability advantages.

Exercises

Exercise 04.1: Understandability Definitions

From memory, define each of the understandability levels:

Level	Name
Obvious	
Straightforward	
Deducible	
Misleading	
Puzzling	

Exercise 04.2: Malleability Definitions

From memory, define each of the malleability levels:

Level	Name
Configurable	
Data-Driven	
Adjustable	
Refactorable	
Prohibitive	

Exercise 04.3: Classify Maintainability

Classify the level of understandability and malleability of the following code:

JavaScript

```
function computeTax (income) {
    var tax;

    // 10% tax bracket
    if (income >= 0 && income < 15100) {
        tax = income * 0.10;
    }
    // 15% tax bracket
    else if (income >= 15100 && income < 61300) {
        tax = 1510 + 0.15 * (income - 15100);
    }
    // 25% tax bracket
    else if (income >= 61300 && income < 123700) {
        tax = 8440 + 0.25 * (income - 61300);
    }
    // 28% tax bracket
    else if (income >= 123700 && income < 188450) {
        tax = 24040 + 0.28 * (income - 123700);
    }
    // 33% tax bracket
    else if (income >= 188450 && income < 336550) {
        tax = 42170 + 0.33 * (income - 188450);
    }
    // 35% tax bracket
    else if (income >= 336550) {
        tax = 91043 + 0.35 * (income - 336550);
    }
    return tax;
}
```

Problems

Problem 04.1: Authentication Maintainability

Characterize the degree of understandability and malleability for the following code and suggest how it can be improved.

```
C
int main() {
    const char * password = "rosebud";
    char userInput[256];
    int numAttempts = 0;

    do {
        printf("password: ");
        getline(userInput, 256);

        if (strcmp(password, userInput) == 0) {
            printf("You are authenticated\n");
            return 0;
        }

        numAttempts++;
    } while (numAttempts < 5);
    printf("You are out of tries\n");
    return 1;
}
```

Problem 04.2: Phone Number Maintainability

Characterize the degree of understandability and malleability for the following code and suggest how it can be improved.

```
C#
public bool IsValidPhone(string input)
{
    if (string.IsNullOrEmpty(Phone))
        return false;
    string pattern =
        @"\(?\d{3}\)\)?[-\.\.]*\d{3}[-\.\.]*[-\.\.]*\d{4}";
    Regex express = new Regex(pattern)
    return express.IsMatch(input);
}
```

Problem 04.3: Roman Maintainability

Characterize the degree of understandability and malleability for the following code and suggest how it can be improved.

C++

```
string convert (int input)
{
    string output;
    int div;
    string r[4][10] = {{ "", "I", "II", "III", "IV",
                         "V", "VI", "VII", "VIII", "IX" },
                        { "", "X", "XX", "XXX", "XL",
                          "L", "LX", "LXX", "LXXX", "XC" },
                        { "", "C", "CC", "CCC", "CD",
                          "D", "DC", "DCC", "DCCC", "CM" },
                        { "", "M", "MM", "MMM", "",
                          "", "", "" }};

    for (int i = 3; i >= 0; --i)
    {
        div = static_cast<int> (pow (10.0, i));
        output += r[i][input / div];
        input %= div;
    }
    return output;
}
```

Problem 04.4: Scores Maintainability

Characterize the degree of understandability and malleability for the following code and suggest how it can be improved.

Python

```
# Update the score
def updateScore(game, scoreMapping):

    # Obtain a list of all the obstacles the user has cleared
    obstaclesCleared = game.getObstaclesCleared()

    # For each obstacle, look up the point value
    for obstacle in obstaclesCleared:
        score += scoreMapping[obstacle]

    # Update the score value and draw it
    game.updateScore(score)
    game.drawScore()
```

Problem 04.5: Chess Maintainability

Characterize the degree of understandability and malleability for the following code and suggest how it can be improved.

C++

```
#define WHITE_WHITE "\E[31;47m" // red / white
#define WHITE_BLACK "\E[30;47m" // white / red
#define BLACK_WHITE "\E[37;41m" // black / white
#define BLACK_BLACK "\E[30;41m" // black / red
#define RESET_COLOR "\E[0m" // reset color
#define RESET_SCREEN "\E[H\E[2J"

Piece::display()
{
    bool whiteSquare = (getPosition() % 2 ? true : false);
    bool whitePiece = isWhite() && (getLetter() != ' ');

    // display the coloring
    if (whitePiece && whiteSquare) // red && white
        cout << WHITE_WHITE;
    if (whitePiece && !whiteSquare) // white && red
        cout << BLACK_WHITE;
    if (!whitePiece && whiteSquare) // black && white
        cout << WHITE_BLACK;
    if (!whitePiece && !whiteSquare) // black && red
        cout << BLACK_BLACK;

    // display the piece
    cout << " " << getLetter() << " ";

    // reset the coloring
    cout << "\E[0m";
}
```

Problem 04.6: Names Maintainability

Characterize the degree of understandability and malleability for the following code and suggest how it can be improved.

Perl

```
#!/usr/bin/perl

while (<>) {
    s/(.*) (.*)\(.*)/$2, $1/;
    print;
}
```

Problem 04.7: MadLib Maintainability

Characterize the degree of understandability and malleability for the following code and suggest how it can be improved.

C++

```
char b[1024] = "#[]{}\\n\\'\\\" ";
while (std::cout << OPENING; ) {
    bool fN = (std::cin.getline(b + 11, 255)) && false;
    std::ifstream fin(b + 11);

    for (char *p; fin >> b + 11; ) {
        bool fP = fN && OP(b[12]) &&
            !(ispunct(b[11]) && b[11] != '<');
        fN = b[12] != '#' && b[12] != '[' && b[12] != '{';
        if (b[11] == '<' && (1 + (i = 0))) {
            while (b[i] == b[12] ? b[12] = 0*(b[11] = b[i + 5]):
                i++ - 5);
            if (isalpha(b[12]) && (p = b + 13)) {
                while (*p = *p == '_' ? ' ' : *p == '>' ? 0 : *p)
                    p++;
                std::cout << "\t" << (char)toupper(b[12])
                    << b + 13 << ":" ;
                std::cin.getline(b + 11, 256);
            }
        }

        strcat(b + 128, (fP) ? b + 10 : b + 11);
        b[128] = '\0';
    }

    std::cout << "\n" << b + 128 << QUESTION;
    if (std::cin >> b + 32 &&
        std::cin.ignore() &&
        b[32] != 'y')
        return (int)(std::cout << CLOSING && false);
}
```

Challenges

Challenge 04.1: MadLib Refactoring

MadLib is a game where the program asks the user a series of questions and the responses are put into a story. The resulting story is then presented to the user. The program reads the story from a file, looking for <noun> and turns it into a question "\tNoun: ". It also turns <[> into open-quote and <#> into a newline.

Refactor the following working code so it is more maintainable.

C++

```
#define OP(b)      ((b) != ']' && (b) != '#' && (b) != '}')
#define CLOSING    "Thank you for playing.\n"
#define OPENING     "Please enter the filename of the Mad Lib: "
#define QUESTION   "\nDo you want to play again (y/n)? "

int main() {
    int i;
    char b[1024] = "#[]{}\\n'\"\\\" ";
    while (std::cout << OPENING; ) {
        bool fN = (std::cin.getline(b + 11, 255)) && false;
        std::ifstream fin(b + 11);

        for (char *p; fin >> b + 11; ) {
            bool fP = fN && OP(b[12]) &&
                !(ispunct(b[11]) && b[11] != '<');
            fN = b[12] != '#' && b[12] != '[' && b[12] != '{';
            if (b[11] == '<' && (1 + (i = 0))) {
                while (b[i] == b[12] ? b[12] = 0*(b[11] = b[i + 5]):
                    i++ - 5);
                if (isalpha(b[12]) && (p = b + 13)) {
                    while (*p = *p == '_' ? ' ' : *p == '>' ? 0 : *p)
                        p++;
                    std::cout << "\t" << (char)toupper(b[12])
                        << b + 13 << ": ";
                    std::cin.getline(b + 11, 256);
                }
            }
            strcat(b + 128, (fP) ? b + 10 : b + 11);
            b[128] = '\0';
        }

        std::cout << "\n" << b + 128 << QUESTION;
        if (std::cin >> b + 32 &&
            std::cin.ignore() &&
            b[32] != 'y')
            return (int)(std::cout << CLOSING && false);
    }
}
```

Challenge 04.2: Roman Numerals Refactoring

Consider the following code from Problem 04.3:

C++

```
string convert (int input)
{
    string output;
    int div;
    string r[4][10] = {{ "", "I", "II", "III", "IV",
                         "V", "VI", "VII", "VIII", "IX" },
                        { "", "X", "XX", "XXX", "XL",
                          "L", "LX", "LXX", "LXXX", "XC" },
                        { "", "C", "CC", "CCC", "CD",
                          "D", "DC", "DCC", "DCCC", "CM" },
                        { "", "M", "MM", "MMM", "",
                          "", "", "" }};

    for (int i = 3; i >= 0; --i)
    {
        div = static_cast<int> (pow (10.0, i));
        output += r[i][input / div];
        input %= div;
    }
    return output;
}
```

Please do the following:

- Refactor the code so it is more maintainable.
- Fix the bugs – there are several!
- Add functionality so it can return a valid Roman numeral for numbers greater than a million.

Challenge 04.3: Open Source

Find an open source project in a language of your choice. Sift through the code, looking for a function or a method which is misleading or convoluted. Rewrite it so it is obvious and pliable.

Chapter 05 Assert

An assert is a check placed in a program representing an assumption the developer thinks is always true. If that assumption ever proves to be false, then a notification is sent to the developer.

When writing code, we often make many assumptions. We assume that an algorithm can perform its task correctly; we assume the input is in a certain format; we assume our own data structures are correctly configured. A diligent programmer would check all these assumptions to make sure her code is robust. Unfortunately, most of these checks are redundant and, to make matters worse, can be a drain on performance. A method is needed to allow a programmer to state all her assumptions, get notified when an assumption is violated, and have these checks not influence the speed or stability of the customer's program. Asserts are designed to fill this need.

An assert is a check placed in a program representing an assumption the developer thinks is always true

An assert is a check placed in a program representing an assumption the developer thinks is always true. In other words, the developer does not believe the assumption will ever be proven false and, if it does, she wants to be notified. An assert is expressed as a Boolean expression where the true evaluation indicates the assumption proved correct and the false evaluation indicates violation of the assumption. Asserts are evaluated at runtime, verifying the integrity of assumptions with each execution of the code.

In many ways, an assert is like a warning light in a car. Your car has dozens of sensors monitoring a wide variety of things. My car monitors the oil pressure, engine temperature, air pressure in each tire, fuel in the gas tank, status of the air filter, and many other things. When the car is operating normally, none of these sensors should fire. When they do, then the "check engine" light illuminates indicating that the car should be taken to a mechanic. The mechanic will then get the error code from the car to determine exactly which sensor triggered the alarm. Asserts work much like this. They do not fire during normal execution of the program. When they do fire, then something serious has just happened. Just like the error code in your car, the assert will direct the programmer to the exact line of code that malfunctioned.

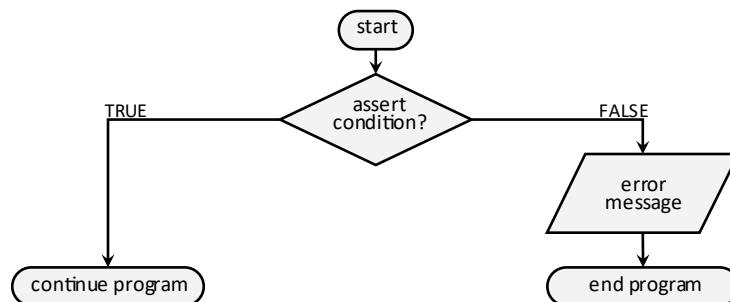


Figure 05.1:
Flowchart describing
an assert

Using Asserts

Asserts are debug mechanisms found in most programming languages. If they are not built into your language, then they probably exist as a library or can be easily created by hand. There are usually three steps in using an assert: including the assert library, adding the assertion code, and interpreting the resulting output.

Including the Assert Library

Most modern programming languages have assertions built into the language, alleviating the need for this step. This includes Python, Java, Swift, C#, VB, and Kotlin. However, most languages written before the turn of the century include assertions in a separate library. C++ and Node.js fall into this category. The following is how to include the assertion library in C++ and Node.js:

C++	<pre>#include <cassert> // using the modern C++ implementation #include <assert.h> // using the older C libraries</pre>
Node.js	<pre>var assert = require('assert');</pre>

Figure 05.2:
Including an assert library

If a language does not have assertions built-in and if no assertion library exists, it is possible to create your own assert. It is easy to create a simple assert, but it may not fulfill all your needs. For example, assertions should not be present in customer-facing code (called “shipping code” or “production code”). In this case the assertions should be compiled away much like comments are. To see how to do this, please see the next section of this chapter titled “Building an Assert.”

The pseudocode for creating a simplistic assert is the following:

Pseudocode	<pre>assert(condition) IF !condition PUT error message</pre>
------------	--

Figure 05.3:
Pseudocode for
a simple assert

JavaScript does not provide an assertion mechanism nor an assertion library (unless you use Node.js). A simplistic assertion function in JavaScript is:

JavaScript	<pre>function assert(condition) { if (!condition) { throw "Assertion failed!"; } }</pre>
------------	--

An error occurred,
but we don't
know where!

Figure 05.4:
A simplistic assert
function in JavaScript

Adding Assertions to Code

An assertion looks like a function call in most languages. It takes a Boolean expression as a parameter and returns nothing. Many assertion implementations allow the programmer to add a second optional parameter sending a custom message to the user. Consider, for example, the assumption that your bank account balance is positive. The following assertions would check that condition.

Most languages have more than one variety of assert

Java

```
assert balance >= 0.00;
assert balance >= 0.00 : "Invalid account balance";
```

Swift

```
assert(balance >= 0.00)
assert(balance >= 0.00, "Invalid account balance")
```

Python

```
assert balance >= 0.00
assert balance >= 0.00, "Invalid account balance"
```

C#

```
Debug.Assert (balance >= 0.00);
```

Node.js

```
assert(balance >= 0.00);
assert(balance >= 0.00, "Invalid account balance");
```

Kotlin

```
assert(balance >= 0.00)
assert(balance >= 0.00, "Invalid account balance")
```

C++

```
assert(balance >= 0.00);
```

VB

```
Debug.Assert(balance >= 0.00)
```

The syntax of asserts is nearly identical in most languages

Figure 05.5:
Assert syntax in various languages

Note that the assert parameter can be any Boolean expression. Since assertions are not meant to impact the normal operation of the program, be cautious of changing state. The following are syntactically correct but should be avoided:

variables should not change in asserts

Pseudocode

```
assert balance < 0
assert initialize() = true
```

functions changing state should not be called in asserts

Figure 05.6:
Misuses of asserts.
Never do this!

Assert Variations

The most used assertion type is a simple function that takes a Boolean and returns nothing. Many assertion libraries that offer some variations which can make the code easier to understand and simpler. A few examples are the following

Name	Example	Description
Default	<code>assert(gpa <= 4.0)</code>	This is the simplest and most common assertion. Most assertions are this type. If the assertion condition is clear and obvious, no further documentation is needed.
Message	<code>assert(gpa <= 4.0, "GPA is too high")</code>	Allows the author to explain something about the assumption being tested. This message is displayed if the assertion failed.
Equality Assertion	<code>assertEquals(numElements, 10)</code>	Tests whether two variables or objects are equals. This can also be done with a default assertion. <code>assert(numElements == 10).</code>
Not Reached	<code>fail()</code>	Should be placed where normal execution should not reach. It is the same as the following: <code>assert(false).</code>
Expected Exception	<code>assertThrows(function())</code>	Verifies that an exception is thrown when a given action is executed. This assertion will catch the exception if thrown and assert if it is not thrown.
Same	<code>assertSame(object1, object2)</code>	Asserts that object1 and object2 refer to the same object. This is useful in languages like Python where variables are references to objects.
Is Null	<code>assertIsNull(object)</code>	Tests whether a given object is a NULL pointer or a null object depending on the object type and the language. <code>assert(object != NULL).</code>
Is Instance	<code>assertIsInstance(object, type)</code>	Tests whether a given object is an instance of a given class. This is useful with polymorphic objects or loosely typed languages.

Most of these assertion variations offer no more descriptive power than the default assert function. They do clarify the intent of the assertion, making the code more understandable and therefore easier to maintain.

Though most modern languages have assertions, it is uncommon for them to have more than the default and message variation. However, most languages implement the xUnit test framework which includes a rich section of assertion variations. To learn more about unit tests, please see Chapter 25 Quality: Unit Test.

Interpreting Assertion Output

If the Boolean expression in an assertion evaluates to true, then nothing happens; the assertion function returns and the program continues as before. If the Boolean expression evaluates to false, then one says that the assertion “fired.” This is akin to a warning light illuminating in your car. In most cases, an error message is displayed and the program terminates.

Some assertion implementations provide the programmer with little insight as to what happened or where. This is true with our simplistic JavaScript assert, with Python, and with Node.js:

If an assertion evaluates to false, we say that it “fired.” This means an error has been detected in the code

```
Python Output
AssertionError

Node.js Output
AssertionError: 0 == true
```

Figure 05.7:
Assert error messages,
not very helpful!

Many modern languages throw exceptions when an assert fires. This means that we get a standard error message which may or may not have any details as to where the error occurred. If your development environment has a debugger, it will take you to the source of the uncaught exception so you can fix the bug from there:

```
Java Output
Exception in thread "main" java.lang.AssertionError
```

Figure 05.8:
Many languages have
asserts throw exceptions

Some languages provide a rich set of data when the assert fires. This directs the programmer to the exact line of code (or more!) where the assert is located. Probably the best example is the C++ `cassert` library. Here, a fired assert will tell you the name of the executable, the source file in which the assertion exists, the line number of the assertion, and the function name. It also tells you the Boolean expression that evaluated to false if you find that useful. In most cases, you only really need the file name and the line number.

```
C++ Output
executable: file.cpp:1265 int function():
Assertion `balance >= 0.0' failed
```

The Boolean
expression is
provided

The file and
line number are
in the message

Figure 05.9:
A full-featured assert
error message. The line
number and file name are
most important

Building an Assert

In the case where your programming language does not provide an assertion statement or when the provided statement does not work as you need, it is often necessary to build your own assert. This can be a useful exercise to get a better idea how asserts work. We will use C++ as our language for this demonstration.

Line Number, File Name, and Function Name

A useful assert outputs its file name and the line number. Fortunately in C++ there is an easy way to get this data. Consider the following code in `figure05_10.cpp`:

```
C++
bool function(int parameter)
{
    cout << "file: " << __FILE__ << endl;
    cout << "line: " << __LINE__ << endl;
    cout << "function: " << __FUNCTION__ << endl;

    return true;
}

Output
file: figure05_10.cpp
line: 7
function: function
```

These macros are resolved at compile time in C++

Figure 05.10:
Displaying the file name,
line number, and
function name

The `__FILE__` macro retrieves the name of the source file at compile time. This is especially helpful in large projects where there may be hundreds of files. The `__LINE__` macro is the line number at compile time. Notice that statement that displays the line number happens to be at line 7 (from the comment at the right). Finally, the function name is contained in the `__FUNCTION__` macro. With these three macros, we can make our first draft of `assert()`:

```
C++
void assert(bool condition)
{
    if (!condition)
    {
        cerr << __FILE__ << ':' << __LINE__
        << ',' << __FUNCTION__ << "(): Assert!\n";
        exit(1);
    }
}
void function()
{
    assert(false);
}

Output
figure05_11.cpp:11 assert(): Assert!
```

We are getting line 11 from assert()

This will reveal the line number of the assertion function, not of where the assert is placed in the code

We should be getting line 18 from function()

Figure 05.11:
A first draft at an assert

There is a problem with this approach: it will unfortunately display the line number of the `assert()` function, not the line number from which it was called.

Correct Line Number

To get the assertion function to report the correct function name and line number, we need to pass that parameter directly to the function.

The screenshot shows a C++ code editor with the following code:

```

C++
void function(char * file, int line, char * func)
{
    cout << "file: " << file << endl;
    cout << "line: " << line << endl;
    cout << "function: " << func << endl;
}
int main()
{
    function(__FILE__, __LINE__, __FUNCTION__);
    return 0;
}

```

The code is numbered from 01 to 14 on the right. A handwritten note on the right side says: "We are not calling the macros from this function now." An arrow points from line 12 to the `__FUNCTION__` macro in the `function` call.

Output

```

file: figure05_12.cpp
line: 12
function: main

```

A handwritten note on the left side says: "We are getting line 12 and function main". An arrow points from this note to the output window.

Figure 05.12
Getting the line number correct

We will now incorporate this code into our draft of the assert function. We will call this function `assertCode()` for now because it takes more parameters than we will eventually want. In the end, the assert function will take only a single Boolean parameter.

The screenshot shows a C++ code editor with the following code:

```

C++
void assertCode(bool condition,
               const char * file,
               int line,
               const char * func,
               const char * conditionString)
{
    if (!condition)
    {
        cerr << file << ':' << line
        << ' ' << func << "():" 
        << "Assertion `" << conditionString << "'"
        << " failed\n";
        exit(1);
    }
}
int main()
{
    assertCode(false == true, __FILE__, __LINE__,
              __FUNCTION__, "false == true");
    return 0;
}

```

The code is numbered from 06 to 26 on the right. A handwritten note on the right side says: "This is too complicated! We should only need to pass one parameter". An arrow points from the `__FUNCTION__` parameter to the `__FUNCTION__` parameter in the `assertCode` call.

Output

```

figure05_13.cpp:23 main(): Assertion `false == true' failed

```

A handwritten note on the left side says: "We need to pass too many parameters!". An arrow points from this note to the `assertCode` call. Another handwritten note on the right side says: "The output is correct!". An arrow points from this note to the output window.

Figure 05.13
An assert with the correct function name and line number

Notice that our new draft gets the function name and the line number correct. However, it needs to pass too many parameters. We need another draft!

Simplify the Parameters

In order to reduce the number of parameters to our custom assertion, we will use a macro. Many languages provide “pre-processor directives,” a collection of macros that can be executed before the program is compiled. C++ is no exception. In the case of the `#define` directive, the pre-compiler will replace all instances of the first parameter with the second.

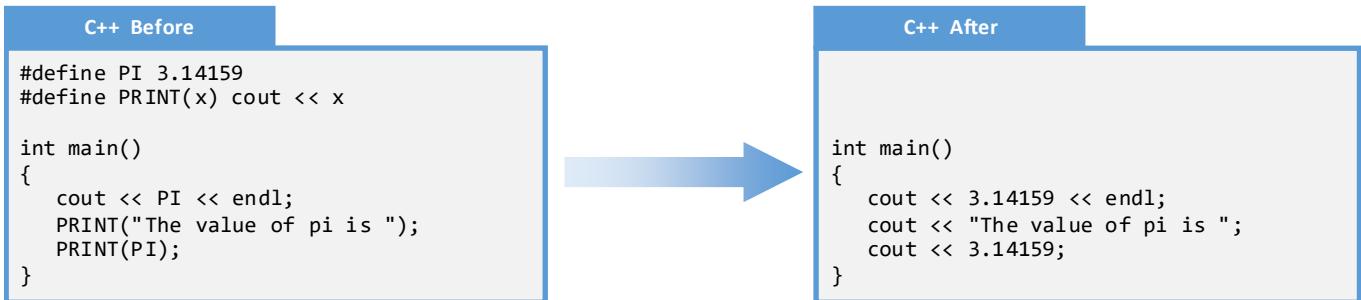


Figure 05.14
Macro expansion

Now, with these `#define` macros, we can simplify the parameters to our assertion function. Note that the user will call `assert()` with one parameter rather than `assertCode()` with five.

Only one parameter is needed here

The user of the assert macro does not need to know about all these parameters

The output is correct!

```
C++  
#define assert(condition) assertCode(condition, __FILE__,  
    __LINE__, __FUNCTION__,  
    #condition)  
  
void assertCode(bool condition,  
    const char * file,  
    int line,  
    const char * func,  
    const char * conditionString)  
{  
    if (!condition)  
    {  
        cerr << file << ':' << line  
        << ' ' << func << "() : "  
        << "Assertion `" << conditionString << "'"  
        << " failed\n";  
        exit(1);  
    }  
}  
int main()  
{  
    → assert(false == true);  
    return 0;  
}
```

Output
figure05_15.cpp:26 main(): Assertion `false == true' failed

Figure 05.15
A second draft
of an assert

Debug Only

There is one final step necessary to make our assert work as expected. We want the assert to disappear if `NDEBUG` is defined. In other words, we would like the production code that the user sees to have no assertions but the debug code that the development sees to have all these checks. Fortunately, we can do this with some more `#define` trickery.

The C++ pre-compiler allows for simple conditions to be resolved at compile time. There are many reasons why this might be useful. We may want the code to compile both on Apple operating systems as well as Microsoft. Since they use different interfaces, we can seamlessly switch between them with pre-compiler directives. To see how this works, consider the following code.

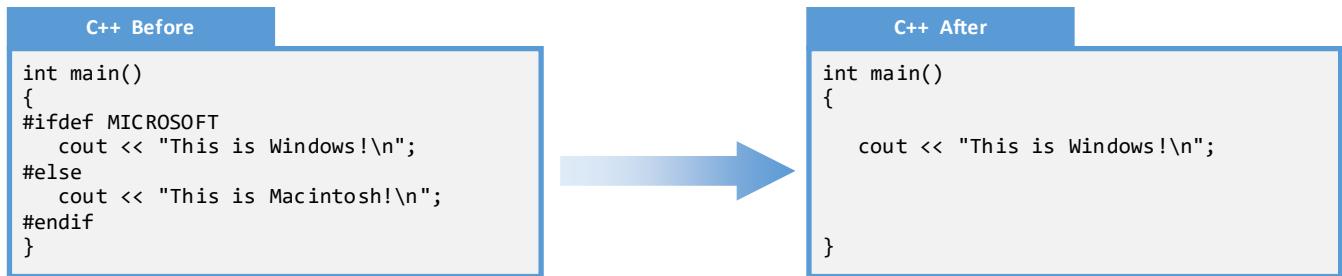


Figure 05.16
Conditional macros

The `#ifdef` construct conditionally compiles depending on whether certain macros are defined. In this case, defining the `MICROSOFT` macro will allow the first `cout` to be compiled, otherwise the second. We can apply this same technique for our `assert()` macro. When the `NDEBUG` macro is defined, then all the assertions in the application get compiled to nothing. This makes it easy to switch between debug and ship modes without changing code.

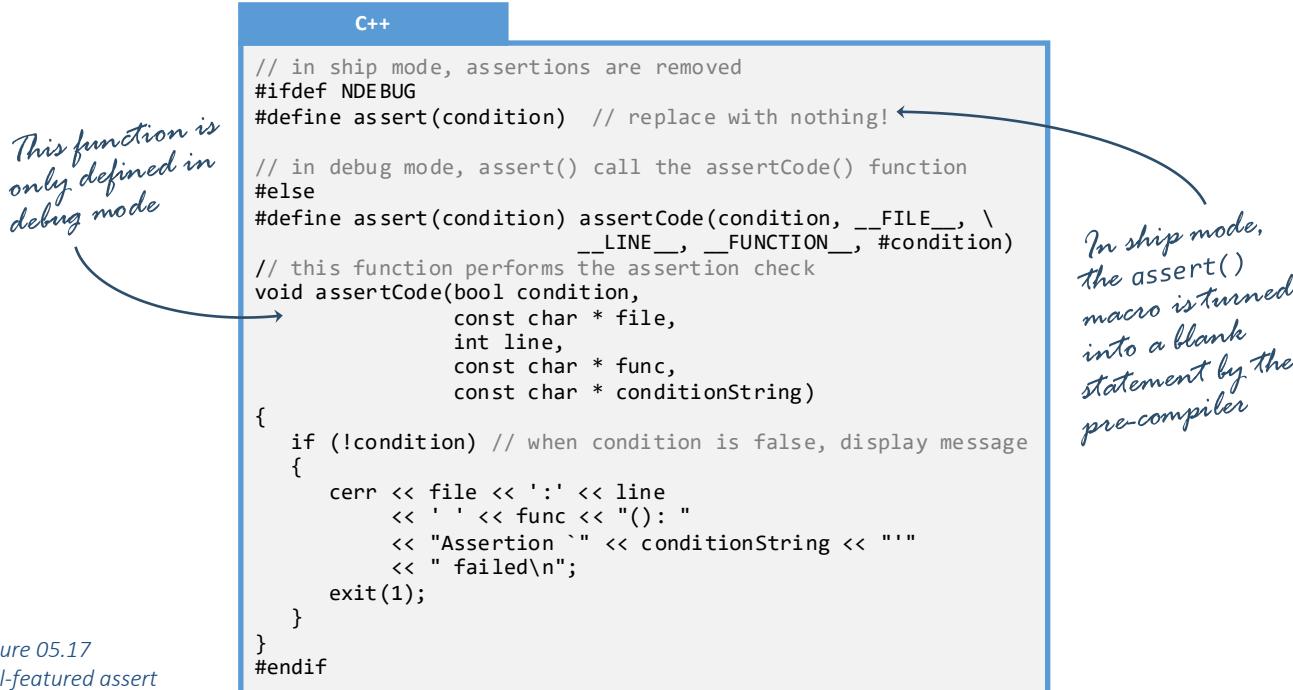


Figure 05.17
Full-featured assert

Preventing Bugs With Asserts

Assertions are among the quickest, easiest, and most effective way to detect bugs. If they are used correctly, they are also the easiest way to fix a bug as well. Developers rejoice when they are asked to fix a bug that begins with an assert. They cower when the problem is some subtle behavior anomaly. If you habitually insert asserts into your code, you will make simple and complex projects so much easier.

Use Asserts for System Runtime Errors

There are two fundamental types of errors: user and system errors. The term “user errors” applies to a broad collection of errors stemming from external input into the system. This could be from files, network entities, other programs, and even users. Generally, a well-designed application should be tolerant of any combination of external input, be that malicious, ignorant, or accidental. When a user error is encountered, the program should gracefully recover and continue to function normally. Normally user errors are detected with IF statements. Asserts are not designed to handle user errors; typically, they provide cryptic error messages and abruptly terminate. This is not what one would call “user friendly!” System errors, on the other hand, are defects within the code. They are the result of mistakes from the programmer. It is the goal of software engineers to prevent or remove all system errors from the codebase. For some reason, it is impossible to achieve a completely defect-free program, no matter how small or simple.

Best Practice 05.1 Use asserts for system errors, use IF statements for user errors

Figure 05.18
Do not use asserts for checks that should be done at compile time

```
Python
age = 42
assert type(age) == type(0)
```

This should be a
compiletime
check, not a
run-time check

There is another classification of errors: compile time and runtime. Compile time errors, as the name suggests, occurs during the process of converting source code into an executable. These errors are the result of the source code not conforming to the syntax of the language. While we all hate compile errors, they are much easier to fix than runtime errors. Runtime errors occur during the execution of the program. They usually take the form of undesirable behavior or sudden termination (crash). Note that some programming languages such as VB and JavaScript are interpreted. This means that compilation occurs at runtime. While these languages blur the line between compile time and runtime, they are still two distinct operations.

Best Practice 05.2 Use asserts for runtime errors, let the compiler to catch compile time errors

Handle this error
with an IF
statement

```
Python
age = int(input("What is your age? "))
assert age > 0
```

Figure 05.19
Do not use asserts
for user errors

Document Assumptions with Asserts

As mentioned previously, programmers make a plethora of assumptions while writing code. Rather than jot them down on a scrap of paper or try to keep them in your head, document assumptions with asserts.

Best Practice 05.3 Draft asserts during the design phase

When making the first draft of an algorithm, the designer may realize that assumptions are being made. Often those assumptions are noted on the flowchart or in the pseudocode. It is not too early to start thinking of how to turn those assumptions into asserts that can be checked. Make a draft of these asserts and be sure to put them in the production code when it is written.

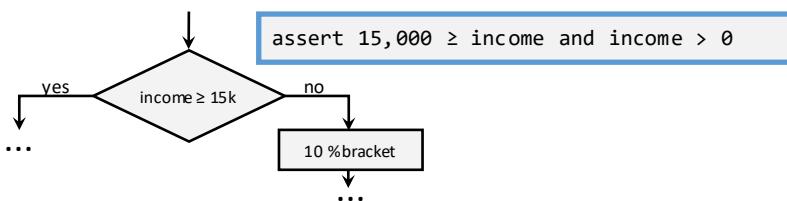


Figure 05.20
Start thinking about
asserts early in
the design phase

Best Practice 05.4 Turn comments into asserts

Figure 05.21
Many comments should
actually be asserts

When writing code, we often feel the urge to explain nonobvious constructs with comments. These constructs may contain assumptions. These should be converted into asserts so you can be notified when they are violated.



Best Practice 05.5 Add a comment next to asserts describing how to fix a bug

If you are writing an assert that you do not expect to ever fire, ask yourself, “what would happen if it did?” At this point in time, you are probably better qualified than any to fix this hypothetical bug. Add a few comments describing how the bug could be fixed if this assert ever did fire. Think how relieved you will be far in the future when you discover this little comment after having been notified that the assert did in fact fire!

Figure 05.22
Add comments to help
you fix bugs if asserts fire

```
Python
sum = 0
for num in numbers:
    sum += num
    # If this assert fires, then put the entire loop
    # in an IF statement and set the average to zero
    assert len(numbers) > 0
    average = sum / len(numbers)
```

The diagram shows a blue box containing Python code with a specific comment highlighted by a curved arrow: "# If this assert fires, then put the entire loop # in an IF statement and set the average to zero". To the right of the box, handwritten text reads: "Comments such as these can be a big time saver if the assert is ever to fire".

Best Practice 05.6 Document that which should be unreachable with an assert

When writing the code, you may notice that something is unreachable or impossible. This might even be true ... right now! After a few rounds of edits and bug fixes, however, it may no longer be true. Document these unreachable points with an assert.

The diagram shows a Java code snippet in a blue-bordered box. The code is a switch statement with several cases for Account types (CHECKING, SAVINGS, STOCK) and a default case. The default case contains a multi-line comment explaining that if a new account type is added, code needs to be added to handle it. It also suggests adding a case in a file-save function. An annotation 'Document unreachable code with an assert.' with a curved arrow points to the 'assert false : account;' line in the default case.

```
Java
{
    switch(account) {
        case Account.CHECKING:
            ...
            break;
        case Account.SAVINGS:
            ...
            break;
        case Account.STOCK:
            ...
            break;
        default:
            // If a new account type is added, you will need
            // to add code here to handle that case. Also add
            // a case in the file-save function as well
            assert false : account;
    }
}
```

Figure 05.23
Put asserts to document unreachable code

Best Practice 05.7 Add an assert even if there is “no way possible” for it to fire

If a statement is obviously true, don't hesitate to put an assert there. There might be something you have not considered or perhaps the code will change after the last time you looked at it. Just because you can't think of any possible way that the condition is not met does not mean that a way does not exist.

Figure 05.24
Use asserts for even obvious checks

The diagram shows pseudocode in a blue-bordered box. It prompts for a number, gets the number, and then performs two operations: an assert statement and an assignment to a buckets array. A handwritten note 'Everyone knows that you get 0 or 1 if you mod by 2!' with a curved arrow points to the assert line.

```
Pseudocode
PROMPT for a number
GET number

assert number % 2 = 1 or number % 2 = 0 ←
buckets[number % 2] ← number
```

In the above example, Python will always give you 0 or 1 when you mod by 2. C++, on the other hand, will give you -1 if `number` is negative and odd. In other words, it is always worthwhile to document assumptions as obvious as they may seem.

Verify Pre-conditions

A pre-condition is the setup that occurs before an algorithm is run. This could be the process of procuring any necessary information, it could be initializing any variables, or it could be a prior step in a multi-step process. Whatever the case, invalid input into an algorithm is almost certainly going to result in invalid output. If the pre-conditions are not met, then there is little point in looking in the algorithm when there is a bug: the problem occurred before the algorithm was even run!

Best Practice 05.8 Validate pre-conditions before every algorithm

No matter how sure you are that the input data to an algorithm is correct, it is worthwhile to write a couple asserts to validate them. In the best case, you have clearly stated your assumptions in the code. In the worst case, a bug exists that is now much easier to find and fix.

C#

```
Debug.Assert (hoursWorked >= 0 && hoursWorked <= 24 * 7);
Debug.Assert (wage >= 0.00);
if (hoursWorked <= 40)
{
    pay = hoursWorked * wage;
}
else
{
    pay = (hoursWorked - 40) * (wage * 1.5) + wage * 40;
}
```

If the wages or hours worked are negative, then my pay will be wrong

Figure 05.25
Verify pre-conditions with asserts

Best Practice 05.9 Write asserts before you refactor code

Refactoring is the process of rewriting existing code to accommodate new functionality. Ideally, we just want to add a new feature or two without breaking what already works. In order to properly refactor something, we need to fully understand how it works. A great way to accomplish this is to first add asserts representing your best understanding of how everything works.

Verify Post-conditions

Post-conditions are the results of an algorithm, namely the format and composition of the output. We need to check that the output of our program is what you expect with a post-condition assert.

Best Practice 05.10 Validate post-conditions after every algorithm

C

```
/* code to sort an array of numbers */
...
/* verify that the code worked */
for (i = 1; i < length; i++)
    assert(array[i - 1] <= array[i]);
```

Make sure our sort worked before handing the new array off to the rest of the program

Figure 05.26
Verify post-conditions with asserts

Examples

Example 05.1: Compute Sales Tax

This example will demonstrate how to add asserts to an existing function.

Problem

Add asserts to the following function:

```
Java
class Purchase {
    ... code removed for brevity ...
    double computeSalesTax(double amount, double rate) {
        double tax = amount * rate;
        return tax;
    }
    ... code removed for brevity ...
}
```

Solution:

Since this function is so simple, we only need to verify the pre-condition and the post-condition. In the pre-condition, we verify that the amount of the purchase is positive. How does one even compute sales tax for negative amounts? We also need to make sure that the tax rate is between 0% and 100%.

The post-condition has two checks. First, the tax amount cannot be negative. Second, we want to make sure that the tax amount does not exceed the amount of the purchase.

```
Java
class Purchase {
    ... code removed for brevity ...
    double computeSalesTax(double amount, double rate) {
        // PRECONDITION: make sure the input is rational
        assert(amount >= 0.00);
        assert(rate >= 0.0 && rate <= 1.0);

        double tax = amount * rate;

        // POSTCONDITION: make sure the output is rational
        assert(tax >= 0.00);
        assert(tax <= amount);
        return tax;
    }
    ... code removed for brevity ...
}
```

Of course we expect neither of these asserts to fire. If they do however, they will give us a head start into finding the bug.

Example 05.2: Is Leap Year

This example will demonstrate how to add asserts to an existing function.

Problem

Add asserts to a function which computes whether a given year is a leap year.

Solution:

Swift

```
func isLeapYear(year: Int) -> Bool {
    // This only works after 1752 when the Gregorian calendar
    // took effect. Anything before that is invalid. If this
    // fires, then add code for the Julian calendar
    assert(year > 1752)

    // Not a leap year if not evenly divisible by 4
    if year % 4 != 0 {
        return false
    }
    assert(year % 4 == 0)

    // A leap year if not divisible by 100
    if year % 100 != 0 {
        return true
    }
    assert(year == 1800 || year == 1900 || year == 2000 ||
           year == 2100 || year == 2200 || year == 2300)

    // only centuries are left
    if year % 400 == 0 {
        assert(year == 2000)
        return true
    } else {
        assert(year == 1800 || year == 1900 ||
               year == 2100 || year == 2200 || year == 2300)
        return false
    }

    // if we are here, the logic of the if statements is wrong
    assert(false)
    return true
}
```

Note that this function will assert if the year 2400 or greater is added. It is better to have the occasional false-positive than to miss important test cases. In general, it is very easy to fix asserts like that. Besides, the programmer would probably want to know why such a crazy year is sent into this function.

Exercises

Exercise 05.1: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Asserts are designed to catch user errors	
Bugs indicated by asserts are easier to fix than behavior bugs	
When the assert condition evaluates to true, no action is taken	
Asserts are designed to catch compile time errors	
Asserts are designed to validate assumptions	

Exercise 05.2: Switch Case

For the following scenario, determine what would be an appropriate response. Justify your response as needed.

While reading some code, you discovered a **SWITCH/CASE** statement with no **DEFAULT**. What should you do next?

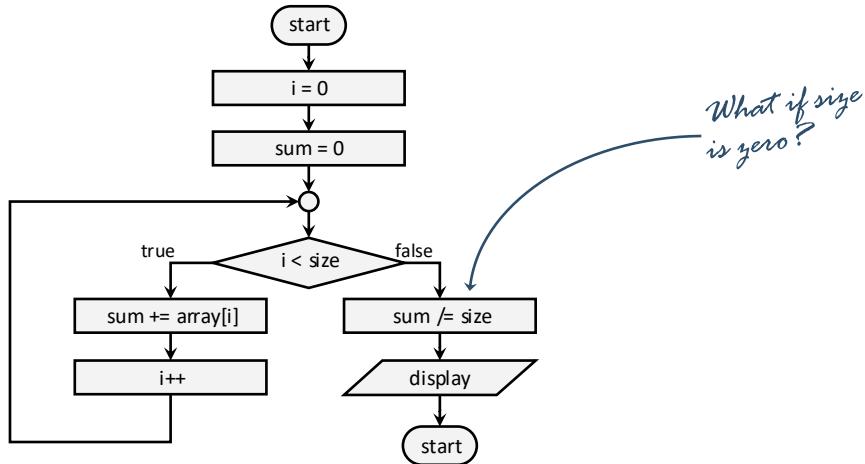
Pseudocode

```
SWITCH taxBracket
    CASE 10%
        tax ← income × 10%
    CASE 15%
        tax ← $1,510 + (income - $15,100) × 15%
    CASE 25%
        tax ← $8,440 + (income - $61,300) × 25%
    CASE 28%
        tax ← $24,040 + (income - $123,700) × 28%
    CASE 33%
        tax ← $42,170 + (income - $188,450) × 33%
    CASE 35%
        tax ← $91,043 + (income - $336,550) × 35%
```

Exercise 05.3: Flowchart

For the following scenario, determine what would be an appropriate response. Justify your response as needed.

During the design process, you discover an assumption made in a flowchart. What should you do next?



Exercise 05.4: Assert False

For the following scenario, determine what would be an appropriate response. Justify your response as needed.

You have encountered an “`assert(false)`” and you have just figured out what it means (a negative 1% will yield `lastDigit == -1`). What should you do next?

C++

```
char computeGradeSign(int grade)
{
    int lastDigit = grade % 10;
    // Display + if the grade ends with 7, 8, 9
    if (lastDigit >= 7 && lastDigit <= 9)
        return '+';

    // display - if the grade ends with 0, 1, 2
    else if (lastDigit >= 0 && lastDigit <= 2)
        return '-';

    // display nothing if the grade ends with 3, 4, 5, 6
    else if (lastDigit >= 3 && lastDigit <= 6)
        return ' ';

    assert(false);
}
```

Exercise 05.5: Index

For the following scenario, determine what would be an appropriate response.
Justify your response as needed.

You have just encountered a comment in the code stating that an index had better be less than the number of elements. What should you do next?

C#

```
class Collection
    // Fetch one element from the collection
    public override double getElement(int index)
    {
        // The index had better be less than list.Count!
        return list[index];
    }

    // code removed for brevity
}
```

Problems

Problem 05.1: Add Asserts to Resize

Add asserts to the following code:

Pseudocode

```
vector::resize(cNew)
    arrayNew ← new(cNew)
    FOR i ← 0 ... numElements
        arrayNew[i] ← array[i]
    delete(array)
    array ← arrayNew
    numCapacity ← cNew
```

Problem 05.2: Add Asserts to Search

Add asserts to the following code:

Python

```
def linearSearch2(array, search):
    for index in range(0, len(array)):
        if array[index] == search:
            return true
    return false
```

Problem 05.3: Add Asserts to Grade

Add asserts to the following code:

VB

```
Sub displayGrade(ByValue grade As String)
    ' a grade such as A+
    Dim letter As Char = grade(0) ' the letter grade: 'A'
    Dim sign   As Char = grade(1) ' the sign: '+'

    Dim message = "Your letter grade is " + grade +
                  " and your sign is "      + sign

    MsgBox(message)
End Sub
```

Problem 05.4: Add Asserts to Compute Tax

Assume the following function is defined in your program:

```
JavaScript
function assert(condition) {
    if (!condition) {
        throw "Assertion failed!";
    }
}
```

Add asserts to the following code:

```
JavaScript
function computeTax (income) {
    var tax;

    if (income >= 0 && income < 15100) {
        tax = income * 0.10;
    }
    else if (income >= 15100 && income < 61300) {
        tax = 1510 + 0.15 * (income - 15100);
    }
    else if (income >= 61300 && income < 123700) {
        tax = 8440 + 0.25 * (income - 61300);
    }
    else if (income >= 123700 && income < 188450) {
        tax = 24040 + 0.28 * (income - 123700);
    }
    else if (income >= 188450 && income < 336550) {
        tax = 42170 + 0.33 * (income - 188450);
    }
    else if (income >= 336550) {
        tax = 91043 + 0.35 * (income - 336550);
    }
    return tax;
}
```

Problem 05.5: Add Asserts to Binary Search

Add asserts to the following code:

Java

```
class BinarySearch {

    boolean binarySearch(int array[], int search)
    {
        int iLast = array.length - 1;
        int iFirst = 0;
        while (iFirst <= iLast) {
            int iMiddle = (iFirst + iLast) / 2;

            if (array[iMiddle] == search)
                return true; // found it!

            if (array[iMiddle] < search)
                iFirst = iMiddle + 1; // ignore lower half
            else
                iLast = iMiddle - 1; // ignore upper half
        }
        return false; // not found!
    }

    ... code removed for brevity ...
}
```

Problem 05.6: Add Asserts to Compute Tax

Add asserts to the following code:

C++

```
float computeTax(float income)
{
    Bracket brackets[] =
    { /* min      max   fixed   rate */
        { 0,      15100,    0, 0.10},
        { 15100,   61300,  1510, 0.15},
        { 61300,  123700,  8440, 0.25},
        {123700,  188450, 24040, 0.28},
        {188450,  336500, 42170, 0.33},
        {336500, 99999999, 91043, 0.35}
    };

    for (int i = 0; i < sizeof(brackets)/sizeof(Bracket); i++)
        if (brackets[i].min <= income &&
            brackets[i].max >= income)
            return brackets[i].fixed +
                   brackets[i].rate * (income - brackets[i].min);
    return 0.0;
}
```

Challenges

Challenge 05.1: Existing Code

In the largest program you have previously completed, add appropriate asserts. Please look at the “Preventing Bugs with Asserts” section of this chapter for hints as to where those asserts should go.

When you are finished, compile and run the program. Try to run the code through as many of the newly added asserts as possible. Were you able to get any of them to fire?

Challenge 05.2: Open Source

Find an open source project in a language of your choice. Add appropriate asserts to this code according to the guidelines in the “Preventing Bugs with Asserts” section of this chapter.

When you are finished, compile and run the program. Try to run the code through as many of the newly added asserts as possible. Were you able to get any of them to fire?

Challenge 05.3: Create an Assert

Create a custom assert in the language of your choice. Try to replicate the functionality of the asserts provided if your language (if they do exist). How close can you get to that functionality? Please see the “Building an Assert” section of this chapter for a hint how this is to be done.

Trace

Chapter 06

Trace is the process of systematically stepping through an algorithm one statement at a time to track how the values of the variables change through program execution.

Finding bugs can be very difficult and frustrating, especially if you don't know where to start. We have all been there before: the program output is not what you expect, you just know that there is a bug somewhere in the vast labyrinth that is the codebase. In desperation, you start scrolling through the code, hoping for a big obvious bug (preferably marked in red) that says "the problem is right here! Fix me!" After a few hours of futile scrolling, you are no closer than when you began. If only there was a better way! Fortunately, there is. We call this technique program trace, also known as a desk check, sketching a memory table, and trace-through.

Trace is the process of systematically stepping through the code, one statement at a time, to track the values of the variables. During this process, you play the role of the computer. With each step of the algorithm, you write down on a sheet of paper how the program would change each of the variables. This technique is effective because most bugs have their origins in an incorrectly set value.

During a trace, you play the role of the computer by systematically tracking the values of the variables at each line of code

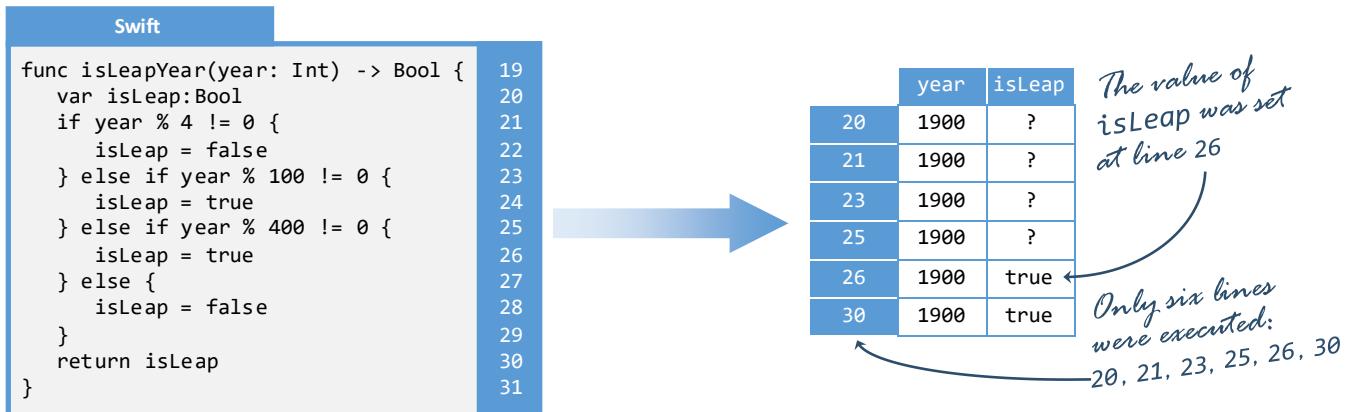


Figure 06.1
Program trace

In the above example, we know that 1900 is not a leap year. However, when we step through the program one line at a time, we realize that it returns `true`. The problem seems to be the IF statement immediately before the `return` value. Upon closer inspection, it looks like we are using `!=` instead of `==` at line 25. Subtle problems like this are easy to miss with a simple visual inspection.

Program traces such as this are very useful in finding bugs in loops and IF statements. We will also see how they can be used to narrow down a bug that may exist somewhere in a large codebase.

Conducting a Program Trace

A program trace is a technique where a table is built containing the values of variables at various stages of execution. This table represents variables in columns, stages of program execution in rows, and the values of variables at specific times in the execution in the interior boxes. This allows the programmer to be able to easily see the value of a variable at a given moment in time.

	year	isLeap
20	1900	?
21	1900	?
23	1900	?
25	1900	?
26	1900	true
30	1900	true

Figure 06.2
A trace table

You conduct a program trace in a four-step process: 1) identify test cases, 2) number the lines in the source code, 3) build the table, and 4) compare the table output to what was expected.

Step 1: Identify Test Cases

The first step is to figure out which input values you are going to run through the program and what the expected output will be. This is called a test case. Sometimes we can find a bug with a single test case, but more often than not a collection of test cases needs to be identified. A test case can be as simple as a start-condition paired with a stop condition.

Start-Conditions

The start-condition describes the input into the algorithm to be tested. In the simplest sense, this is the initial state of the variables before the algorithm is executed. If we are testing a function, then this is the input parameters.

When an algorithm is tested that is reading a file, the start-condition is the composition of the input file. Similarly, if the algorithm accepts input from the user, then the start-condition is the composition and the order of the user input.

Inputs
2001
2004
1900

Figure 06.3
Start-conditions

For example, consider our leap year function from above. Since there is only one parameter to `isLeapYear()`, the following would be a suitable set of pre-conditions: 2001, 2004, and 1900. Each of these start-conditions will result in a different program trace and will create a different trace table.

End-Condition

The end-condition describes the state of the program after the algorithm is tested. This can be the value of variables, the composition of the output file, the summation of all the output that the algorithm put on the screen, or it could be the return value of a function.

In order to determine the end-conditions, the programmer will need to manually compute the expected output. It is usually a good idea to select input so the output computation process is a little easier. Note that if you cannot determine the end-condition for whatever reason, then you will get little benefit from conducting a program trace. The trace will produce an output, but you will have no idea whether it is right or wrong!

Inputs	Outputs
2001	false
2004	true
1900	false

*Post-conditions
are a list of the
expected outputs*

Figure 06.4
End-conditions

Test Cases

Test cases are a collection of start-conditions and end-conditions which inform us of the correctness of a program. There is more to picking good test cases than meets the eye. Please see Chapter 18: Test Cases for a more in-depth discussion on the topic. For now, it is enough to say that we should select test cases which cover one or two typical start-conditions in the system, a couple which are known to be invalid, and some cases which are right on a boundary.

A test case is the combination of the start and end-condition of an algorithm representing the correct behavior of a program

Back to our leap year function, we need to remind ourselves of the logic behind a leap year:

According to the Gregorian calendar, which is the civil calendar in use today, years evenly divisible by 4 are leap years, except for centurial years that are not evenly divisible by 400. Therefore, the years 1700, 1800, 1900 and 2100 are not leap years, but 1600, 2000, and 2400 are leap years.

Here, we can see that there are five main cases. First, there is a typical non-leap year that is within a century. Examples would include 2001, 2002, 2003, 2005, 2006, ... 2099. We will call this the “Non-Leap” case. Next, there are the typical leap years, including 2004, 2008, 2012, ... 2096. We will call this case “Leap.” Next, the century years which are not evenly divisible by 400. These include 1800, 1900, 2100, 2200, etc. We will call this “Century.” The remaining years are those evenly divisible by 400: 2400, 2800, etc. We will call this “Quad-Century.” What remains? The Gregorian calendar was adopted in 1752. Therefore, any year before that is in error. From this, we will generate the following test case table:

Test Case Name	Inputs	Outputs	Rationale
Non-Leap	2001	false	Typical non-leap year
Leap	2004	true	Non-century leap year
Century	1900	false	Century, not quad century
Quad-Century	2000	true	Quad-century
Error	1750	error	Gregorian started 1752

Give each test case a short name or label

It is often useful to explain why the test case was chosen

Figure 06.5
Test cases

Step 2: Number the Lines

Every row in a trace table corresponds to a line of code. We therefore need to have some sort of numbering system connecting a single statement with a row on the table.

```
C#  
if (hours <= 40)  
{  
    pay = hours * wage;  
}  
else  
{  
    pay = (hours - 40) * (wage * 1.5)  
        + wage * 40;  
}
```

142
143
144
145
146
147
148
149
150

There are two main numbering systems: source line numbers and custom line numbers. In the former case, you utilize the line numbering feature of your code editor to uniquely identify each statement. Most editors will put line numbers on the left side or the right side of the window. This, of course, is very easy and is the most commonly used technique. The problem is that any edits made to the code will cause line numbers to shift. Of course, the process of fixing bugs necessitates editing the code, so the shifting line number problem is more common than you may think.

Figure 06.6
Source line numbers

The other technique is to add line numbers or labels as comments to the code. These are easy to add and do not shift but are not without difficulties. Say, for example, that an algorithm is labeled with A, B, and C. If you want to add another line number between B and C, then you need to awkwardly have a "B1" or something like that. Despite this, it is often very convenient because you only label statements which will have a corresponding entry in the trace table.

```
C#  
if (hours <= 40) // A  
{  
    pay = hours * wage; // B  
}  
else  
{  
    pay = (hours - 40) * (wage * 1.5) // C  
        + wage * 40;  
}
```

Figure 06.7
Custom line numbers

Regardless of how you number the statements in the algorithm, another very important decision needs to be made: how detailed will the program trace be? In other words, you could choose to have every single statement in the algorithm have representation in the program trace table or you could choose to only represent every fourth or fifth statement. An exhaustive program trace will have one entry in the trace table for every statement that changes a variable or produces output. This will skip lines of code having no impact on the algorithm (such as comments, blank lines, and such) but will capture all the details of the algorithm. A coarse program trace will group related statements into a single line number, making the resulting trace table much smaller but perhaps less detailed. Which is best? That depends on the bug you are trying to locate. Generally, most programmers start with a coarse numbering scheme and move to something more detailed if the first attempt failed.

Very coarse, only showing loop body

Very typical line numbering

Perhaps too exhaustive. More detail than needed

```
C++  
for (int i = 0; i < 10; i *= 2)  
    cout << i << endl; // 1
```

```
C++  
for (int i = 0; i < 10; i *= 2) // 1  
    cout << i << endl; // 2
```

```
C++  
for (int i = 0;  
     i < 10;  
     i *= 2) // 1  
    cout << i << endl; // 2  
                                // 3  
                                // 4
```

This will have an entry in the trace table for every statement changing or looking at our counter variable

Figure 06.8
Different ways to number lines

Step 3: Create the Trace Table

A trace table is a table containing the values of variables at various stages of execution. The column headers (along the top of the table) contain a list of the variables to be tracked. The row headers (along the left side of the table) contain the line numbers. The rest of the cells have the values of the variables.

To demonstrate how this works, we will trace a simple algorithm consisting of a loop and a conditional. We will start with an empty trace table consisting only the names of the variables to be tracked: `count`, `even`, and `odd`.



Figure 06.9
Setting up a trace table

The first line to be executed is the FOR loop, labeled with the letter A. When this line is finished, the variable `count` will contain the value 1 and the other two variables have yet to be declared. We indicate that with a slash /.

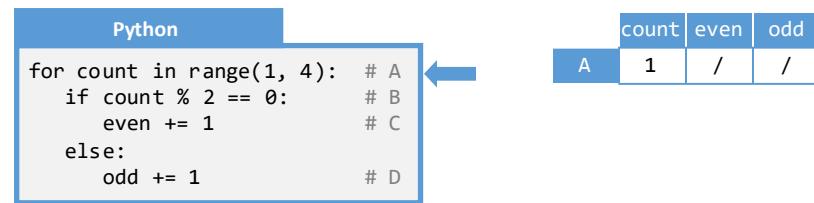


Figure 06.10
After one statement
is executed

The next line of code to be executed is the IF statement. Here, the values of the variables are not changed, but the program counter moves.

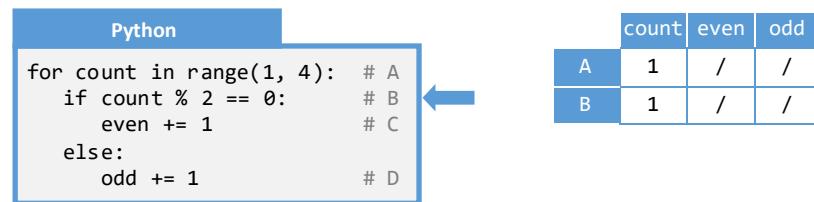


Figure 06.11
After two statements
are executed

Since `count` equal one and therefore `count mod two` equals one, this Boolean expression will evaluate to `False`. Therefore, the next line of code to be executed is D and not C. Variables get declared when they are first used in Python, and integers are initialized to zero. This means that the variable `odd` will be instantiated in line D and be set to one at the end of execution.

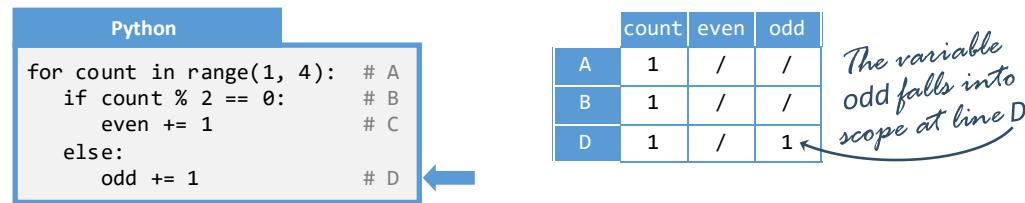


Figure 06.12
After the third
statement is executed

The second time through the FOR loop will increment the value of **count** to two. At this point, an important characteristic of the trace table becomes evident. For a given line of code, the value of the variables can be different at various stages of execution. The trace table allows you to look back in time through the course of execution and see how values change.

Python

```
for count in range(1, 4): # A
    if count % 2 == 0: # B
        even += 1 # C
    else:
        odd += 1 # D
```

Line A is executed a second time

	count	even	odd
A	1	/	/
B	1	/	/
D	1	/	1
A	2	/	1

Notice how the value of count is different than the first time through

Figure 06.13
After the fourth statement is executed

We will now skip ahead two steps and execute line C for the first time. This will instantiate the variable **even** and give it its first value.

Python

```
for count in range(1, 4): # A
    if count % 2 == 0: # B
        even += 1 # C
    else:
        odd += 1 # D
```

Line C is now executed for the first time, thereby instantiating even

	count	even	odd
A	1	/	/
B	1	/	/
D	1	/	1
A	2	/	1
B	2	/	1
C	2	1	1

Figure 06.14
After the sixth statements is executed

When the trace table is completed, then we have a full history of the execution of the algorithm.

You can see how the value of count is different in each pass through the loop

The three different iterations through the loop are clearly evident

	count	even	odd
A	1	/	/
B	1	/	/
D	1	/	1
A	2	/	1
B	2	/	1
C	2	1	1
A	3	1	1
B	3	1	1
D	3	1	2

Figure 06.15
A completed trace table

Test Cases

You create one trace table for every test case. This means that five test cases will result in five individual trace tables. A common mistake is to have one trace table somehow represent more than one test case.

Step 4: Interpret the Results

After the trace table has been completed and the values of all the variables are known, what next? How can we tell whether there is a bug and, if there is one, what the cause might be?

A completed trace table needs to be compared with the post-condition predicted in Step 1. This is the first and most important indication of whether there is a bug. Note that a discrepancy could result in how the post-condition was hand-computed or it could be a result of a defect in the design. In either case, more investigation is required. Going back to our first example of the `isLeapYear()` function in Swift, we have the following code which will be run through our five test cases: 2001, 2004, 1900, 2000, and 1750:

```
Swift
func isLeapYear(year: Int) -> Bool {
    var isLeap:Bool // (0)
    if year % 4 != 0 {
        isLeap = false // (1)
    } else if year % 100 != 0 {
        isLeap = true // (2)
    } else if year % 400 != 0 {
        isLeap = true // (3)
    } else {
        isLeap = false // (4)
    }
    return isLeap // (5)
```

Figure 06.16
A function ready
to be traced

Notice that there are five test cases. This means we will need five individual trace tables, each table starting with a different value for `year`. Our five trace tables are:

	year	isLeap		year	isLeap		year	isLeap		year	isLeap	
(0)	2001	?		(0)	2004	?	(0)	1900	?	(0)	2000	?
(1)	2001	false		(2)	2004	true	(3)	1900	true	(4)	2000	false
(5)	2001	false		(5)	2004	true	(5)	1900	true	(5)	2000	false

Figure 06.17
Trace results

Here, we have the incorrect answer for three of the five test cases; Century, Quad-Century, and Error are all incorrect.

Test Case Labels	Inputs	Outputs	Actual	
Non-Leap	2001	false	false	
Leap	2004	true	true	
Century	1900	false	true	
Quad-Century	2000	true	false	
Error	1750	error	false	

Three results are different than we expect. There are bugs!

Figure 06.18
Compare actual results
with expected results

To fix this bug, we will start by noticing that both Century and Quad-Century are based on the same IF statement. Further investigation reveals that the condition is reversed. It should be `==` instead of `!=`. The second thing we will notice is that we completely forgot to handle the case of years smaller than 1753. We will need to add a check for that. Notice that we need to renumber our code; a new statement was added between the old (0) and (1). Our new code is the following:

```

Swift
func isLeapYear(year: Int) -> Bool {
    var isLeap:Bool // (0)

    assert(year > 1752) // (1)

    if year % 4 != 0 { // (2)
        isLeap = false
    } else if year % 100 != 0 { // (3)
        isLeap = true
    } else if year % 400 == 0 { // (4)
        isLeap = true
    } else { // (5)
        isLeap = false
    }
    return isLeap // (6)
}

```

Figure 06.19
The same function as before with fixed bugs

We will rerun our tests. Notice that each trace table gets a new row and how the final table is noticeably shorter. This is because the program execution stops when the assertion fires.

	year	isLeap		year	isLeap		year	isLeap		year	isLeap		year	isLeap	
(0)	2001	?		(0)	2004	?	(0)	1900	?	(0)	2000	?	(0)	1750	?
(1)	2001	?		(1)	2004	?	(1)	1900	?	(1)	2000	?	(1)	1750	?
(2)	2001	false		(3)	2004	true	(4)	1900	false	(5)	2000	true	(6)	2004	true
(6)	2001	false		(6)	2004	true	(6)	1900	false	(6)	2000	true			

Figure 06.20
Validation of bug fixes

Collecting the results from our five test cases, we get the following:

Test Case Name	Inputs	Outputs	Actual
Non-Leap	2001	false	false
Leap	2004	true	true
Century	1900	false	false
Quad-Century	2000	true	true
Error	1750	error	assertion

With all tests passing, we can have greater confidence that our program works.

Figure 06.21
Test results

Automatic Trace

Conducting a program trace is a tedious task. This is especially true if we need to execute a lot of code or need to go through many iterations of a loop. Fortunately, we can write code to automate this task. First, consider the following program which sums the powers of two:

Figure 06.22
Code to be traced

```
C++  
  
int inc = -1;  
int sum = 0; // A  
  
for (inc = 1; inc < 10; inc *= 2) // B  
    sum += inc; // C  
  
cout << sum << endl; // D
```

*Code to be
verified with
four line
numbers*

Figure 06.23
Trace code is added

```
C++  
  
cout << "\tinc\tsum\n";  
  
int inc = -1;  
int sum = 0;  
cout << "A\t" << inc << "\t" << sum << endl; // A  
  
for (inc = 1; inc < 10; inc *= 2)  
{  
    cout << "B\t" << inc << "\t" << sum << endl; // B  
    sum += inc;  
    cout << "C\t" << inc << "\t" << sum << endl; // C  
}  
  
cout << "D\t" << inc << "\t" << sum << endl; // D
```

*Trace code
added*

When the code is executed, the hand-compute trace table on the left matches the automatically generated table on the right.

Figure 06.24
Trace results with
both hand-traced
and auto-traced tables

	inc	sum
A	-1	0
B	1	0
C	1	1
B	2	1
C	2	3
B	4	3
C	4	7
B	8	7
C	8	15
B	16	15
D	16	15

	Output	
	inc	sum
A	-1	0
B	1	0
C	1	1
B	2	1
C	2	3
B	4	3
C	4	7
B	8	7
C	8	15
D	16	15

*Automatically
produced trace
table*

Variations

There are three common variations to program trace: flares, memory dumps, and scoreboards. The first two are primarily used in automatic traces and the last is used for hand-tracing.

Flare

Imagine an application of three hundred lines of code. You run the program and it crashes. You don't know exactly where the crash happens, you just know that it happened. How do you find the offending line of code? One solution is to add a flare. In the physical world, a flare is a signal that a guard would send when the enemy is approaching his watchtower. If several guards were stationed on the battlefield, then the commander could monitor the advancing enemy by watching for flares. Software flares work much the same way.

A flare is a simple display statement added to the code indicating that execution has reached a given line

A flare is a simple display statement added to the code indicating that execution has reached a given line. They are a very coarse form of program trace where not even the values of the variables are displayed. Back to our three-hundred-line example,

you could place a display statement every thirty lines. We will number them 1, 2, 3, etc. When you rerun your program you notice that flares 1-5 are executed, but not number 6. Clearly the bug is between flares 5 and 6! To get some more detail, you add a few more flares: 5.1, 5.2, 5.3, etc. This execution reveals that 5.4 fired but not 5.5. Now we have narrowed it down to just three lines of code!

Pseudocode

```
... code removed for brevity ...
PUT #1

... code removed for brevity ...
PUT #2

... code removed for brevity ...
PUT #3

... code removed for brevity ...
```

Figure 06.26
Simple flares

Flares are very easy and simple to implement. They can be used effectively in extremely large codebases consisting millions of lines of code. With just a few iterations, it is often possible to narrow down a bug to just a handful of lines of code even if you have never seen that code before. In cases like that, an effective technique is to put a flare at the beginning of a function and at the end. If execution entered a function but never left, then the bug exists there. It is also helpful to add code to display the function parameters. See the following program for an example:

Python

```
def computeTax(income):
    print("BEGIN computeTax income=", income)

    ... code removed for brevity ...

    print("END computeTax tax=", tax)
```

Figure 06.27
Begin and end flares

Memory Dump

A memory dump is another very coarse version of automatic program trace. Rather than adding a row to a trace table when a statement is executed, a report of all the relevant variables is displayed. Instead of putting the variable names in the column headers as we do with a trace table, they are listed on their own line next to the values they contain. Memory dumps are placed in one or two key locations in the code to give the programmer an idea of how the code is functioning. To see how this works, consider the following code which will add an element to the end of a list and remove an element from the head:

A memory dump is display code to put the values of variables on the screen so they can be visually verified by the programmer

```
C#  
  
// initial array of strings  
List<string> stuff = new List<string> {"one", "two", "three"};  
  
// add an element to the tail and remove one from the head  
stuff.Add("four");  
stuff.Remove(0);
```

Figure 06.28
Code to be verified using a
memory dump

We don't need to create a trace table here. We just want to know the state of **stuff** at the end of execution. We can accomplish this by writing some memory dump code:

```
C#  
  
// initial array of strings  
List<string> stuff = new List<string> {"one", "two", "three"};  
  
// add an element to the tail and remove one from the head  
stuff.Add("four");  
stuff.Remove(0);  
  
// verify this worked as expected  
int index = 0;  
foreach (int element in stuff)  
{  
    Console.WriteLine($"stuff[{index}] : {element}");  
    index++;  
}  
Console.WriteLine($"num elements: {stuff.Length}");
```

Figure 06.29
Memory dump code

The output will look like this:

```
Output  
  
stuff[0] : two  
stuff[1] : three  
stuff[2] : four  
num elements: 3
```

Figure 06.30
Memory dump output

A memory dump is useful when there are too many variables in an algorithm to reasonably put in a single trace table. It is also useful when there is a large array or when there is a change in the number of variables in scope.

Scoreboard

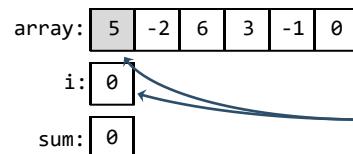
A scoreboard is an alternate form of the manual program trace technique. Instead of building a trace table, values are updated in slots called a scoreboard. Perhaps this is best demonstrated by example. Consider the JavaScript program illustrated in Figure 06.31. Notice that there are three variables: `array`, `i`, and `sum`. We start the test at the FOR loop. Note that the first two lines of code will be removed for brevity's sake. The scoreboard technique involves updating the values of the variables as the program executes. Unlike the trace table, we do not preserve a history of the values of the variables; once a value has been overwritten, we effectively forget the old value.

```
JavaScript
var array = [5, -2, 6, 3, -1, 0];
var sum = 0;

for (var i = 0; i < array.length; i++)
    if (array[i] < 0)
        array[i] = 0;
    else
        sum++;
```

Figure 06.31
Code to be verified

```
JavaScript
for (var i = 0; i < array.length; i++)
    if (array[i] < 0)
        array[i] = 0;
    else
        sum++;
```



Initially the
index is zero

Figure 06.32
Beginning of execution

From here, the next line of code that will change anything is the ELSE statement. This will update the value of `sum`.

```
JavaScript
for (var i = 0; i < array.length; i++)
    if (array[i] < 0)
        array[i] = 0;
    else
        sum++;
```

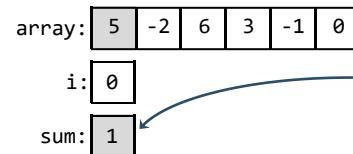
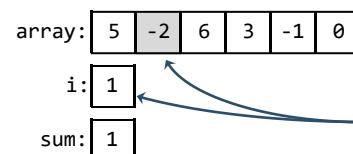


Figure 06.33
ELSE code executed

Next, execution returns to the FOR loop where `i` will be incremented by one.

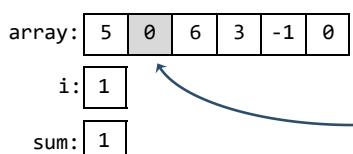
```
JavaScript
for (var i = 0; i < array.length; i++)
    if (array[i] < 0)
        array[i] = 0;
    else
        sum++;
```



Now the index
is referring
to the second
element
Figure 06.34
In the FOR loop again

This time we will go into the TRUE condition of the IF statement. Now the value in `array` will be set to zero. Notice that we overwrite the that slot in memory, thereby forgetting the previous value. It is easy to represent arrays this way, but harder to look back at what happened.

```
JavaScript
for (var i = 0; i < array.length; i++)
    if (array[i] < 0)
        array[i] = 0;
    else
        sum++;
```



The value of
array[1] is
changed here
Figure 06.35
In the TRUE condition

Examples

Example 06.1: Decision Trace

This example will demonstrate how to perform a program trace with IF statements and multiple test cases.

Problem:

Create a program trace for the following code with the input \$12,120 and \$89,321. We expect to get 10% and 25% from this algorithm.

```
JavaScript
if (tax < 61300.00) {
    if (tax < 15100.00) {
        bracket = 0.10;
    } else {
        bracket = 0.15;
    }
} else {
    if (tax < 123700.00) {
        bracket = 0.25;
    } else {
        bracket = 0.28;
    }
}
368
369
370
371
372
373
374
375
376
377
378
379
380
```

Solution:

There are two trace tables for the two test cases. Notice that different lines of code get executed according to the results of the Boolean expressions in the IF statements.

	tax	bracket
368	12120	?
369	12120	?
370	12120	0.10

	tax	bracket
368	89321	?
374	89321	?
375	89321	?
376	89321	0.25

In both cases, the program output matched the expected results:

	Pre	Post	Actual
10%	12120	10%	0.10
25%	89321	25%	0.25

Example 06.2: Loop Trace

This example will demonstrate how to perform a program trace with one test case and a FOR loop.

Problem:

Create a program trace for the following code:

Python

```
iUp = 0
iDown = 10
while iUp < iDown:
    print(iUp, " - ", iDown)
    iUp += 1
    iDown -= 1
```

Solution:

The trace table is built from top to bottom, starting at line # 1 and moving down. Notice that the various iterations in the loop are colored. The first iteration is grey, the second white, and the third grey. This is done to make it easier to read.

	iUp	iDown	Output
1	0	8	
2	0	8	
3	0	8	0 - 8
4	1	7	
2	1	7	
3	1	7	1 - 7
4	2	6	
2	2	6	
3	2	6	2 - 6
4	3	5	
2	3	5	
3	3	5	3 - 5
4	4	4	
2	4	4	

Exercises

Exercise 06.1: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Each number along the left side of a trace table is a test case	
In a test case, end-conditions are unnecessary	
A flare is a type of automatic trace	
Variable names are on the row headers in a trace table	
Three test cases will have three table rows	

Exercise 06.2: Simple Trace

Create a program trace for the following code:

Pseudocode

```
convert
    yards ← 7           # A
    feet ← yards × 3    # B
END
```

Exercise 06.3: Decision Trace

Create a program trace for the following code:

Pseudocode

```
age ← 16          # A
IF age ≥ 18       # B
    can_vote ← true # C
ELSE
    can_vote ← false # D
```

Exercise 06.4: Loop Trace

Create a program trace for the following code:

Pseudocode

```
number ← 1          # A
WHILE number ≤ 5    # B
    number ← number + number # C
```

Exercise 06.5: Two Test Case Trace

Consider the following code:

Pseudocode

```
GET number          # A
number ← number x -1 # B
```

Create a program trace for two test cases:

Inputs	Outputs
4	-4
-2	2

Problems

Problem 06.1: Trace a FOR Loop

Create a program trace for the following code:

```
C
{
    int sum = 0;
    int count;

    for ( count = 1;
          count < 9;
          count *= 2)
        sum += count;
    printf("sum = %d\n", sum);
}
```

Problem 06.2: Program Trace

Create a program trace for the following code:

```
Swift
let legs = ["snake": 0, "snail": 1, "person": 2,
            "dog": 4,      "butterfly": 6, "crab": 8]
let total = 0;
for (name, count) in legs {
    total += count;
}
print ("there are \(total) legs")
```

Problem 06.3: Program Trace

Create a program trace for the following code:

```
Kotlin
val count = 0

while (count < 5) {
    count++
}

println("count == $count")
```

Problem 06.4: IF Trace

Consider the following code:

Pseudocode
PROMPT for grade GET numberGrade IF numberGrade ≥ 80 IF numberGrade ≥ 90 letterGrade ← A ELSE letterGrade ← B ELSE IF numberGrade ≥ 70 letterGrade ← C ELSE letterGrade ← D PUT letterGrade

Create a manual program trace for these six test cases:

	Pre	Post	Actual
A	95	A	
B	82	B	
C	79	C	
D	60	D	
F	59	F	
error high	101	error	
error low	-1	error	

Problem 06.5: Automatic Trace

Add automatic trace code to the following program.

Python

```
# Prompt for grade
number = int(input("Please enter a positive number: "))

# Initialize sum
sum = 0

# Count it up
for count in range(1, number):
    sum += count

# Report
print("The sum is:", sum)
```

Run this code with the following test cases. If you discover a bug, fix the bug and rerun the test cases.

	Input	Output	Actual
small	3	6	
zero	0	0	
error	-1	error	

Challenges

Challenge 06.1: Binary Search

The following is the pseudocode algorithm for the binary search.

Pseudocode

```
search(array, search, iBegin, iEnd)
    IF iBegin > iEnd
        RETURN FALSE

    iMiddle ← (iBegin + iEnd)/2

    IF array[iMiddle] = search
        RETURN TRUE

    IF search > array[iMiddle]
        RETURN search(array, search, iMiddle + 1, iEnd)
    ELSE
        RETURN search(array, search, iBegin, iMiddle - 1)
END
```

Translate this code to the programming language of your choice and add automatic trace code. Run the following tests:

	Pre	Post	Actual
Find	array = {1,2,3} search = 2 iBegin = 0 iEnd = 2	TRUE	
Not found	array = {1,2,3} search = 7 iBegin = 0 iEnd = 2	FALSE	
Duplicate	array = {3,6,8,8,19} search = 3 iBegin = 0 iEnd = 4	TRUE	
Duplicate Not Found	array = {8,8,8,8,8} search = 3 iBegin = 0 iEnd = 4	FALSE	

Is there a bug with this algorithm? If so, please fix the bug and run your automatic trace again.

Challenge 06.2: Bubble Sort

The following is the algorithm for the bubble sort:

Pseudocode

```
FOR iSpot ← numItems ... 0
    FOR iCheck ← 0 ... iSpot - 1
        IF array[iSpot] < array[iCheck]
            swap( array[iSpot], array[iCheck] )
```

Please do the following:

1. Translate this pseudocode into a programming language of your choice.
2. Identify three test cases: an array of 5 numbers already sorted, an array of 5 numbers sorted in reverse, and an array of 5 numbers with random values.
3. Perform a manual program trace using these three test cases. Fix any bugs.
4. Add automatic trace code to your algorithm. Run the same three test cases. Again, if you encounter a bug, fix it.

Challenge 06.3: Insertion Sort

The following is the algorithm for the insertion sort:

Pseudocode

```
sortInsertion(array, n)
    FOR iPivot ← n - 2 ... 0
        valuePivot ← array[iPivot];
        iInsert ← binarySearch(array, valuePivot,
                               iPivot + 1, n - 1)
        iInsert--
        for iShift ← iPivot ... iInsert
            array[iShift] ← array[iShift + 1]
        array[iShift] ← valuePivot
```

Please do the following:

1. Translate this pseudocode into a programming language of your choice.
2. Identify three test cases: an array of 5 numbers already sorted, an array of 5 numbers sorted in reverse, and an array of 5 numbers with random values.
3. Perform a manual program trace using these three test cases. If you encounter a bug, then fix it.
4. Add automatic trace code to your algorithm. Run the same three test cases. Again, if you encounter a bug, fix it.

Challenge 06.5: Unnamed Sort

The following is the algorithm for the insertion sort:

Imagine a teacher sorting a collection of assignments by the student's last name. She picks up the stack of papers (representing the unsorted array) and looks for the student's name that sorts last (the highest value). Because the stack is unsorted, she must perform a linear search. When the highest value is found, she places it on her desk (representing the sorted portion of the array) and continues by looking for the next highest value. As she continues, the stack of papers in her hand (the unsorted array) gets smaller and the stack of papers on the desk (the sorted array) gets larger. When there are no papers remaining in her hand, the stack on the desk is the resulting sorted list.

A more detailed description:

A central idea of the sort is the pivot which separates the side of the array that is already sorted from the side that remains to be sorted. In the first iteration, **iPivot** refers to the rightmost slot (**iPivot = num - 1**). This represents the slot where the largest element in the unsorted side of the array must go.

Now it is necessary to find the index of the largest element to the left of the pivot. Two variables are required: **iCheck** representing the index in the unsorted side of the array that is currently being searched, and **iLargest** representing the largest element found thus far. If the element under **iCheck** is greater than **iLargest**, then **iLargest** receives the value under **iCheck**. This continues as **iCheck** goes from the slot one to the left of **iPivot** all the way to slot 0. Now all that remains is to swap the element under **iPivot** with the element under **iLargest** if the element under **iLargest** happens to be bigger than the one under **iPivot**.

Please do the following:

1. Create the pseudocode describing the algorithm.
2. Verify your algorithm by generating test cases and performing a trace.
3. Compute the algorithmic efficiency of this sorting algorithm.

Decisions

Chapter 07

Using decision structures more effectively can increase the maintainability and performance of your code. They are among the most commonly used constructs in every programming language.

Decision statements (IF, SWITCH, and conditional expressions) are among the most common programming constructs used in source code, constituting approximately 20% of all statements. They are ten times more commonly used than loops (FOR, WHILE, and DO-WHILE loops)! Only function calls are used more frequently than decision statements.

As with most aspects of programming, there is not a single issue responsible for all decision statement mistakes. There certainly is not a single technique which will solve all decision problems. To make matters worse, some best practices seem to run contrary to each other. Such is the world of software development. If it were easy, then anyone would be able to do it! However, there is some consolation. Many mistakes are made in ignorance because programmers were blindsided by some important consideration. Most of the time, just being aware of the issues is enough to design effectively. There are four classifications of decision-making structures in software:

Level	Summary
One Option	Simplifying Boolean expressions
Two Options	Put the most common condition in the TRUE clause
A Few Options	Common conditions go first, least common go last
Many Options	Use SWITCH/CASE statements effectively

Though the considerations overlap with each of these types of decision-making structures, each highlights a unique component to using decisions effectively.

One Option

The simplest form of a decision is a simple Action/No-Action IF statement. This is characterized by an IF statement, a single Boolean expression, and a collection of statements inside the TRUE condition.

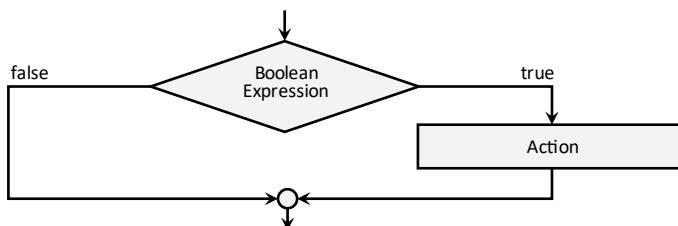


Figure 07.1
Flowchart of a
one option decision

The most common mistake made in a one option decision structure is to have an overly complicated Boolean expression. To see how this works, consider the following code to express whether an individual is eligible for an income tax child tax

credit. If the individual makes more than \$110,000 or if the child is greater than 17 years old, then no child tax credit is possible. Otherwise, the tax burden is decreased by \$1000. The C# code for this is represented in Figure 07.2:

```
C#
if (! (! (income <= 110000.00 ) || age >= 17))
{
    tax -= 1000.00;
}
```

Figure 07.2
Complex Boolean
Expression

This is quite difficult to understand and is unnecessarily slow to execute. With a little work, we can simplify this statement to the following:

```
C#
if (income <= 110000.00 && age < 17)
{
    tax -= 1000.00;
}
```

Figure 07.3
Simplified Boolean
Expression

When working with Boolean expressions, it is more convenient to use the mathematical notation for logical operators rather than the computational notation. The mathematical notation for the logical operators is the following:

Name	Code Notation	Mathematical Notation
And	&&	\wedge
Or		\vee
Not	!	\neg
XOR	\wedge	\oplus

There are algebraic transforms that you can make with Boolean expressions like what you can make with numbers. These identities can be applied to Boolean expressions to simplify them, making them easier to understand and faster to execute. These identities are listed in the table below. It is worthwhile to memorize this list as it will surely come in handy while simplifying Boolean expressions.

Name	Boolean Algebra Identity	
Annulment	$X \wedge \text{false} \Leftarrow \text{false}$	$X \vee \text{true} \Leftarrow \text{true}$
Identity	$X \wedge \text{true} \Leftarrow X$	$X \vee \text{false} \Leftarrow X$
Idempotent	$X \wedge X \Leftarrow X$	$X \vee X \Leftarrow X$
Complement	$X \wedge \neg X \Leftarrow \text{false}$	$X \vee \neg X \Leftarrow \text{true}$
Commutative	$X \wedge Y \Leftarrow Y \wedge X$	$X \vee Y \Leftarrow Y \vee X$
Simplification	$X = \text{true} \Leftarrow X$	$X = \text{false} \Leftarrow \neg X$
Associative	$X \wedge (Y \wedge Z) \Leftarrow (X \wedge Y) \wedge Z$	$X \vee (Y \vee Z) \Leftarrow (X \vee Y) \vee Z$
Distributive	$X \wedge (Y \vee Z) \Leftarrow (X \wedge Y) \vee (X \wedge Z)$	$X \vee (Y \wedge Z) \Leftarrow (X \vee Y) \wedge (X \vee Z)$
De Morgan	$\neg(X \vee Y) \Leftarrow \neg X \wedge \neg Y$ $\neg(X \wedge Y) \Leftarrow \neg X \vee \neg Y$	
Absorption	$X \wedge (X \vee Y) \Leftarrow X$	$X \vee (X \wedge Y) \Leftarrow X$
Inverse Inequality	$X < Y \Leftarrow !(X \geq Y)$	$X > Y \Leftarrow !(X \leq Y)$
Double Negative	$\neg\neg X \Leftarrow X$	

Back to our child tax credit example, we can see if we can derive the simplified mathematical expression from the original one:

Boolean Expression	Description
<code>!(!income <= 110000.00) (age >= 17)</code>	Original
<code>!!(income <= 110000.00) && !(age >= 17)</code>	De Morgan
<code>income <= 110000.00 && !(age >= 17)</code>	Double Negative
<code>income <= 110000.00 && age < 17</code>	Inverse Inequality

With numeric algebraic expressions, it is often difficult to know whether you got the transform correct. The only way to tell for sure that it will work for all values in the variables is to write a mathematical proof. Fortunately, things are much easier with Boolean algebra. Because there are a very small number of possible values, we can simply enumerate them. This is called a truth table. In the leftmost column, we enumerate all the possible values for the variables. We then compute the values according to the expressions on the subsequent column headers.

For example, say you wished to assure yourself that $\neg(A \vee B) \leq \neg A \wedge \neg B$. The following truth table will establish these Boolean expressions are equivalent for all values of A and B:

	$\neg(A \vee B)$	$\neg A \wedge \neg B$
A = false B = false	true	true
A = false B = true	false	false
A = true B = false	false	false
A = true B = true	false	false

Notice that the values for $\neg(A \vee B)$ column are identical to those of $\neg A \wedge \neg B$. Since all possible combinations of A and B are covered, this means that the two expressions are equivalent. This is slightly more complicated when non-Boolean values are utilized in our Boolean expression, but the principle is the same. Let's look back to our child tax credit example:

	<code>!(!inc <= 110000) age >= 17)</code>	<code>(inc <= 110000 && age < 17)</code>
inc = \$120,000, age = 16	false	false
inc = \$120,000, age = 18	false	false
inc = \$100,000, age = 16	false	false
inc = \$100,000, age = 18	true	true

Since the truth table reveals that `!(!income <= 110000) || age >= 17` has the same values as `(income <= 110000 && age < 17)` for all relevant values of `income` and `age`, we can see that our Boolean expressions are equivalent.

Best Practice 07.1 Simplify every Boolean expression

It is often the case that needlessly complex Boolean expressions find their way into the code because the developer stuck with the first solution that presented itself. It could be that the business logic behind the Boolean expression was expressed in a convoluted format, or it could be that it became complex as the developer expressed the business logic in terms of the available variables. Whatever the cause, there are a few warning signs: many parentheses, the use of “not,” and the use of a single variable more than once. When you notice these things, take a moment to untangle and simplify the Boolean expressions.

Two Options

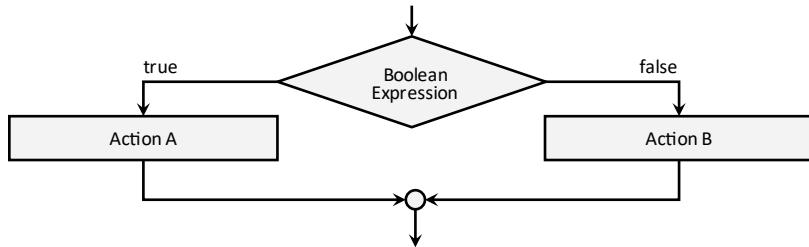


Figure 07.4
Flowchart of a two-option decision

A two-option IF statement occurs when the program diverts flow to one of two sets of statements. Note that from a logical perspective, it does not matter which part of the IF statement corresponds to the TRUE option, and which corresponds to the FALSE or ELSE option. It turns out, however, that there are significant performance implications in this selection. Generally, the most likely of the two options should be the TRUE condition. To understand why this is the case, it is necessary to understand how modern CPUs execute conditional jumps.

Each instruction executed on a CPU goes through several stages of processing. These stages are instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MA), and register write-back (WB). In the early days of computing, a CPU would execute one instruction, and then when the instruction was complete, the next instruction would be executed. Note that with this model, the instruction fetcher (IF) would be idle 80% of the time. The same would be true for most parts of the CPU.

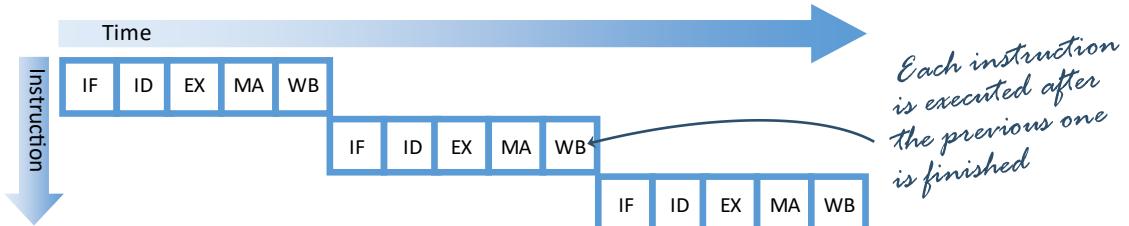


Figure 07.5
Three instructions executed on a CPU

In the early 1990s, CPU designers found a way to speed up execution without making the individual pieces faster. This was accomplished with pipelining. Here, while one instruction is being decoded, a second instruction is fetched. By the time the first instruction is finished, there are four other instructions in various stages of processing. Thus, every part of the CPU is busy all the time.

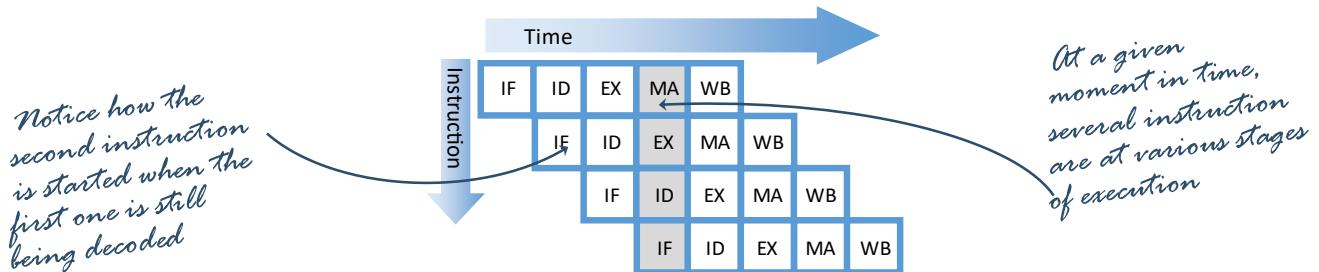


Figure 07.6
Pipelined instructions executed on a CPU

On the surface, it appears that this results in an automatic five-fold speedup. There is a problem, however. When the CPU encounters a conditional jump (an IF statement), it is unsure which instruction will be next. In fact, it won't know until after the Boolean expression is fully evaluated. Rather than wait for the results to be computed, the CPU assumes the value is TRUE. In other words, the CPU starts to execute the statements in the TRUE part of the IF statement even though the outcome of the controlling Boolean expression is unknown. When the Boolean expression finally resolves to TRUE, then all is well and the CPU continues. When it is FALSE, then all the work in the pipeline beyond the Boolean expression is discarded. The CPU then starts to execute the statements corresponding to the FALSE condition. Clearly this process is much faster when the Boolean expression is always TRUE. The question is: how much faster? That depends on the specific architecture of the CPU executing your program. If the CPU has a "deep pipeline," there may be a dozen or more statements being executed at once. In other words, the more modern the CPU, the bigger the penalty.

Unfortunately, this is a bit more complicated than that. There is an entire science of branch prediction, which is about determining whether to default to TRUE or FALSE at a given conditional jump. Some always guess TRUE (called static branch prediction), others remember the last time the conditional jump was performed (called one-level branch prediction), others the last several times (called multi-level branch prediction). A quick experiment will reveal which is favored on your computer:

```
C++  
{  
    int optionA = 1;  
    int optionB = 1;  
    bool boolean;  
  
    // start the timer  
    int msBegin = clock();  
  
    for (int i = 0; i < number; i++)  
    {  
        boolean = (optionA + optionB) && true;      // optionA  
        // boolean = (optionA + optionB) && false;    // optionB  
        // boolean = (optionA + optionB) & 0x0001;      // alternate  
  
        // perform the experiment  
        if (boolean)  
            optionA++;  
        else  
            optionB++;  
    }  
  
    // report the results  
    int msEnd = clock();  
    int elapse = msEnd - msBegin;  
    cout << "\t" << elapse << "ms\n";  
}
```

Figure 07.7
Instrumentation code to measure the CPU's branch prediction algorithm

In the case of the author's CPU, alternating options were significantly slower than always selecting `optionA` or `optionB`.

Best Practice 07.2 Create an experiment to determine the branch prediction algorithm for your CPU

When making performance-related decisions on high-stakes code, it is a good idea to write some instrumentation code to validate your decisions. Generally, most performance optimization decisions should be backed up with experimental data.

A Few Options

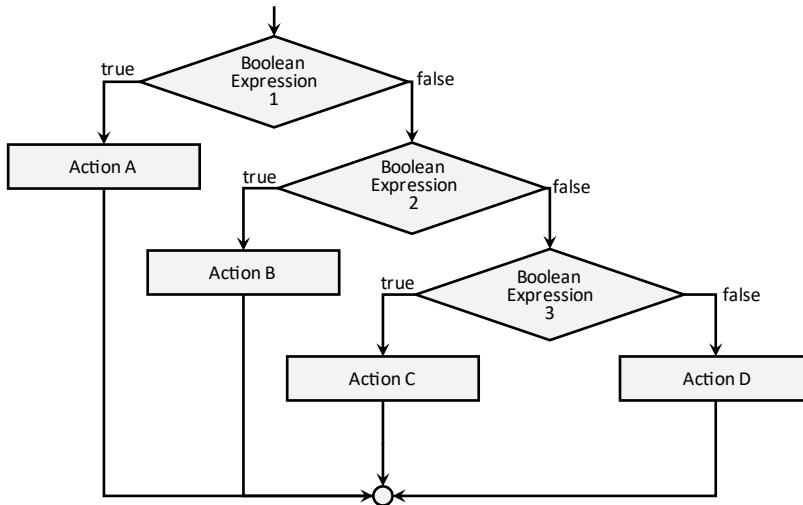


Figure 07.8
Flowchart of a
few-option decision

Many decision structures in code are not binary, instead containing three or more options. The most common way to represent these is with an IF/ELSE-IF structure. This corresponds to the following pseudocode:

Pseudocode
IF Boolean Expression 1 Action A ELSE IF Boolean Expression 2 Action B ELSE IF Boolean Expression 3 Action C ELSE Action D

Figure 07.9
IF/ELSE-IF
in pseudocode

There are a few important things to notice about this structure. First, observe that if Boolean Expression 1 (BoolExp1) evaluates to TRUE, then we never evaluate Boolean Expression 2 (BoolExp2) or 3 (BoolExp3). The second observation is that we can order the Actions any way we want, though with a considerable amount of work to the accompanying Boolean expressions.

Best Practice 07.3 Avoid redundancy in Boolean expressions

Multi-way IF statements have several Boolean expressions by necessity. If BoolExp1 evaluates to TRUE, then you can assume that all the statements within the body of that IF statement are also under that TRUE condition. There is no need to evaluate BoolExp1 again! The same can be said about the FALSE component of the IF statement; BoolExp1 will certainly evaluate to FALSE for all the statements contained therein.

A common mistake programmers make in creating multi-way IF statements is to duplicate BoolExp1 in the other Boolean expressions. To see this point, consider the following code to determine a letter grade from a number grade:

Figure 07.10
A redundant clause

```
Pseudocode
IF grade ≥ 90%
    letter ← 'A'
ELSE IF (grade ≥ 80% ) and (grade < 90% )
    letter ← 'B'
ELSE ...
...
```

This check is redundant; we already know grade < 90% due to the previous grade ≥ 90%

The grade $< 90\%$ component of BoolExp2 is redundant. We could better say:

Figure 07.11
The redundant clause
is removed

```
Pseudocode
IF grade ≥ 90%
    letter ← 'A'
ELSE IF grade ≥ 80%
    letter ← 'B'
ELSE ...
...
```

We removed the redundant clause

Best Practice 07.4 Put the most common case in a multi-way IF statement first

Multi-way IF statements work in such a way that once the statement body is executed, the instruction pointer leaves the entire sequence of IF/ELSE-IF statements. There may be a hundred ELSE-IF statements in the sequence, but if the first body is executed, then other ninety-nine are skipped. It is therefore much more efficient to put the most common cases at the front and the least common cases at the back. Consider the following code again that computes letter grades from number grades:

Figure 07.12
An inefficient design
puts the least
common case first

```
Pseudocode
IF grade < 60%
    letter ← 'F'
ELSE IF grade < 63%
    letter ← 'D- '
ELSE IF grade < 67%
    letter ← 'D'
...
...
```

Least common case is first for some reason

When we look at the distribution of grades in the target population, we come to realize that Ds and Fs are somewhat rare, but As and Bs are common. Since there are 12 different grades, we are going through 10–12 Boolean expressions in the common case and 1–2 in the exceptional case. It would be more efficient to reorder the statements so the most common case is on top.

Figure 07.13
The common case
should be on top

```
Pseudocode
IF grade > 93%
    letter ← 'A'
ELSE IF grade > 90%
    letter ← 'A- '
ELSE IF grade < 87%
    ...
...
```

It is much more efficient to have the common case on top

Best Practice 07.5 Use nested IF statements when there are several options

Imagine a problem that has 128 IF-ELSE statements where each condition has an equal chance of being used. This means that, on average, 64 Boolean expressions will need to be evaluated before the correct statement is found. This is $O(n)$ where n is the number of Boolean expressions. Instead, imagine a design where the first IF statement divides the problem state in half, the second divides it in half again, and so on. Now, instead of 128 Boolean expressions in the worst case and 64 in the common case, we have only 7. It is no longer $O(n)$, but now $O(\log_2 n)$. Back to our letter grade example:

Pseudocode

```
IF grade ≥ 80% ←
  IF grade ≥ 90% ←
    IF grade ≥ 93% ←
      letter ← 'A'
    ELSE
      letter ← 'A-'
  ELSE
    IF grade ≥ 87%
      letter ← 'B+'
    ELSE IF ≥ 83%
      letter ← 'B'
    ELSE
      letter ← 'B-'
  ELSE
    ...
  ...
```

We can compute for fewer Boolean expressions by nesting the If statements

Figure 07.14
Nested IF statements are much more efficient

As with all performance optimizations, it is worthwhile to instrument this code (see Chapter 03 Metric: Efficiency for details on how this is accomplished) to make sure that the most common condition is what you expect.

Best Practice 07.6 Avoid empty bodies in IF statements

It is not uncommon to find an IF/ELSE-IF sequence with an empty body. The following is an example:

Pseudocode

```
IF grade MOD 10 ≥ 7
  letter += '+'
ELSE IF grade MOD 10 ≥ 3
  ←
ELSE
  letter += '-'
```

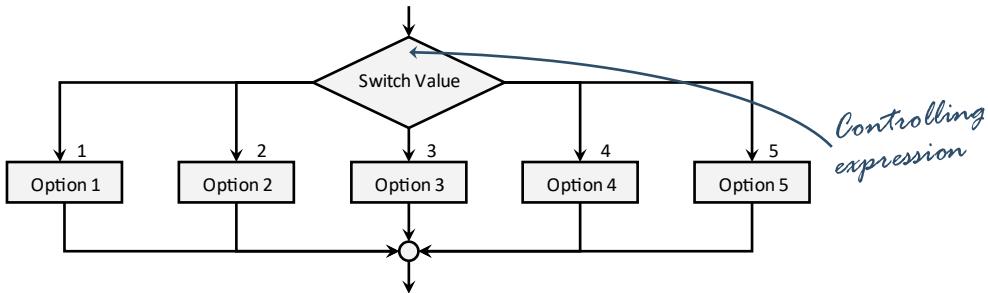
This is left blank, but why?

Figure 07.15
Avoid empty bodies in IF statements

At best, an empty body in an IF statement is a sign of sloppy code. It reflects poorly on the programmer who wrote it. More commonly, however, it belies a more serious problem: it indicates a redundant or unnecessary Boolean expression. It could even indicate that a case was missing or not properly handled.

Many Options

Figure 07.16
Flowchart of a
many-option decision



As a rule, when there are more than 2–3 decision options, one should think about using a SWITCH/CASE statement rather than IF/ELSE-IF statements. Most programmers consider SWITCH/CASE statements to be much more understandable and most compilers have very efficient implementations for them. Using SWITCH/CASE statements, however, can be difficult and is not always possible. Most languages require CASE labels to have single values, not a range of values or Boolean/algebraic expressions. If you cannot reduce your many-option decision problem into a single integral value, then IF/ELSE-IF statements might be your only possibility.

There are four general strategies that compilers use with SWITCH/CASE statements: IF/ELSE-IF statements, iteration, binary search, and table lookup. Most compilers use several of these strategies depending on the composition of the SWITCH/CASE statement. In cases like this, it behooves the programmer to write the code in such a way that the most efficient strategy is used.

IF/ELSE-IF

The easiest implementation, and least efficient, is to turn the SWITCH/CASE statement into a series of IF/ELSE-IF statements. This is typically a tiny bit faster than IF/ELSE-IF statements implemented by the programmer because all the IF conditions are tightly packed and therefore are closer together in memory. As a rule, the closer together code is in memory, the faster it executes.

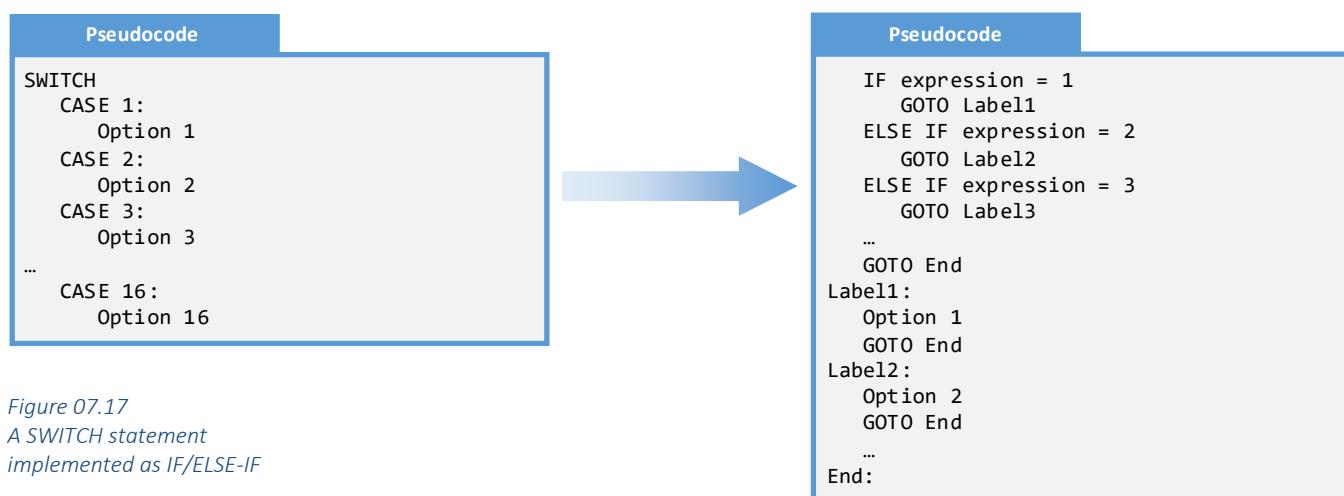
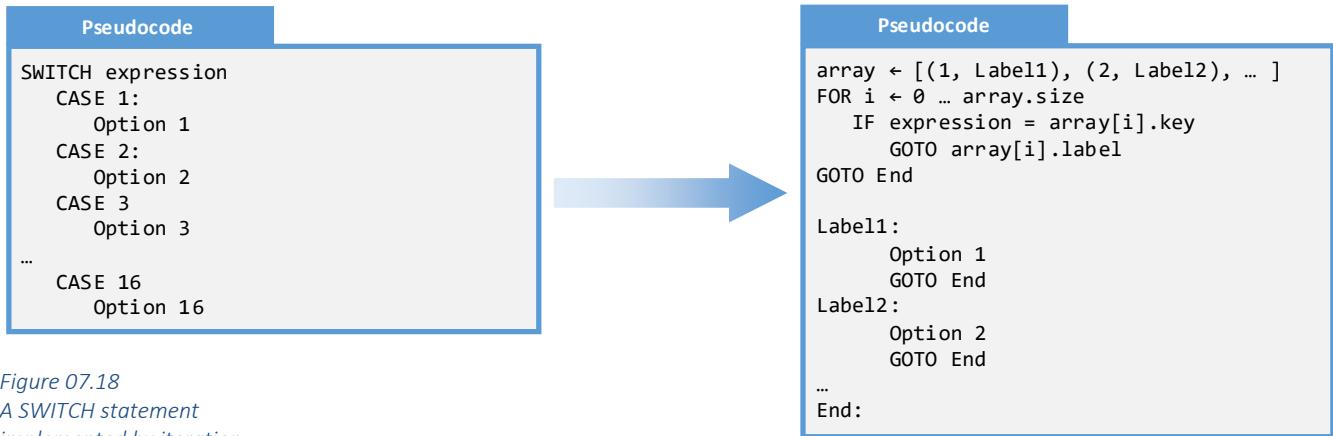


Figure 07.17
A SWITCH statement
implemented as IF/ELSE-IF

Note that this is $O(n)$ where n is the number of case labels. The worst case is when the expected value is not found because all the other cases must be checked first. Because this is the slowest implementation, we should write our code to avoid it whenever possible. If a language allows the controlling expression to be non-integral (such as JavaScript, C#) or when expressions are allowed in case labels (such as with Perl, Swift, Kotlin), then this is often the implementation chosen by the compiler. These language features are certainly convenient, but often come at a price.

Iteration

A second and more efficient implementation involves turning the SWITCH/CASE statement into a loop. Here, every CASE statement has two parts: the CASE label and the associated code. This is represented as a pair: the key representing the CASE label and a pointer to the location of the code.



*Figure 07.18
A `SWITCH` statement
implemented by iteration*

The presence of a loop is a sure indication that this is $O(n)$, the worst case being when the value is not found in the loop (the `DEFAULT` case). That being said, it is considerably faster than the `IF/ELSE-IF` implementation because the code is more compact and the Boolean expression is trivial.

What would cause a compiler to use iteration as a method for a `SWITCH/CASE` implementation? The most common reason is that a non-integral controlling variable is used. For example, if a programming language allows for strings or floating-point numbers to be in the CASE label, then the only way to determine which label matches the controlling expression is to loop through them all. This is true with languages such as Java.

Best Practice 07.7 Use integers in CASE labels whenever possible

A simple change that can be made in many codebases is to avoid non-integral controlling variables. Though the code may be more readable using another data type, it almost certainly comes with a performance penalty.

Binary Search

Some languages offer a more sophisticated and efficient algorithm for SWITCH/CASE statements. They create a binary search through the range of labels. This is like the nested IF statement implementation discussed previously in the “A Few Options” section.

To accomplish this, the compiler sorts the case labels. From this list, it finds the midpoint and inserts an IF statement to discern if the controlling expression value is in the upper range or the lower range. From the resulting half, another IF statement is created. This continues until the individual case label is isolated.

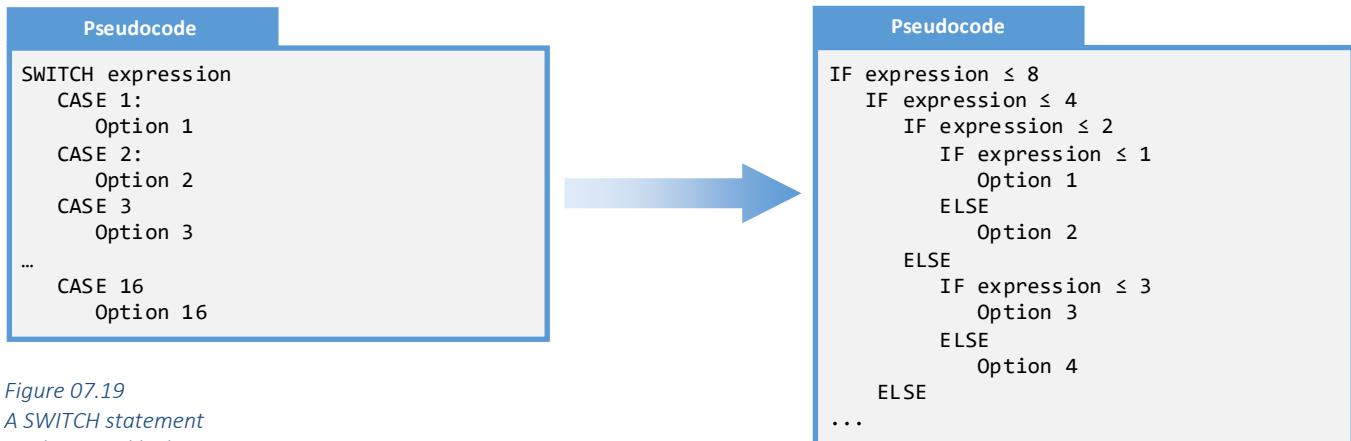


Figure 07.19
A `SWITCH` statement
implemented by binary
search

This algorithm is $O(\log n)$ and is much more efficient than the linear algorithms of Iteration and IF/ELSE-IF previously discussed. Sophisticated compilers often implement this algorithm when the number of CASE labels is greater than 7. Note that some languages do not provide binary search capability (such as all languages relying on the Java Virtual Machine). If you are relying on $O(\log n)$ performance for your SWITCH/CASE implementation, you should check the compiler output in the disassembly window of your debugger to ensure that it is what you expect.

Jump Table

The most efficient way to implement the SWITCH/CASE statement is to create a jump table. Here, the compiler moves all the code corresponding to the body of CASE labels to a separate location in memory. These locations are stored in an array of pointers. The compiler will then index into this table based on the value from the controlling expression.

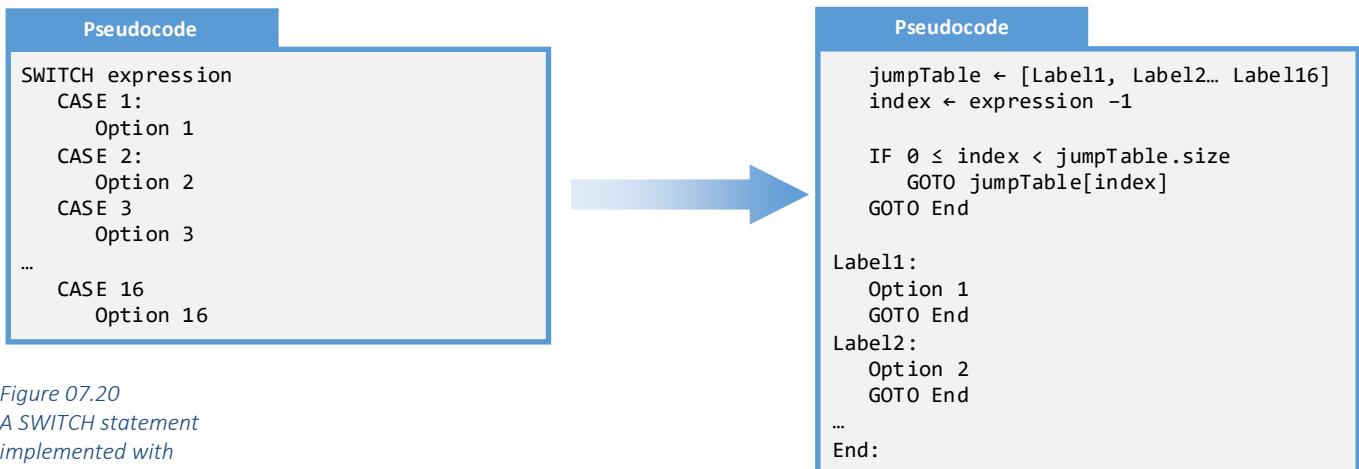
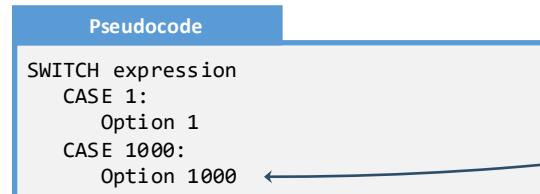


Figure 07.20
A SWITCH statement
implemented with
a jump table

This algorithm is O(1), taking exactly the same amount of time to execute if there is one CASE label as if there are a million. Furthermore, the DEFAULT or missing value case is not penalized as being slowest. Clearly, this is the best implementation in many cases. However, when there are large gaps of unused values in the jump table, this can consume a large amount of memory. Consider, for example, a simple program with two case labels: 1 and 1000. We would need to create a thousand-entry jump table in this case!

Figure 07.21
Code precluding
a jump table



Languages such as Java only utilize the jump table implementation when the biggest gap between case label values is 1 and when there is no more than one gap. They also cannot use jump tables when there are non-integral controlling expressions. In both cases, Java and related languages default to the Iteration implementation.

Best Practice 07.8 Use integral SWITCH controlling expressions with minimal gaps whenever possible

In order to maximize the chance that your code compiles to the jump table implementation, use integral controlling expressions whenever possible. It is also worthwhile to compact the resulting CASE labels so as to reduce or eliminate gaps in values.

Examples

Example 07.1: Expression Simplification

This example will demonstrate how to simplify a Boolean expression.

Problem

Simplify the Boolean expression in the following one-option IF statement:

VB

```
IF Not ((a > 5) And ((b <> 1) And (c <> True))) then  
    Value = true  
END IF
```

Solution

First we will turn the VB Boolean expression into mathematical notation, then we will reduce the Boolean expression to something simpler, and finally we will convert it back to VB notation.

Boolean Expression	Rationale
Not ((a > 5) And ((b <> 1) And (c <> True)))	
$\neg(a > 5) \wedge ((b \neq 1) \wedge (c \neq \text{true}))$	Convert to math notation
$\neg(a > 5) \wedge (\neg(b = 1) \wedge \neg(c = \text{true}))$	Definition of not equals
$\neg(\neg(a \leq 5) \wedge (b = 1 \vee c = \text{true}))$	De Morgan
$\neg\neg(a \leq 5) \vee (b = 1 \vee c = \text{true})$	De Morgan
$a \leq 5 \vee (b = 1 \vee c = \text{true})$	Double Negative
$a \leq 5 \vee (b = 1 \vee c)$	Simplification
$a \leq 5 \vee b = 1 \vee c$	Associative
$a \leq 5 \text{ Or } b = 1 \text{ Or } c$	Convert to VB notation

VB

```
IF a <= 5 Or b = 1 Or c Then  
    Value = true  
END IF
```

The final step is to validate our conversion to make sure we did not make a mistake.

	Not ((a > 5) And ((b <> 1) And (c <> True)))	a <= 5 Or b = 1 Or c
a = 6, b = 1, c = False	True	True
a = 6, b = 1, c = True	True	True
a = 6, b = 0, c = False	False	False
a = 6, b = 0, c = True	True	True
a = 4, b = 1, c = False	True	True
a = 4, b = 1, c = True	True	True
a = 4, b = 0, c = False	True	True
a = 4, b = 0, c = True	True	True

Example 07.2: Prioritize IF/ELSE-IF Clauses

This example will demonstrate how to optimize a Boolean expression.

Problem

Optimize the following function:

JavaScript

```
function isLeapYear(year) {  
    if (year % 400 == 0)  
        return true;                                //Quad - century  
    else if ((year % 100 == 0) && (year % 400 != 0))  
        return false;                               //Century  
    else if ((year % 4 == 0) && (year % 100 != 0))  
        return true;                                //Quad - year  
    else  
        return false;                                //Odd - year  
}
```

Solution

Notice the redundancy in the Boolean expressions (see Best Practice 07.1 for details).

JavaScript

```
function isLeapYear(year) {  
    if (year % 400 == 0)  
        return true;                                //Quad - century  
    else if (year % 100 == 0)  
        return false;                               //Century  
    else  
        return year % 4 == 0;                      //Quad - year  
                                            //and Odd year  
}
```

Next, we will put the most common case first: the odd years.

Case	Number of years in 400
Quad century	1
Century	3
Quad year	96
Odd year	300

JavaScript

```
function isLeapYear(year) {  
    if (year % 4 != 0)  
        return false;                                //Quad - year  
    else if (year % 100 != 0)  
        return true;                               //Leap year  
    else  
        return year % 400 == 0;                     //Century and  
                                                //Quad-Century  
}
```

Exercises

Exercise 07.1: Boolean Identities

For each of the following Boolean expressions, state the equivalent value and identify the associated Boolean Algebra identity.

Original Expression	Transformation	Boolean Algebra Identity
$\neg(B \vee A)$		
$A \wedge B$		
$A \wedge (B \wedge C)$		
$A \wedge \text{false}$		
$A \wedge \neg A$		
$A \vee (B \wedge C)$		
$A \vee \text{false}$		
$A = \text{false}$		
$A \wedge (A \vee B)$		
$(A \wedge B) \vee (A \wedge C)$		
$\neg B \vee \neg A$		
$A \wedge A$		
$A \vee \neg A$		

Exercise 07.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
All SWITCH/CASE statements compile to O(1)	
The CPU favors the FALSE option over the TRUE option	
It is considered sloppy to have a blank statement in an IF/ELSE-IF sequence	
With an IF/ELSE-IF sequence, put the most common option last	
You can use a truth table to verify your Boolean algebra transformations	

Exercise 07.3: Name the Identity

For the following Boolean transformation, name the associated Boolean Algebra identity.

Transformation	Identity Name
$\neg(\text{isHappy} \wedge \text{knowIt}) \Leftrightarrow \neg\text{isHappy} \vee \neg\text{knowIt}$	
$\neg\neg\text{isGraduated} \Leftrightarrow \text{isGraduated}$	
$(\text{income} > 400) = \text{true} \Leftrightarrow (\text{income} > 400)$	
$(\text{grade} \geq 90\%) \wedge \text{false} \Leftrightarrow \text{false}$	
$\text{score} > \text{threshold} \Leftrightarrow \neg(\text{score} \leq \text{threshold})$	
$\text{isFinished} \vee \text{false} \Leftrightarrow \text{isFinished}$	

Problems

Problem 07.1: computeTax Optimized

Consider the following implementation of a function that computes the tax burden for an individual based on their income:

JavaScript

```
function computeTax (income) {
    var tax;

    // 10% tax bracket
    if (income >= 0 && income < 15100) {
        tax = income * 0.10;
    }
    // 15% tax bracket
    else if (income >= 15100 && income < 61300) {
        tax = 1510 + 0.15 * (income - 15100);
    }
    // 25% tax bracket
    else if (income >= 61300 && income < 123700) {
        tax = 8440 + 0.25 * (income - 61300);
    }
    // 28% tax bracket
    else if (income >= 123700 && income < 188450) {
        tax = 24040 + 0.28 * (income - 123700);
    }
    // 33% tax bracket
    else if (income >= 188450 && income < 336550) {
        tax = 42170 + 0.33 * (income - 188450);
    }
    // 35% tax bracket
    else if (income >= 336550) {
        tax = 91043 + 0.35 * (income - 336550);
    }
    return tax;
}
```

Refactor this design taking into account the following statistics with regards to the number of taxpayers in each tax bracket. We wish this function to be as efficient as possible.

Bracket	Number of individuals in this tax bracket
0%	36,860,716
10%	27,400,094
15%	42,195,123
25%	24,009,141
28%	4,603,602
33%	1,768,562
35%	1,245,124

Problem 07.2: Level Completion

In the programming language of your choice, create the code to match the following description:

The user has 120 seconds to complete a level in a game. If that time expires, the level is terminated regardless of whether the end has been reached. The level is also terminated if the player hits a wall, the fuel is expended (a total of 50 units of fuel are provided on the onset of each level), or an obstacle has been hit. If the user has encountered an extra life prize, then a wall or obstacle collision will be forgiven and the prize is removed. The user can also encounter a time bonus prize, which will extend the level completion time by 20 seconds.

Problem 07.3: Is Leap Optimization

Consider the `isLeapYear()` function from Example 07.2.

JavaScript

```
function isLeapYear(year) {  
    if (year % 400 == 0) {  
        return true; //Quad-century  
    } else if (year % 100 == 0) {  
        return false; //Century  
    } else {  
        return year % 4 == 0; //Quad-year  
    } //and Odd year
```

Can you write this as a single Boolean expression? Though there are many solutions on the internet, see if you can do this without looking up the answer.

Problem 07.4: Busy Intersection

Consider the following problem:

Two roads meet at an intersection with a 4-way stop. A car pulls up to this intersection with no pedestrians. The car has two options: proceed into the intersection or wait. What should the driver do?

Research the traffic laws governing a 4-way stop. Create a flowchart representing a solution which will tell the driver whether he/she can proceed or whether he/she should wait. In the programming language of your choice, implement your solution.

Problem 07.5: SWITCH/CASE Optimization

Consider the following Kotlin code which generates a message for a student based on the grade earned. Note that the `when` statement in Kotlin works much like a SWITCH/CASE statement.

Kotlin

```
var message = when(grade) {  
    in 90..100          -> "Great Job!"  
    in 80.. 89          -> "Solid work"  
    in 60.. 79          -> "Perhaps more work is needed"  
    in  1.. 59          -> "Please try again"  
    0                 -> "Failed to submit"  
    !in 101..Int.MAX_VALUE -> "Unbelievable"  
    else               -> "Invalid"  
}
```

Which of the four options (IF/ELSE-IF, iteration, binary search, or jump table) can the Kotlin compiler utilize to implement this construct?

How can the code writer refactor this code to make it more likely that a faster implementation can be used?

Problem 07.6: McLaren

McLaren Automobiles has several cars in their lineup:

Model	Price
GT	\$213,500
570S	\$203,500
600LT	\$259,000
720S	\$299,000
765LT	\$375,000
Elva	\$1,700,000
650S	\$349,500
Senna	\$837,000
P1	\$1,150,000
Speedtail	\$2,100,000
MP4-12C	\$231,500
SLR	\$188,000
F1	\$15,620,000

What would be the most effective way to convert the car model (as text) into a price (an integer)?

Challenges

Challenge 07.1: Monopoly

Consider the Monopoly hotel-buying algorithm developed in Challenge 01.3. Convert that flowchart design to a working program, considering all the decision optimizations we have learned about this semester.

Challenge 07.2: Legacy Code

Find some code in a project you have previously worked in. In this code, do the following:

1. Find one instance of a decision structure that was not used optimally.
1. Come up with a better design.
2. Justify each change you made in the context of what you have learned in this chapter.

Challenge 07.3: Open Source

Find an open source codebase in a programming language of your choice. In this code, do the following:

1. Find one instance of a decision structure that was not used optimally.
2. Come up with a better design.
3. Justify each change you made in the context of what you have learned in this chapter.

Collections

Chapter 08

Collections, also known as aggregate data types, are groups of values contained under a single variable name.

Broadly speaking, collections are groups of values under a single variable name. Not only are they commonly used programming tools, there is also a wide variety of flavors of collections the programmer can choose between.

Aggregate type	Description
Structure	Custom data type consisting of a collection of named variables
Tuple	A fixed-size collection of unnamed variables, each referenced by position
Array	Homogeneous elements stored in a continuous region of memory facilitating random access
List	Homogeneous elements allowing for rapid insertion and removal while providing sequential access
Dictionary	Ordered collection of homogeneous pairs facilitating fast key-value insertion, removal, and retrieval

Structures, tuples, arrays, lists, and dictionaries are often called aggregates or aggregate data types. The term “aggregate” means “formed by combining several individual elements.” In many ways, this term is more suitable than collections because it describes what we are doing: building more complex data structures from existing ones.

Aggregates are collections of individual values under a single variable name used to represent a design concern

Many beginning programmers latch onto one or two collection types, using them to the exclusion of all others. The purpose of this chapter is to illustrate the different types of collections and describe how they can be effectively used to solve programming challenges.

Structure

A structure is a simple aggregate data type where a collection of variables can be contained in a single custom data type. This allows the programmer to create new data types specifically tailored for the needs of a given application. Each variable contained in a structure is called a member variable, and any combination of data types can be contained therein. Structure elements are reachable by name, not by index. This means it is impossible to iterate through all member variables in a structure. Note that there is no performance penalty to using a structure. For most languages and in most compilers, they are treated exactly the same as individual variables.

Structures are defined before they are used in the application

Figure 08.1
Structures defined in different languages

Structures need to be defined before they are used, making them early binding constructs. This means that the compiler has complete knowledge of the composition of the collection and how it is used; there is nothing left to be resolved at runtime.

C	Java
<pre>struct Person { char lastName[256]; char firstInitial; int age; };</pre>	<pre>class Person { public String lastName; public char firstInitial; public int age; };</pre>
Ruby	Swift
<pre>Person = Struct.new(:lastName, :firstInitial, :age)</pre>	<pre>struct Person { var lastName: String? var firstInitial: Char var age = 0 }</pre>

Best Practice 08.1 Structures should not contain large collections, collections that grow, or collections that shrink

A structure would not be suitable for storing a collection of pixels representing an image or a collection of transactions representing an account register. In both these cases, the number of member variables are large and may not be known at compile time. An array or a list would be a more suitable tool for that job. That being said, you may want to use a structure to represent a picture which contains three member variables: the name, the type, and an array of pixels.

Best Practice 08.2 Structures should map to design concerns

When variables are collected into a structure, make sure the structure should mean something in the problem domain. In other words, structures should not be a random collection of variables. The programmer should be able to readily explain the purpose of the structure to another programmer or even the project sponsor.

Best Practice 08.3 Name your structures carefully

In most programming languages, structures are defined before they are used and the definition requires a name. Choose your name carefully. As the adage goes, “if you cannot name something, you probably do not understand it.” Structure names should be nouns because they represent things. A good structure name should be precise, complete, and singular. You should not use conjunctions such as “and” or “or” in a structure name. This is an indication that the structure does not represent a single well-defined concept.

Best Practice 08.4 Only define a structure globally if it is to be used by the entire program

Recall that structures need to be defined before they are used. The visibility of the structure definition is called the scope. Generally, the scope of a programming construct should be related to the utility of the programming construct. If the entire application could benefit from the structure definition, then that definition should be available to the entire program. This is called global scope. On the other hand, if only a small part of the program could make use of the structure, limit the scope. The specifics of how to specify scope in an application depend on the programming language you are using. Several common tools to specify scope include blocks, functions, classes, namespaces, packages, and files.

Tuples

A tuple is an ordered set of values. Conceptually they are very similar to structures, but there are two key differences. First, tuples refer to member variables by position whereas structures refer to them by name. Second, tuples do not need to be predefined as structures do; you can simply start using a tuple without first defining a new data type. Nevertheless, both tuples and structures are early binding constructs. Thus, you can describe tuples as unnamed anonymous structure types with unnamed members. As with structures, access to individual member variables in a tuple is akin to accessing a standalone variable in most programming languages.

C#

```
var t1 = ("Helfrich", 'J', 47);
var t2 = (r: 6, c: 2);

lastName = t1.Item1;
row = t2.r;
```

C++

```
std::tuple<string, char, int> t;
t = make_tuple("Helfrich", 'J', 47);

lastName = t.get<0>(t);
```

Python

```
t = ('Helfrich', 'J', 47)

lastName = t[0]
```

Ruby

```
t = Tuple.new("Helfrich", 'J', 47)

lastName = t._1
```

Most languages reference member variables by index rather than by name

Figure 08.2
Different ways to implement tuples

There is considerable variation in implementation and use of tuples across various programming languages. It behooves the programmer to researching these implementation details before using them. In most languages, tuples are mutable. This means that member variables can be changed after the tuple is created. Python is an exception. In most languages, tuples do not have named member variables. C# and Swift are exceptions. In most languages, you cannot iterate through a tuple. Python is an exception. Finally, you cannot add or remove elements from a tuple.

Best Practice 08.5 Don't use a tuple when you could use a structure

As mentioned before, tuples are anonymous structures with unnamed member variables. The question is: why are your member variables unnamed? Why do you not name the structure? Naming things adds order and clarity to code. It encourages you to model your data with greater fidelity (more on that in Unit 2). In many cases, programmers use tuples because they are too lazy to create define a structure. This laziness often results in more time spent interpreting and debugging code. Tuples are useful for temporary groups of related values; if your variable going to be used beyond a single function or two closely related functions, use a structure instead.

Best Practice 08.6 Add a comment when referring to member variables within tuples

Because most languages do not allow the programmer to name or label a tuple member variable, it is almost always helpful to add a comment describing its use.

Best Practice 08.7 Use tuples to get more than one return value from a function

There is a strong analogy between the parameters passed to a function (an ordered set of anonymous values of different data types) and tuples. Note that most

programming languages only allow one return value from a function. You can address this asymmetry by returning a tuple from a function.

Array

As with tuples and structures, arrays are a “buckets of variables.” The difference is that every member of an array must be the same data type in most languages, and each individual element is referenced by index. In most implementations, arrays are a fixed size; once they are created, they cannot be grown to accommodate more elements. They also reside in a contiguous block of memory, so the n^{th} element resides immediately after element $n - 1$. There is a slight performance penalty for accessing an element from an array. The compiler needs to add the address of the first element to the index. This value is then dereferenced to retrieve the element. This is an $O(1)$ operation; it takes the same amount of time to access an element from a million-item array as it does from a two-item array. Adding and removing elements from the middle of an array is expensive, however. It is necessary to shift every element after the one added or removed. This is $O(n)!$

There are three flavors of arrays in C++

C++

```
int a1[] = {1, 2, 3};    // fixed length
std::array<int> a2;      // fixed length
std::vector<int, 3> a3; // can grow
a1[0] = -1;
```

Swift

```
let a1 = [1, 2, 3]
var a2: [Integer] = []
var a3: Array<Integer> = Array()
```

Java

```
int[] a;
a[0] = -1;
```

PHP

```
$a = array(1, 2, 3);
$a[0] = -1;
```

VB

```
Dim a(3) As Integer
a(0) = -1
```

JavaScript

```
var a = [1, 2, 3];
a[0] = -1;
```

Figure 08.3
Different ways to implement arrays

Some programming languages implement arrays as immutable, meaning they cannot be changed. Most do not. Some languages allow arrays to grow as more elements are added. Most do not. Finally, some languages even allow arrays to contain more than one data type. Again, most do not. You should investigate the specific implementation of arrays in your programming language of choice. There are two ways of looking at arrays in a program: a collection of values and table look-up.

Arrays as Collections

Arrays are frequently used to store collections of data. In almost all collection problems, two questions need to be answered:

1. How do you iterate through the collection? See Chapter 09 Strategy: Loops for more details on this.
2. What happens to each item in the collection?

In the typical collection problem, all or a fixed fraction of the elements in the array must be visited. These problems tend to be $O(n)$ because the number of elements visited is directly related to the size of the array.

Arrays for Table Lookup

Though the most common use for arrays is to store collections of elements, a powerful paradigm is to use them to look up values in a carefully organized table. To illustrate how this can be used, consider the following code displaying the name of a month from the month number:

The diagram shows a box containing Swift code. The code defines an array of month names and prints the month corresponding to a given month number. A handwritten note on the left says "No IF statement needed to select the month name". A handwritten note on the right says "The array of month names remains fixed".

```
Swift
var months = ["January", "February", "March", "April",
              "May", "June", "July", "August", "September",
              "October", "November", "December"]
print(months[monthNumber - 1])
```

The above code is much more efficient than a traditional IF/ELSE statement or even faster than a SWITCH/CASE statement. We are leveraging the fact that the month numbers are in the range of 1–12 with no gaps. Notice that we need to slightly transform the month number to be in the array range of 0–11. In most table-lookup scenarios, a little data translation is necessary. In the end, this is O(1), completely independent of the size of the dataset we are working with.

There are two steps in the table-lookup process:

1. Create a table with the data to be referenced
2. Write the code to extract the data from the table

Create the Table

The most important part of the table-lookup technique is to place the necessary data into a table in such a way that it can be conveniently extracted in the second step of the process. The trick here is to look for patterns. For example, say we wanted to determine the letter grade based on the number grade. There are 101 possible grades, making for a very large table! However, we will notice that a pattern exists: most of the letter grades have 10 corresponding number grades. This means we can get away with only 11 elements in the table, not 101.

The diagram shows a box containing C# code. It defines an array of characters representing letter grades ('F' through 'A') and includes a comment indicating the percentage scale (0% to 100%).

```
C#
//          0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
char[] grades = {'F', 'F', 'F', 'F', 'F', 'F', 'D', 'C', 'B', 'A', 'A'};
```

Extract the Data

The last part of the table lookup technique is to extract the data. This needs to be efficient, preferably O(1).

The diagram shows a box containing C# code. It uses Debug.Assert statements to validate the number grade and then extracts the corresponding letter grade from the array.

```
C#
Debug.Assert(numberGrade / 10 >= 0
Debug.Assert(numberGrade / 10 < grades.Length);
char grade = grades[numberGrade / 10];
```

Since we are accessing an item in an array, we need to check to make sure our computed index does not go outside the range of valid values. At very least, an assert should be used to validate this assumption.

Figure 08.4
Table lookup
using an array

Figure 08.5
Creating a table
for table lookup

Figure 08.6
Extracting data
for table lookup

Lists

A list is like an array in that it contains a collection of elements, each of the same data type. With most implementations, lists can grow to accommodate new elements. Another key difference between a list and array is that lists do not facilitate random-access; to reach the n^{th} element in a list requires first iterating through the first $n - 1$ elements. Lists are usually implemented with a linked list data structure: a collection of nodes where each node points to the next.

We can easily append to lists, and need to use iterators to loop through them

VB

```
Dim grades As New List(Of Float)  
  
grades.Add(3.9)  
grades.Add(3.1)  
  
For Each grade In grades  
    Console.WriteLine(grade)
```

C++

```
list <float> grades;  
  
grades.push_back(3.9);  
grades.push_back(3.1);  
  
for (float grade : grades)  
    cout << grade << endl;
```

C#

```
List<float> grades = new List<float>();  
  
grades.Add(3.9);  
grades.Add(3.1);  
  
foreach (float grade in grades)  
    Console.WriteLine(grade);
```

Java

```
List<float> grades;  
grade = new LinkedList<float>();  
  
grades.add(3.9);  
grades.add(3.1);  
  
for (float grade : grades)  
    System.out.println(grade);
```

Figure 08.7
Various implementations of lists

Notice that all these list examples require an iterator to access elements. You cannot index into a list. Therefore, lists are only effective as a collection mechanism, not as a table-lookup mechanism as arrays are. The choice between a list and an array might not always be obvious. Perhaps the following table will illustrate the differences.

Operation	Arrays	Lists
Create empty	All the space you will eventually need must be allocated when the array is created	Initially, no memory is used. We only consume memory as it is needed
Add element on end	If there is available space, this is fast. If the array is full, this is not possible	It takes a fixed and small amount of time to add an element on the end of a list
Add element on front	If there is space available, the entire array must be shifted. This takes $O(n)$ time	It takes a fixed and small amount of time to add an element on the front of a list
Add element to middle	If there is space available, all the elements after the inserted must be shifted. Takes $O(n)$ time	It takes a fixed and small amount of time to add an element on the middle of a list
Remove from end	This takes almost no time: $O(1)$	It takes a fixed and small amount of time to remove an element from the end of a list
Remove from front	The entire array must be shifted. This takes $O(n)$ time	It takes a fixed and small amount of time to remove an element from the front of a list
Remove from middle	Every element after the one removed must be shifted. This takes $O(n)$ time	It takes a fixed and small amount of time to remove an element from the middle of a list
Jump to the middle	This takes $O(1)$ time because we can index to any spot in an array	It is necessary to iterate through the list to find the element you need; no jumping is allowed: $O(n)$

Basically, if you need random access to elements in the middle of the collection, then the array is the best choice. If you need to periodically add or remove elements from the middle or beginning of the collection, then the list is the best choice.

Dictionaries

A dictionary, also known as an associative array, map, and symbol table, is a container that stores tuples: a key and a value. Elements in a dictionary are organized by the key, which can be just about any data type. Dictionaries are designed in such a way as to allow efficient insertion and removal while still retaining efficient lookup. Dictionaries are useful for table lookup.

```
C++  
map <string, int> score;  
scores["wall"] = 2;  
scores["rock"] = 4;  
  
cout << scores["wall"];
```

```
Swift  
var scores: [String: Integer] = [  
    "wall": 2,  
    "rock": 4]  
  
print(score["wall"])
```

```
JavaScript  
let scores = new Map();  
scores.set("wall", 2);  
scores.set("rock", 4);  
  
console.log(scores.get("wall"));
```

```
PHP  
$scores = array("wall"=>2, "rock"=>4);  
  
print $scores["wall"];
```

Figure 08.8
Various implementations
of dictionaries

Dictionaries are powerful tools that allow you to look up a value based on a key. In the above example, we are looking up the score corresponding to various objects in a game (a wall or a rock). Note that we do not need to reduce the key into an integer as we do with an array; just about any data type will work for a key.

Dictionaries can insert new elements, remove old elements, and lookup existing elements in $O(\log n)$ time. They accomplish this by using a data structure called a red-black balanced binary search tree. The details of how this works are unimportant right now; the only thing you need to know is that any key that supports the less-than operator can be used in a dictionary.

Examples

Example 08.1: Structure

This example will demonstrate how to create a structure to solve a problem.

Problem

Write a program to determine whether a user has access to a secret resource:

The secret code of “42” will be released only if the username is “Kane” and the password is “Rosebud”

Solution

Note that we could easily complete this task without using any type of collection. However, since the three components (the username, the password, and the code) are part of the same concept, it makes sense to collect them in a collection.

The best collection type here appears to be a structure. This is because each component has an obvious name and the sequence in which they appear in the collection does not matter.

Swift

```
struct Secret {
    var username: String?
    var password: String?
    var secretCode = 0
}

let secret = Secret(username: "Kane", password: "Rosebud",
                    secretCode: 42)

let username = readLine()
let password = readLine()

if username == secret.username && password == secret.password{
    println("The secret is: \(secret.secretCode)")
}
```

Example 08.2: Tuple

This example will demonstrate how to use tuples to solve a problem.

Problem

Write a program to represent a coordinate on a chessboard.

Solution

We could solve this problem with two variables (row and column), but each variable is part of a single concept. In other words, only having the row or only having the column would not be enough to perform any meaningful action.

We could also solve this with a structure, but the common convention of using two values is so strongly entrenched that we do not really gain anything by naming the member variables; the position is enough.

Python

```
# Set the position
coordinate = 5,6

# This will display "The current position is (5, 6)"
print("The current position is", coordinate)

# We can unpack a tuple into the individual components
row, column = coordinate
print("The row is", row)
print("The column is", column)
```

Example 08.3: Collection Array

This example will demonstrate how to represent an array of structures.

Problem

Write a program to determine whether a given user can access the system. The following are the list of allowed usernames and passwords:

UserName	Password
admin	123456
accounting	password
sam	ASDF
sue	computer
postmaster	trustno1
public	qwerty

Solution

Currently there are six elements in our list of users. It seems reasonable that there may be more added and that the passwords may change. This means we want to make it as easy as possible to alter the list without changing the program logic. Therefore, an array seems to be a much better option than hard-coding these values in a big IF statement.

Since there are two fields in every row of this array and we are not using this beyond the scope of our algorithm here, we can get away with a tuple for each row. Creating a structure seems too heavy for our application.

C#

```
Tuple <string, string>[] users =
{
    Tuple.Create("admin",      "123456"),
    Tuple.Create("accounting ", "password"),
    Tuple.Create("sam",        "ASDF"),
    Tuple.Create("sue",        "computer"),
    Tuple.Create("postmaster", "trustno1"),
    Tuple.Create("public",     "qwerty")
};

string username = Console.ReadLine();
string password = Console.ReadLine();

foreach (Tuple <string, string> user in users)
    if (username == user.Item1 && password == user.Item2)
        Console.WriteLine("You are authenticated");
```

Note how we iterate through this array of tuples using the instance method. We could have used indices, but the code would have been more complicated than necessary.

Example 08.4: Table Lookup Array

This example will demonstrate how to use a table-lookup array:

Problem

Write a function to compute an individual's tax based on their income:

Income range	Tax is
\$0 – \$15,100	10% of amount over \$0
\$15,100 – \$61,300	\$1,510 plus 15% of amount over \$15,100
\$61,300 – \$123,700	\$8,440 plus 25% of amount over \$61,300
\$123,700 – \$188,450	\$24,040 plus 28% of amount over \$123,700
\$188,450 – \$336,550	\$42,170 plus 28% of amount over \$188,450
\$336,550 – no limit	\$91,043 plus 35% of amount over \$336,550

Solution

Note that the tax brackets are presented in a tabular format. This is an indication that perhaps we can use the table-lookup method to solve this problem. Note that tax tables change every year so we want to make this as easy as possible to update this without introducing bugs.

The challenge is that the income ranges are not uniform. We therefore have to store the minimum and maximum amount of the range in the table and iterate through the array to find the appropriate tax bracket. We will use an array of structures to solve this problem.

```
C
float computeTax(float income)
{
    struct Bracket {
        float min;
        float max;
        float fixed;
        float rate;
    } brackets[] = {
        /*      min      max   fixed   rate */
        { 0,      15100,    0,  0.10},
        { 15100,   61300,  1510, 0.15},
        { 61300,  123700, 8440, 0.25},
        {123700,  188450, 24040, 0.28},
        {188450,  336500, 42170, 0.33},
        {336500, 99999999, 91043, 0.35}
    };

    for (int i = 0; i < sizeof(brackets)/sizeof(Bracket); i++)
        if (brackets[i].min <= income &&
            brackets[i].max >= income)
            return brackets[i].fixed +
                   brackets[i].rate * (income - brackets[i].min);
    return 0.0;
}
```

Example 08.5: List

This example will demonstrate how to use a list to solve a problem.

Problem

Write a function to advance the movement of the elements in a 3D game. Every time the `advance()` function is called, each element in the game will be advanced and the dead elements will be removed from the collection.

Solution

Since elements in the middle of the collection may be removed, it would be inefficient to use an array to store the game elements. Here, a list is a natural fit.

We will use an iterator to advance through the list of `Elements`. We cannot use an index or a pointer to traverse the `elements` list because C++ list objects only support iterators.

For each element in the list, we will advance them (calling the element's individual `advance()` function). Note that this may kill an element, especially if it impacted a wall or got hit by a bullet. In this case, the `erase()` function will be called which will remove it from the collection. The `erase()` function is $O(1)$ for lists and $O(n)$ for arrays.

C++

```
void Game :: advance()
{
    list<Elements *>::iterator it;
    for (it = elements.begin(); it != elements.end();)
    {
        // inertia advancement (plus spin for rocks)
        (*it)->advance();

        // kill those zombies
        if ((*it)->isDead())
            it = elements.erase(it);
        else
            it++;
    }
}
```

Example 08.6: Dictionary

This example will demonstrate how to use a dictionary to solve a problem.

Problem

Write a program to implement a simple phone book: this will look up a user's phone number based on their name.

Solution

This is a table lookup problem. If we could reduce the key into an integer value, then an array might be an option. However, since the key is the user's name, we will need to use a dictionary. Here, the key will be the user's name and the value will be the phone number.

Swift

```
var addressBook: [String: String] = [
    "Jenny":          "867-5309",
    "Almighty":       "776-2323",
    "Scott":          "(212) 664-7665",
    "Etta":           "842-3089",
    "Alicia":         "(347) 489-4608",
    "Steve":          "888-8888",
    "Mike":           "(330) 281-8004"]

let name = readLine()
print(addressBook[name])
```

Notice that we do not need to iterate through the list. The square-bracket array-like notation suggests that the value will be returned instantaneously (in $O(1)$). However, because this is a dictionary, it will be done in $O(\log n)$. To get an idea of how fast this is, if there were 220,000,000 phones in America, then it will take only 28 iterations of the loop to find a given phone number.

Exercises

Exercise 08.1: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Arrays have member variables	
Tuples are anonymous structures	
You can iterate through a structure in most languages	
Lists are designed to grow	
Dictionaries consist of key/value pairs	

Exercise 08.2: When Size is Known

For each of the following, specify whether the whose size is known at compile time or at runtime. Justify your response.

Collection Type	Compile time or Runtime
Structure	
Tuple	
Array	
List	
Dictionary	

Exercise 08.3: Access Time

How long does it take to retrieve a specific item somewhere in the middle of a collection? For each collection type, specify the algorithmic efficiency where n is the number of elements in the collection.

Collection Type	Algorithmic Efficiency of Access
Structure	
Tuple	
Array	
List	
Dictionary	

Exercise 08.4: Removal Time

How long does it take to remove a specific item somewhere in the middle of a collection? For each collection type, specify the algorithmic efficiency (where n is the number of elements in the collection) or indicate that it is not possible.

Collection Type	Algorithmic Efficiency of Removal
Structure	
Tuple	
Array	
List	
Dictionary	

Problems

Problem 08.1: Evens and Odds

In the pseudocode or a programming language of your choice, write the code to read a hundred values from a file. Then, determine how many evens were in the collection and how many odds.

Pseudocode

```
OPEN file
FOR i ← 0 ... 99
    READ values[i] from file

numEvenOdd[2] ← 0, 0

... your code goes here ...

PRINT There are numEvenOdd[0] evens
PRINT There are numEvenOdd[1] odds
```

Hint: This can be accomplished with a FOR loop and an IF statement. As a challenge, can you complete this task without an IF statement or any other Boolean expression?

Problem 08.2: Race Points

Professional cyclists determine their world standing based on a point system. For a large race, like the Paris Roubaix, the point breakdown is the following:

Position	Points
1	500
2	400
3	325
4	275
5	225
6	175
7	150
8	125
9	100
10	85
11	70
12	60
16	50
17	40

In the language of your choice, write the code necessary to compute how many points will be awarded to a rider if they come in at a certain given place. Note that the point scheme goes all the way down to the 60th place. Can you find all the values?

Problem 08.3: Extremes

We have a file containing a collection of numbers representing stock data. We wish to know the highest and the lowest value in the file. Write a program to open the file, read each element, and report the highest and the lowest values.

Problem 08.4: Word Count

Write a program to read a collection of words from a file. The program will tabulate the number of instances of each word. Then prompt the user for a word and display the number of instances of that word in the file.

Problem 08.5: Compute Balance

Write a program to determine how much money is in an individual's bank account. The program will start by assuming the account balance is at \$0.00. The program will then read all the transactions into a list. After the transactions are loaded, the program will sum the transactions to determine the balance. This balance will be displayed to the user.

When this is done, the user will realize that the balance is in error. Prompt the user for a transaction number where 1 is the first transaction in the register. The program will then remove that transaction from the register.

Challenges

Challenge 08.1: Fibonacci

The value for a given number in the Fibonacci sequence can be computed by adding the previous two values in the sequence:

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n - 1) + F(n - 2) & \text{if } n > 1 \end{cases}$$

This can be performed recursively as we learned in Chapter 03 Metric: Efficiency. It is also extremely slow.

Ruby

```
def fibonacci(n)
    n <= 1 ? n : fibonacci( n - 1 ) + fibonacci( n - 2 )
end
```

Write the code in the programming language of your choice which computes the first 50 numbers in the Fibonacci sequence. This should be done in $O(n)$ time where n is the size of the sequence you are computing.

Hint: Use an array to store the values.

Challenge 08.2: Find Prime Numbers

To find all the prime numbers below a certain value N , we will start with an array containing N elements. Each element in the array will correspond to a number. We will start by ruling out all the multiples of two greater than 2. Thus, we will "cross out" the values 4, 6, 8, 10, 12, ... N . We know these values are not prime because they have 2 as a factor (as well as 1 and the number itself).

We will then rule out all the multiples of three greater than 3. Thus, we will "cross out" the values 6, 9, 12, 15, 18, ... N . Note that 6, 12, 18 were already crossed out. This shouldn't matter; crossing out a value twice keeps it crossed out.

Next, we will rule out all the multiples of four. Notice that 4 is already crossed out! We did that when we went through all the multiples of 2. Thus, we can be assured that 4 is not a prime (its factors are 1, 2, and 4).

Next, we will rule out the multiples of five. Since 5 is not crossed out already, we will iterate through all the multiples of five greater than 5: 10, 15, 20, 25, ... N .

This process continues until we attempt to rule out the square root of N . We can stop here because everything above the square root of N is already crossed out. Why is that? See if you can figure it out.

Challenge 08.3: Reverse an Array

Sam is writing a program to manipulate stock market data. The file he is working with has the prices of a certain stock: one entry per day for several years. There is just one problem: the stocks are in the wrong order. In the file, the most recent price is first and the oldest price is last. His program needs to have the stocks in the opposite order.

Write the code to reverse the elements in an array. The program will:

1. Read a list of numbers from a file.
2. Display the contents in order.
3. Reverse the elements so the last is first and so on.
4. Display the contents reversed.

Challenge 08.4: Largest Sum

James is an avid cyclist. He has a power meter on his bicycle which measures how much power he is applying to his pedals. This power meter makes 10 measurements a second. When finished, the power meter creates a file with all the power measurements. Since the typical ride is 90 minutes in length, there are about 54,000 individual power measurements in a file. James wants to know what his best one-minute power was: the highest average power over any one-minute interval.

Write a program to compute the highest average for a given range of values (say one-minute). The program will:

1. Read the numbers from a file.
2. Identify the subarray with the largest sum.
3. Compute the average of that subarray.
4. Display the results.

Chapter 09 Loops

Loops are mechanisms allowing for repeated execution of a single block of code. Most code in a typical application is contained within at least one loop.

Loops, otherwise known as repetitive control structures, are defined as mechanism allowing for the repeated execution of one or more statements. Though they only account for a small percentage of statements, they have an outsized influence on the behavior and performance of a program. Most code in a typical application is contained within at least one loop.

Most code is contained within one or more loops

Loops are easy to spot in flowcharts because execution leaves a decision diamond and circles back to a previous point in the algorithm. Every flowchart loop must have that decision diamond and the circle back arrow.

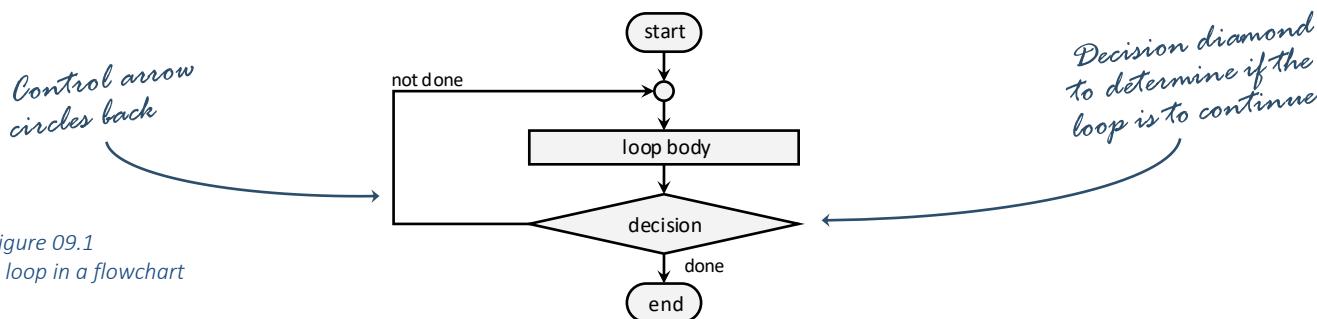


Figure 09.1
A loop in a flowchart

Programming languages use a variety of looping mechanisms: FOR, WHILE, DO-WHILE, and FOREACH. While the syntax for each is distinct and their recommended usage may vary, they are all equivalent: any FOR loop can be converted to a WHILE loop and so on.

Pseudocode

```
FOR count ← 1 ... 10
    PUT count
```

Pseudocode

```
count ← 1
WHILE count ≤ 10
    PUT count
    COUNT ← count + 1
```

Figure 09.2
Two loops performing
the same action

From a design standpoint, there are four flavors of loops:

Type	Description
Event-controlled	Repeat until an event occurs
Counter-controlled	Repeat a fixed number of times
Sentinel-controlled	Repeat until a gatekeeper indicates the loop is finished
Collection	Iterate through all the elements in a collection

Event-Controlled

An event-controlled loop is a loop that continues until a given event occurs. The number of repetitions is typically not known before the program starts; the controlling event could occur on the first iteration of the loop or after a million iterations. In almost all cases, an event-controlled loop is implemented with a WHILE or a DO-WHILE statement.

For example, a program may prompt the user for her age. If the user entered a negative number or anything else that is not within the acceptable age range for a human, then the program would prompt the user again. In this case, an event-controlled loop is probably the right tool for the job.

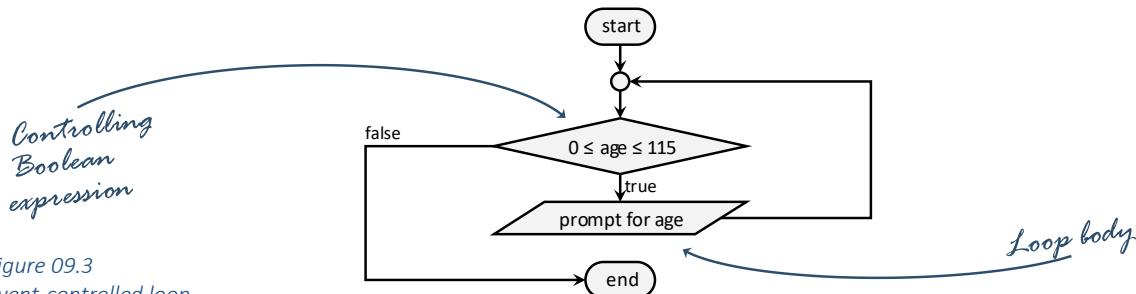


Figure 09.3
Event-controlled loop
in a flowchart

In the above example, notice that the decision either sends control back up to the prompt or continues on. This same algorithm can be expressed in pseudocode.

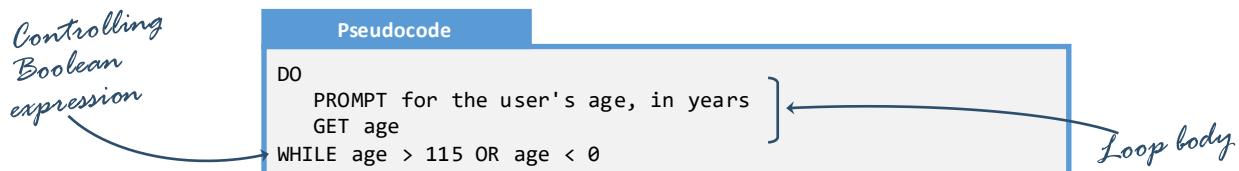


Figure 09.4
Two parts of an event-
controlled loop

An event-controlled loop consists of two parts: the controlling expression and the loop body. Every event-controlled loop needs to address these two parts.

Controlling Expression

All event-controlled loops are governed by a Boolean expression. In every programming language, a TRUE condition indicates that the loop is to continue and FALSE indicates that the loop is finished.

```
Java
int age = 0;
Scanner input = new Scanner(System.in);

do {
    // prompt the user for the user's age
    System.out.println("What is your age in years? ");
    age = input.nextInt();
}

while (age > 115 || age < 0);
```

Figure 09.5
Controlling Boolean
expression

When designing an event-controlled loop, the first question to be answered is “How do you know when the loop is done?” It is critical to get a clear answer to this question early in the design process.

Best Practice 09.1 Simplify the loop controlling expression as much as possible

As with the controlling Boolean expression with decisions, the expression should be simplified to make it efficient to execute and easy to understand. Many developers work to reduce and simplify the Boolean expressions associated with IF statements but neglect to do this with their loop expressions. This is backwards! The controlling Boolean expression is evaluated on every iteration of the loop, meaning that efficiency of execution is much more important.

It is not uncommon to have a function in the controlling expression. A function that returns a Boolean is called a predicate.

Best Practice 09.2 Predicates called from loop controlling expressions should be extremely efficient

While performance is very important in all aspects of software design, it is critical in predicates called in the controlling expression of a loop. Every effort should be taken to streamline their evaluation.

Loop Body

Most event-controlled loops have a loop body containing the code that will be executed every iteration. There are no limits or guidelines how much code this may contain; it is common for a loop body to be nearly empty or contain millions of lines of code.

```
Java
int age = 0;
Scanner input = new Scanner(System.in);

do {
    // prompt the user for the user's age
    System.out.println("What is your age in years? ");
    age = input.nextInt();
}
while (age > 115 || age < 0);
```

Figure 09.6
Loop body

Because an event-controlled loop will continue until the controlling Boolean expression evaluates to FALSE, most loop bodies provide code to alter the state of the controlling expression. This could be a prompt that updates a value checked in the controlling expression or it could be as simple as a pause.

When designing an event-controlled loop, the second question to be answered is “What *does* change or *could* change every iteration?” If this question is difficult to answer, then an event-controlled loop might not be the right tool for the job.

Best Practice 09.3 Avoid unnecessary code in a loop body

Because a loop body is executed multiple times, performance is critical. A common mistake for new programmers to make is to put unnecessary code in a loop body. If code needs to be executed only once, then it should be placed outside the loop. When a loop design is finished, it is worth taking a second pass through the design to make sure nothing unnecessary is in the body.

Counter-Controlled

A counter-controlled loop is a repetitive control structure that executes a given block of code a fixed number of times. Typically, the number of iterations is known before the loop begins execution. For example, a program may need to sum the numbers below a certain threshold. Since the number of items is known before execution begins, a counter-controlled loop is probably the right tool for the job.

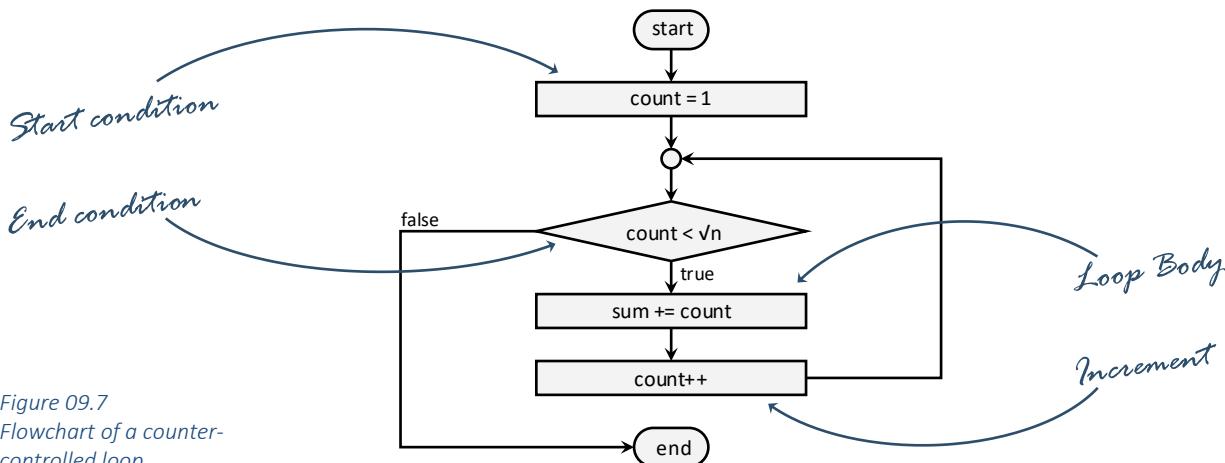


Figure 09.7
Flowchart of a counter-controlled loop

In the above example, notice that two statements are executed many times (where the `sum` variable is updated and where `count` is incremented). This is the loop body, much like an event-controlled loop. The big difference is that the controlling expression is governed by a single variable: the counter. Most counter-controlled loops have an integer variable which dictates whether the loop will continue.



Figure 09.8
The four parts of a counter-controlled loop

With all counter-controlled loops, there are four parts: where to start, where to end, how to iterate, and what happens with every iteration. Most languages have a special construct for counter-controlled loops: a FOR loop. These loops usually have fields for each of these four parts.

Start-Condition

All counter-controlled loops have a starting condition. This usually involves setting the counter variable to some initial value. Some languages may mask this step because variables are automatically declared or initialized. Language features such as this do not mean the step is unnecessary! They just mean it is implied.

Best Practice 09.4 Make starting conditions of a counter-controlled loop explicit

If a starting condition is set in a location of the code removed from the loop itself, add a comment indicating this.

Note that the starting condition of a loop is the only part that is executed a single time. Performance is therefore not nearly as critical here as in the other parts of the loop.

Best Practice 09.5 Move complicated code from the end-condition to the start-condition when possible

If a program needs to count from X to Y , it might be that computing X is easy whereas computing Y is difficult. Since the start-condition is executed only once and the end-condition is computed many times, it is much more efficient to count down from Y to X than it is to count up from X to Y .

The start condition sets the initial value of the counter

```
VB
For count As Integer = 1 to Math.Sqrt(n) Step 1
    sum += count
Next
```

Figure 09.9
Start-condition of a counter-controlled loop

In the above example, notice that the end-condition is very expensive. Rather than computing the square root in the end-condition, we can move it to the start-condition which is executed only once:

Place the complex or expensive operations in the start condition

```
VB
For count As Integer = Math.Sqrt(n) to 1 Step -1
    sum += count
Next
```

Figure 09.10
Simplifying the end-condition at the expense of the start-condition

End-Condition

The end-condition is the same as the controlling Boolean expression of an event-controlled loop. Here, we keep looping until the end-condition is met: when the controlling Boolean expression evaluates to FALSE.

```
VB
For count As Integer = Math.Sqrt(n) to 1 Step -1
    sum += count
Next
```

The end condition is just a number in VB

```
C#
for (int count = Math.Sqrt(n); count >= 1; count--)
    sum += count;
```

C# uses a Boolean expression for the end condition

Figure 09.11
End-condition of a counter-controlled loop

Note that if the loop were to execute N times, then the end-condition Boolean expression will be executed exactly $N+1$ times. There will be N evaluations of TRUE and a single FALSE execution.

Unlike event-controlled loops, counter-controlled loops almost always have a single counter variable in the controlling expression. This Boolean expression is typically a less-than operator or a not-equals operator.

Increment

The increment part of a counter-controlled loop modifies the counter. Hopefully, every iteration of the loop will move the counter closer to the end-condition. In most loops, the increment part adds or subtracts one to the counter.

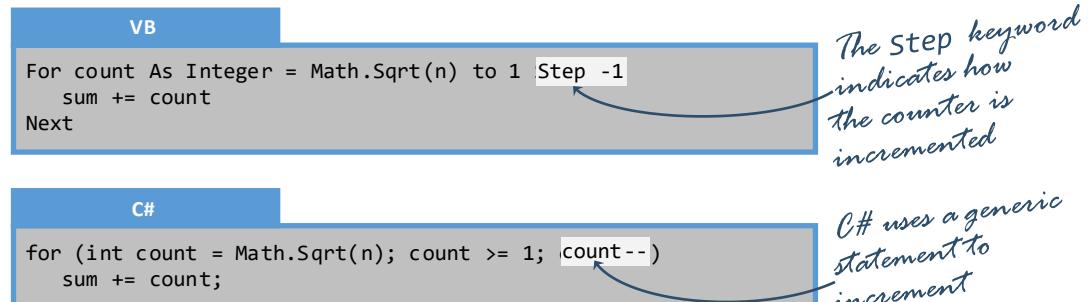


Figure 09.12
Increment of a counter-controlled loop

Best Practice 09.6 Be wary of a counter-controlled loop without an increment statement

If a counter-controlled loop has no increment statement, it usually means that it is actually an event-controlled loop in disguise. In cases such as these, do not use a FOR loop, but a WHILE loop instead.

Best Practice 09.7 Counter-controlled loops should have the increment statement in the FOR loop

Some programmers find it more convenient to put the increment statement in the loop body rather than directly in the FOR loop. This works, of course. However, it is also unnecessarily difficult to read. Most programmers expect to find the start-condition, end-condition, and increment code all in the loop statement.

Having the increment code in the loop body also increases the chances of an infinite loop. What happens when the increment code is skipped? In cases such as these, the counter never advances and the loop continues forever.

Best Practice 09.8 Each loop should have exactly one increment statement

Some programmers find it necessary to have an increment statement in the FOR loop and another one in the loop body. This is usually done to skip-over certain counter values. In general, this is a bad practice. It makes it difficult to understand the loop design and makes it more difficult to find bugs. Some languages such as Python forbid this practice: any modification to a counter variable in the loop body is ignored!

Loop Body

The loop body of a counter-controlled loop is the same as that of an event-controlled loop. It is executed exactly once per iteration, the controlling Boolean expression is guaranteed to be TRUE in the body of the loop, and it should be devoid of any statements that do not need to be there.

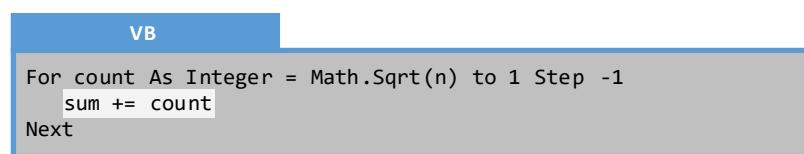


Figure 09.13
Loop body of a counter-controlled loop

Sentinel-Controlled

A sentinel is a guardian or a gatekeeper. The sentinel in a castle determines who enters and who is denied access. This analogy is fitting for sentinel-controlled loops, repetitive control structures where the controlling Boolean expression is a single variable called a sentinel. Usually, several conditions can indicate to the sentinel that the loop is finished. It is the job of the sentinel to collect these values.

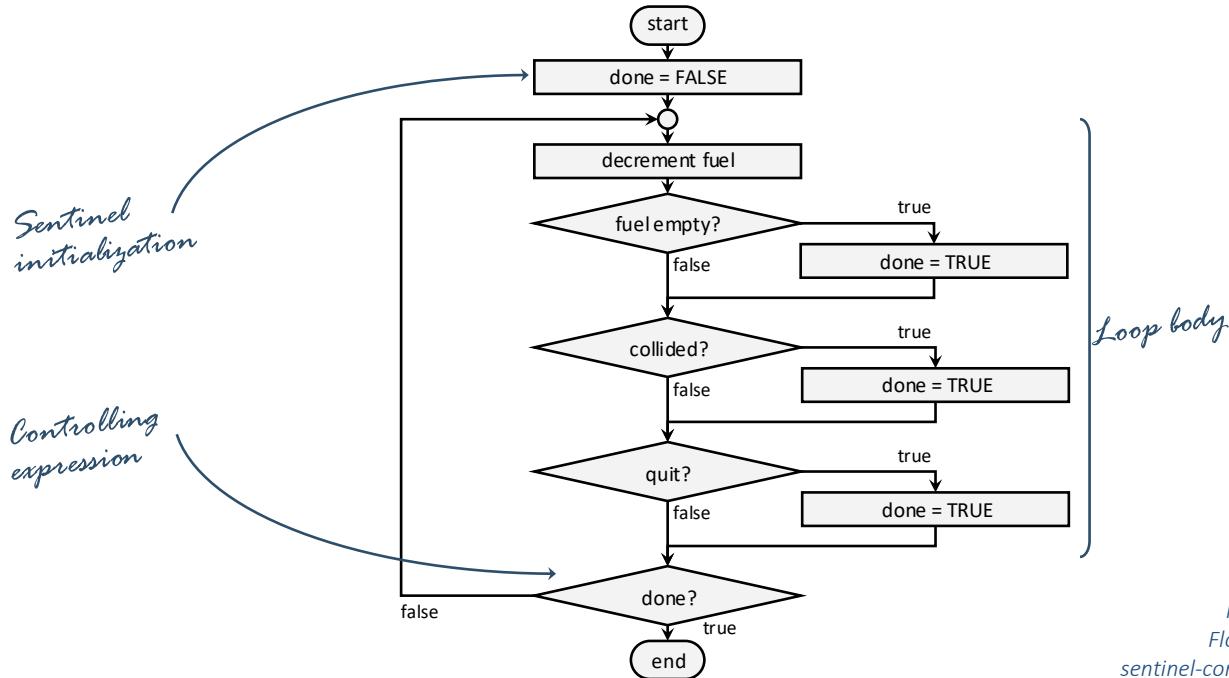


Figure 09.14
Flowchart of a sentinel-controlled loop

In the above example, there is a loop whose body encompasses many individual statements. Note that the controlling expression is based on a single sentinel variable: `done`. However, several events can cause the variable to be set to TRUE. These include running out of time, hitting a wall, or the user giving up. Whenever one of these conditions is met, the sentinel is set to TRUE and the loop exits.

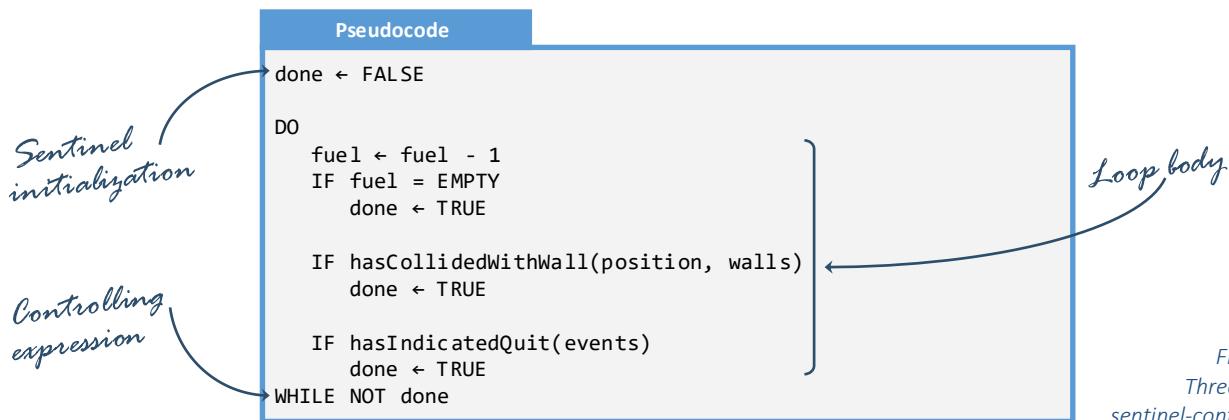


Figure 09.15
Three parts of a sentinel-controlled loop

There are three parts to a sentinel-controlled loop: the sentinel initialization, the controlling expression, and the loop body.

Sentinel Initialization

One of the most common mistakes in a sentinel-controlled loop is to fail to initialize the sentinel variable. The sentinel should be initialized with a value that indicates the loop should be continued.

```
C++
bool done = false;

do {
    // consume fuel
    fuel--;
    if (fuel == 0)
        done = true;

    // check for hitting a wall
    if (hasCollidedWithWall(position, walls))
        done = true;

    // check if the user has quit the game
    if (hasIndicatedQuit(events))
        done = true;
} while (!done);
```

Figure 09.16
Initialization a sentinel-controlled loop

Notice that the variable is declared and initialized immediately before the loop, making it easier for others to understand how the loop works.

Controlling Expression

The big advantage of sentinel-controlled loops is that the controlling expression is so simple. The logic controlling the loop is distributed in many locations throughout the body of the loop.

```
C++
bool done = false;

do {
    // consume fuel
    fuel--;
    if (fuel == 0)
        done = true;

    // check for hitting a wall
    if (hasCollidedWithWall(position, walls))
        done = true;

    // check if the user has quit the game
    if (hasIndicatedQuit(events))
        done = true;
} while (!done);
```

FALSE means continue

Post-condition

Figure 09.17
Controlling expression of a sentinel-controlled loop

There are two decisions that need to be made when designing the controlling expression: use a pre-condition or post-condition, and use a TRUE or FALSE loop.

Pre-Condition vs. Post-Condition

A pre-condition loop verifies the controlling expression before the loop is entered and the post-condition loop does so after. With a pre-condition loop, it is possible for the controlling Boolean expression to evaluate to FALSE on the first iteration, meaning that the body of the loop is never entered. In other words, the loop body can be executed 0 to N times. This is not true with post-condition loops. In this case, control always enters the body of the loop first. Only after the body of the loop is finished is the controlling expression evaluated. Thus, the loop body can be executed 1 to N times.

Figure 09.18
Pre- and post-condition loops

The figure shows two side-by-side code snippets in C++.

Left (Pre-condition loop):

```
C++
bool done = false;
while (!done) {
    ...
}
```

Right (Post-condition loop):

```
C++
bool done = false;
do {
    ...
} while (!done);
```

Usually pre-condition loops are called WHILE loops whereas post-condition are called DO-WHILE loops. Not all programming languages offer DO-WHILE loops.

Best Practice 09.9 Use a post-condition for a sentinel-controlled loop if that option is available

Because the loop-control logic for a sentinel-controlled loop resides inside the body of the loop, we always need to enter the body of the loop at least once. This means that a post-condition loop is better for sentinel-controlled loops.

True- and False- Controlled Loop

The controlling expression must evaluate to TRUE for the loop to continue and FALSE for the loop to exit. This is a feature of the programming language, beyond the control of the programmer. However, whether the sentinel uses TRUE or FALSE to continue the loop is a programmer decision.

Figure 09.19
True- and false-controlled sentinels

The figure shows two side-by-side code snippets in C++.

Left (True-controlled sentinel):

```
C++
bool done = false;
do {
    ...
} while (!done);
```

Right (False-controlled sentinel):

```
C++
bool notDone = true;
do {
    ...
} while (notDone);
```

In the above example on the left, the variable `done` is the sentinel. If it evaluates to TRUE, then the loop exists. The “WHILE !done” part of the loop is read “while not done,” so it is easily understood. In the example on the right, the variable `notDone` is the sentinel. If it evaluates to FALSE, then the loop exists. The “WHILE notDone” part of the loop also reads “while not done,” so it is equally understandable.

Best Practice 09.10 Consistently use true- or false-controlled sentinels in your codebase

Whether your sentinel is true-controlled or false-controlled is up to you. It is usually a good idea to use the same convention throughout so the code is more easily understood.

Loop Body

As with an event-controlled and counter-controlled loop, the loop body consists of all the code to be executed multiple times.

Java

```
bool done = false;

do {
    // consume fuel
    fuel--;
    if (fuel == 0)
        done = true;

    // check for hitting a wall
    if (hasCollidedWithWall(position, walls))
        done = true;

    // check if the user has quit the game
    if (hasIndicatedQuit(events))
        done = true;
} while (!done);
```

Figure 09.20
Loop body of a sentinel-controlled loop

There is one important difference with the other loop types: the logic for setting the sentinel is present in the loop body.

C++

```
bool done = false;

do {
    // consume fuel
    fuel--;
    if (fuel == 0)
        done = true;

    // check for hitting a wall
    if (hasCollidedWithWall(position, walls))
        done = true;

    // check if the user has quit the game
    if (hasIndicatedQuit(events))
        done = true;
} while (!done);
```

Figure 09.21
Loop logic distributed in the body of the loop

Notice that the loop control logic is spread throughout the loop. This can be an advantage: complex program logic does not need to be crammed into a single overly-complex Boolean expression. It can also be a disadvantage: the loop control logic can be difficult to find.

Best Practice 09.11 Make the loop control logic in a sentinel-controlled loop obvious

Take care to make the loop control logic obvious in your program. At a minimum, use prominent comments and a very consistent way of setting the sentinel variable so readers of the code can understand how the loop functions.

Collection-Controlled

A special flavor of the counter-controlled loop is called the collection loop. Here the purpose is to visit each member of a collection in sequence. This works with arrays, lists, and other data structures. We typically cannot iterate through collections containing named members such as structures or tuples.

Best Practice 09.12 Memorize all the collection loops in each programming language you use

Collection-controlled loops are such a common part of programming that we should not have to think of how they work. Each collection loop should be memorized in your programming language of choice, so they are used consistently in your code and you never make a mistake using them.

The nature of a collection loop is tightly coupled with the specifics of the collection. The main types of collection loops are indices, pointers, linked lists, iterators, and references.

Index Loops

Arrays can be traversed by index by starting at the first element (usually 0 but some languages start at 1) and ending at the last (usually `length - 1`). Most languages allow you to query the array itself to know its length as is the case this JavaScript example (Figure 09.22). Other languages, such as C, requires the programmer to maintain the length of the array as a separate variable.

```
JavaScript
var score = [13, 72, 83, 56, 35];
var index;
var sum = 0;

for (i = 0; i < score.length; i++)
    sum += score[i];
```

Figure 09.22
Looping through an array using indices

Index collection loops are closely related to counter-controlled loops. Three of the four parts of a counter-controlled loop are almost always the same with index loops:

Loop part	Description
Start-condition	<code>i = 0</code>
End-condition	<code>i < size</code>
Increment	<code>i++</code>
Loop body	Depends on the application

Index iteration is often the slowest way to traverse an array because an arithmetic operation and a dereference need to be conducted to fetch an element from an array using an index. This performance penalty is magnified because it needs to be done n times, though the overall algorithmic efficiency remains $O(n)$ for the entire operation.

Pointer Loops

In languages that have pointers, there is a faster way to traverse an array than with indices. This leverages the fact that arrays are contiguous elements in memory; if you know the address of the first element, you can deduce the elements of the rest as they are a fixed offset from the beginning.

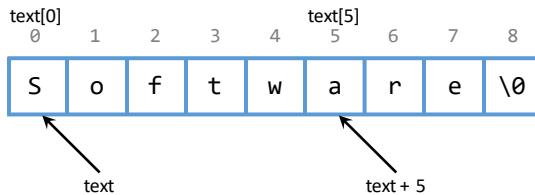


Figure 09.23
Finding the address of elements in an array

In the above example, the pointer `text` refers to the first element in the string. In this case, it is the letter 'S'. If we were to use the array notation, we can retrieve this letter with `text[0]`. The sixth letter is the letter 'a'. We can point to this letter with `text + 5` since it is five elements to the right of the first letter.

Programming languages such as C and C++ allow us to declare a variable that is a pointer. This variable does not contain data, but rather an address. We can retrieve the address of a variable with the address-of operator and retrieve the data that a pointer refers to with the dereference operator.

```
C
// A pointer variable is declared with '*'
int * pointer;

// The address-of operator is the '&'
pointer = & number;

// The dereference operator is the '>'
assert(*pointer == number);
```

Figure 09.24
Declaring and using a pointer

Perhaps the most common example of pointer traversal involves C strings where the last element in the string is the null character. Since the null character maps to the value 0 which is `false`, you keep iterating until the false value is encountered. This method is very efficient because each iteration only requires a dereference. In fact, only four assembly instructions are required for each iteration of the loop.

```
C
char text[] = "software engineering";
for (char *p = text; *p; p++)
    *p = toupper(*p);
```

Figure 09.25
Looping through an array using pointers

Pointer collection loops are closely related to counter-controlled loops. Three of the four parts of a counter-controlled loop are the same with pointer loops:

Loop part	Description
Start-condition	<code>p = text</code>
End-condition	<code>*p</code>
Increment	<code>p++</code>
Loop body	Depends on the application

Linked List Loops

A linked list is a special type of container where every element in the list is connected to the next through pointers. Each element in the linked list is called a node: a structure containing a data element and a pointer to the next item in the sequence.

```
C++  
struct Node {  
    int data;           // the data contained in each element  
    Node * pNext;      // a pointer to the next item in  
};                   // the collection
```

Figure 09.26
A single linked list node

When a collection of nodes point to each other, we have a linked list. The following linked list has the values 2, 4, 6, 8 in sequence.

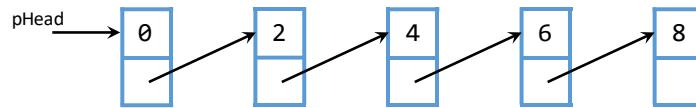


Figure 09.27
A linked list

To traverse a linked list, a pointer to a node is needed. The pointer advances through the linked list by taking on the value of pNext.

```
C++  
for (Node * p = pHead;           // start at the head  
     p;                         // continue until the null address  
     p = p->pNext)              // advance to the next node  
     cout << p->data << endl;   // access each element
```

Figure 09.28
Looping through
a linked list

This loop assumes that the end of the linked list is demarcated with the **NULL** address. This address has the **0x00000000** value, corresponding to FALSE. As with index and pointer loops, linked list loops are closely related to counter-controlled loops. Three of the four parts of a counter-controlled loop are always the same with linked list loops:

Loop part	Description
Start-condition	p = pHead
End-condition	p
Increment	p = p->pNext
Loop body	Depends on the application

Best Practice 09.13 When working with linked lists, be wary of infinite loops

The other types of container loops offer the assurance that each increment will bring us closer to the end-condition. This assurance is not present with linked lists. Because nodes can physically reside in any location in memory, there is no guarantee that the second node is after the third in memory; it could be before it! In fact, an ill-formed linked list could contain a circuit. These are hard to detect and fix.

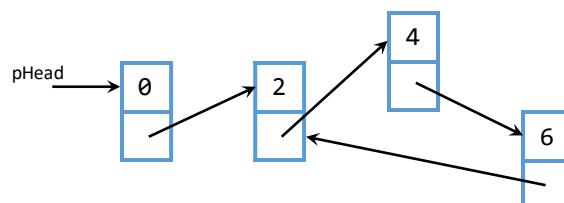


Figure 09.29
A linked list
containing a circuit

Iterator Loops

Iterators are generic looping structures that work the same regardless of the underlying data structure. Though binary trees, linked lists, arrays, and strings may be implemented differently, the programmer can use the same iterator loop to traverse each of these data structures. Iterators are designed to be efficient; most implementations are as efficient as the pointer loop traverse.

Kotlin

```
val scores: IntArray
    intArrayOf(13, 72, 83, 56, 35)

Iterator it = scores.iterator()
while (it.hasNext())
    sum += it.nextInt()
```

Figure 09.30
Looping through an array using iterators

Notice that the iterator is its own data type: an `Iterator`. It also uses a standard start-condition (`iterator()`), end-condition (`hasNext()`), and increment statement (`nextInt()`). Though the syntax is quite different, the same components exist in other languages:

C++

```
vector <int> :: iterator it;

for (it = scores.begin(); it != scores.end(); it++)
    sum += *it;
```

Java

```
Iterator it;

for (it = scores.iterator(); it.hasNext(); )
    sum += it.nextInt();
```

Figure 09.31
Other implementations of iterators

Best Practice 09:14 If your programming language has an iterator, prefer that loop over others

If a language has an iterator implementation for a given collection, it probably has another loop implementation as well. For example, you can iterate through a standard string object in C++ using an index loop, a pointer loop, or an iterator. When multiple options are available, use the iterator loop. It is guaranteed to be at least as fast as the other options and is more generic. It also gives the author of the collection freedom to alter implementation details without impacting client code.

Instance Loops

The most modern and some would argue most intuitive way of iterating through an array is to use instances. Here, the instance variable is a reference to the element in the array. Note that instance loops are often called “FOREACH” loops.

Swift

```
for score in scores {  
    sum += score  
}
```

Figure 09.32

Looping through an array using instances

Notice that the start-condition, end-condition, and increment statement are not directly specified by the programmer. In fact, the programmer does not even have control over whether the loop goes forwards (low index to high) or backwards (high index to low). In most applications, this is perfectly acceptable; the programmer does not care how the loop is performed, just that every element is visited exactly once. In other words, the programmer is surrendering some descriptive power for the sake of making the code simpler and perhaps more efficient.

The instance implementation is just a more elegant way of using an iterator, which is itself a thin veneer over the pointer implementation. In other words, it is very efficient and sufficiently generic that it can be used for just about all collection types. Thus, the choice of an instance loop or an iterator loop is at the programmer’s discretion.

Most modern languages provide support for instance loops. Though the syntax varies wildly, the result is the same.

Java

```
for (int score : scores)  
    sum += score;
```

VB

```
For Each score As Integer In scores  
    sum += score  
Next
```

C++

```
for (const int & score : scores)  
    sum += score;
```

Ruby

```
scores.each { |score| sum += score }
```

Python

```
for score in scores:  
    sum += score
```

JavaScript

```
for (var score in scores) {  
    sum += score;  
}
```

C#

```
foreach (int score in scores)  
{  
    sum += score;  
}
```

Figure 09.33

Other implementations of instance loops

Examples

Example 09.1: Event-controlled loop

This example will demonstrate how to design an event-controlled loop.

Problem

Write the code necessary to address the following problem:

A budgeting program prompts the user for his or her monthly income. The value must be positive to be considered valid.

Solution

It is first necessary to determine which type of loop would be best to solve this problem. There is no collection and the number of iterations are not known from the onset. This rules out the counter-controlled loop and the collection loop. There also appears to be a simple controlling expression, precluding the need for a sentinel-controlled loop. Thus, an event-controlled loop is selected.

Two questions need to be answered: how do you know when you are finished, and what happens each iteration? The first question is answered with “when the user inputs an income that is zero or positive.” This can be encoded in a Boolean expression: `income ≥ $0.00`.

The second question is what happens each iteration. Here, we need to prompt the user for an income value and display an error message if the value is invalid. Notice that there will be one less error message than prompt because the last prompt will not be accompanied by an error message.

With these two pieces of information, the pseudocode can be written:

Pseudocode

```
PROMPT the user for income  
GET income  
  
WHILE income < $0.00  
    PUT error message  
  
    PROMPT the user for income  
    GET income
```

Example 09.2: Counter-Controlled Loop

This example will demonstrate a counter-controlled loop.

Problem

Write the code necessary to address the following problem:

A prime number is only evenly divisible by itself and the number one. Write the code to prompt the user for a number and display whether that number is prime.

Solution

It is first necessary to determine which type of loop would be best to solve this problem. Since the number of iterations is known from the beginning, this is a counter-controlled loop.

We will start at the number 2 because all numbers are evenly divisible by 1. The end-condition is tricky. It may appear that we will end at the number being considered. However, we only need to go to the square-root of *n* rather than *n* itself. We will iterate by ones. Finally, the loop body will check the remainder when dividing the counter by the number being considered:

Loop part	Description
Start-condition	2
End-condition	square root of number
Increment	count++
Loop body	number % count

Pseudocode

```
PROMPT the user for a positive number  
GET number  
  
FOR counter ← 2 ... √number  
    IF number % counter = 0  
        PUT not prime!  
  
    PUT prime!
```

There are three problems with this. First, we compute the square root of **number** on each iteration. We should move this out of the end-condition. Second, this will display “not prime!” on each iteration. Finally, we should exit the loop once we have determined that the number is not prime. We can solve all these at once.

Pseudocode

```
PROMPT the user for a positive number  
GET number  
  
isPrime ← TRUE  
endCondition ← √number  
FOR counter ← 2 ... endCondition AND isPrime  
    isPrime ← isPrime AND number % counter = 0  
  
PUT isPrime
```

Example 09.3: Sentinel-Controlled Loop

This example will demonstrate a sentinel-controlled loop.

Problem

Write the code necessary to address the following problem:

A student must continue to take a class until a grade of greater than D is earned and the student did not cheat.

Solution

It is first necessary to determine which type of loop would be best to solve this problem. Notice that two completely unrelated conditions are present here. Thus, a sentinel-controlled loop is a good candidate.

The first condition is whether the student earned a grade greater than a D. This is accomplished with a prompt and an IF statement:

Pseudocode

```
PROMPT for grade  
GET grade  
IF grade ≥ 70%  
    passed
```

The second condition is whether the student cheated. Again, this is accomplished with a prompt and an IF statement:

Pseudocode

```
PROMPT if the student cheated  
GET cheated  
IF cheated = TRUE  
    failed
```

We will identify a sentinel and place it all together in a single DO-WHILE loop.

Pseudocode

```
passed ← FALSE  
  
DO  
    PROMPT for grade  
    GET grade  
    IF grade ≥ 70%  
        passed ← TRUE  
  
    PROMPT if the student cheated  
    GET cheated  
    IF cheated = TRUE  
        passed ← FALSE  
    WHILE !passed
```

Example 09.4: Collection-Controlled Loop

This example will demonstrate how to iterate through a collection.

Problem

Write the code necessary to address the following problem:

User-provided text could be in UPPER CASE, lower case, or MiXeD CaSe. Turn this text into Title Case.

Provide as many versions of this code as possible.

Solution

The first version will use an index to iterate through the string. The loop variable will be the index `i` and the way to access each element will be `text[i]`.

C++

```
bool setToUpper = true;

for (int i = 0; i < text.size(); i++)
{
    text[i] = setToUpper ? toupper(text[i]) : tolower(text[i]);
    setToUpper = isspace(text[i]);
}
```

The second version will use a pointer, the variable being `p` and access will be `*p`.

C++

```
bool setToUpper = true;

for (char * p = text.c_str(); *p; p++)
{
    *p = setToUpper ? toupper(*p) : tolower(*p);
    setToUpper = isspace(*p);
}
```

The third version will use an iterator (there is no linked list loop for strings).

C++

```
bool setToUpper = true;

string::iterator it;
for (it = text.begin(); it != text.end(); it++)
{
    *it = setToUpper ? toupper(*it) : tolower(*it);
    setToUpper = isspace(*it);
}
```

Exercise

Exercise 09.1: Types of Loops

From memory, describe when each of the following loop types should be used.

Loop type	Situation
Event-controlled	
Counter-controlled	
Sentinel-controlled	
Collection: Index	
Collection: Pointer	
Collection: Linked list	
Collection: Iterator	
Collection: Instance	

Exercise 09.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Event-controlled loops use FOR statements	
Counter-controlled loops increment	
Event-controlled loops have four considerations	
Sentinel-controlled loops can use DO-WHILE	
Collection-controlled loops use WHILE	
Sentinel-controlled loops have a counter	
Instance loops are also called FOREACH loops	

Exercise 09.3: Identify the Loop Type

Identify the loop type from the following example:

```
Pseudocode
timeToGo ← false

WHILE !timeToGo
    PROMPT is it midnight?
    GET deadline
    IF deadline = TRUE
        timeToGo ← true

    PROMPT what are we going to do tonight?
    GET plan
    IF plan = "cruise main street"
        timeToGo ← true

    IF creepy = TRUE
        timeToGo ← true

PUT please take me home; I think our date is over.
```

Exercise 09.4: Identify the Loop Type

Identify the loop type from the following example:

```
Pseudocode
highest ← 0%

FOREACH grade IN grades
    IF grade > highest
        highest ← grade

PUT the best grade in class is: highest%
```

Exercise 09.5: Identify the Loop Type

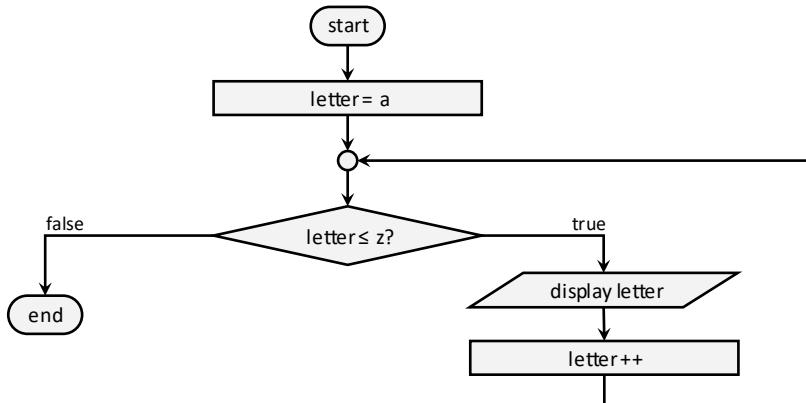
Identify the loop type from the following example:

```
Pseudocode
level ← UNKNOWN

WHILE level = UNKNOWN
    PROMPT for level
    GET level
    IF level ≠ NINJA or EXPERT or NOVICE
        PUT error message and prompt again
    level ← UNKNOWN
```

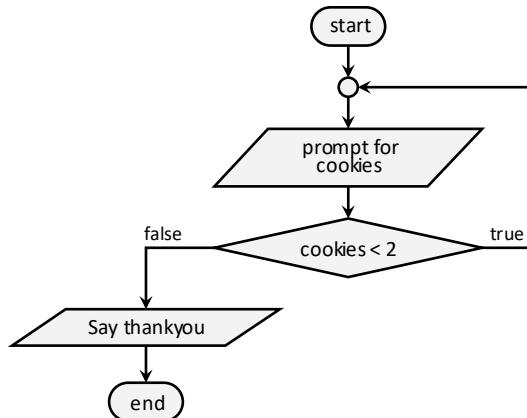
Exercise 09.6: Identify the Loop Type

Identify the loop type from the following example:



Exercise 09.7: Identify the Loop Type

Identify the loop type from the following example:



Exercise 09.8: Identify the Loop Type

Identify the loop type from the following example:

```
C
char * text = "Software Design";
int length = 0;

while (*text++)
    length++;

printf("Length of string: %d", length);
```

Problems

Problem 09.1: Sum the Values

There are five parts to the following problem:

Prompt the user for a number and sum the values from 0 to the number. For example, if the user enters '5,' then sum $0 + 1 + 2 + 3 + 4 + 5$

1. Identify the type of loop (event, counter, sentinel, or collection. If collection, what flavor?). What is your rationale for using this loop type?
2. Write the pseudocode to solve this problem.
3. Test your pseudocode using program trace.
4. Implement your solution in the programming language of your choice.
5. Test your solution.

Problem 09.2: Read From a file

There are five parts to the following problem:

Read all the names from a file and display them on the screen.

1. Identify the type of loop (event, counter, sentinel, or collection. If collection, what flavor?). What is your rationale for using this loop type?
2. Write the pseudocode to solve this problem.
3. Test your pseudocode using program trace.
4. Implement your solution in the programming language of your choice.
5. Test your solution.

Problem 09.3: Guessing Game

There are five parts to the following problem:

A simple guessing game will randomly generate a number between 1 and 100. The game will prompt the user for the number. If she guesses correctly, then the game quits. If the user types in -1, then she gave up and the game quits. If the user has failed to guess the number in 7 tries, then the game quits. Otherwise, the game prompts the user for the number again.

1. Identify the type of loop (event, counter, sentinel, or collection. If collection, what flavor?). What is your rationale for using this loop type?
2. Write the pseudocode to solve this problem.
3. Test your pseudocode using program trace.
4. Implement your solution in the programming language of your choice.
5. Test your solution.

Problem 09.4: Powers of 2

There are five parts to the following problem:

Prompt the user for a positive number n . Display the first n powers of two. For example, if the user enters '5,' then display 2, 4, 8, 16, 32

1. Identify the type of loop (event, counter, sentinel, or collection. If collection, what flavor?). What is your rationale for using this loop type?
2. Write the pseudocode to solve this problem.
3. Test your pseudocode using program trace.
4. Implement your solution in the programming language of your choice.
5. Test your solution.

Problem 09.5: Net Worth

There are five parts to the following problem:

The user has an array of account balances. Some are positive (such as checking, savings, and investment), and some are negative (such as mortgage, credit card, and car loan). Write a function to compute his/her net worth: the summation of all his/her accounts.

1. Identify the type of loop (event, counter, sentinel, or collection. If collection, what flavor?). What is your rationale for using this loop type?
2. Write the pseudocode to solve this problem.
3. Test your pseudocode using program trace.
4. Implement your solution in the programming language of your choice.
5. Test your solution.

Problem 09.6: Tic-Tac-Toe

There are five parts to the following problem:

The game of tic-tac-toe has a board where each square has an X, O, or space in it. It is Xs turn if there are an odd number of spaces and Os turn if there are an even number. Write the code necessary to determine whose turn it is.

1. Identify the type of loop (event, counter, sentinel, or collection. If collection, what flavor?). What is your rationale for using this loop type?
2. Write the pseudocode to solve this problem.
3. Test your pseudocode using program trace.
4. Implement your solution in the programming language of your choice.
5. Test your solution.

Challenges

Challenge 09.1: Insertion Sort

Research on the web how the insertion sort works. Please do the following:

1. Write the code to read a collection of numbers from a file into an array.
2. Implement the insertion sort algorithm.
3. Display the array before the sort and after the sort.
4. Write the code necessary to verify that the sorted array is in fact sorted.

Challenge 09.2: Fibonacci

The Fibonacci value for a given number can be computed by adding the previous two values in the sequence:

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n - 1) + F(n - 2) & \text{if } n > 1 \end{cases}$$

Please do the following:

1. Write the code to prompt the user for a positive number N .
2. Allocate an array of the size of N .
3. Write the code to put the first N Fibonacci numbers in the array.
4. Display the results.

Challenge 09.3: Sudoku

Sudoku is a numbers game where a board is stored on a 9x9 grid. Each element in the grid can have the number 1 through 9 or a space represented with the number zero. The board is further subdivided into 9 3x3 inside squares. The rules of Sudoku are the following:

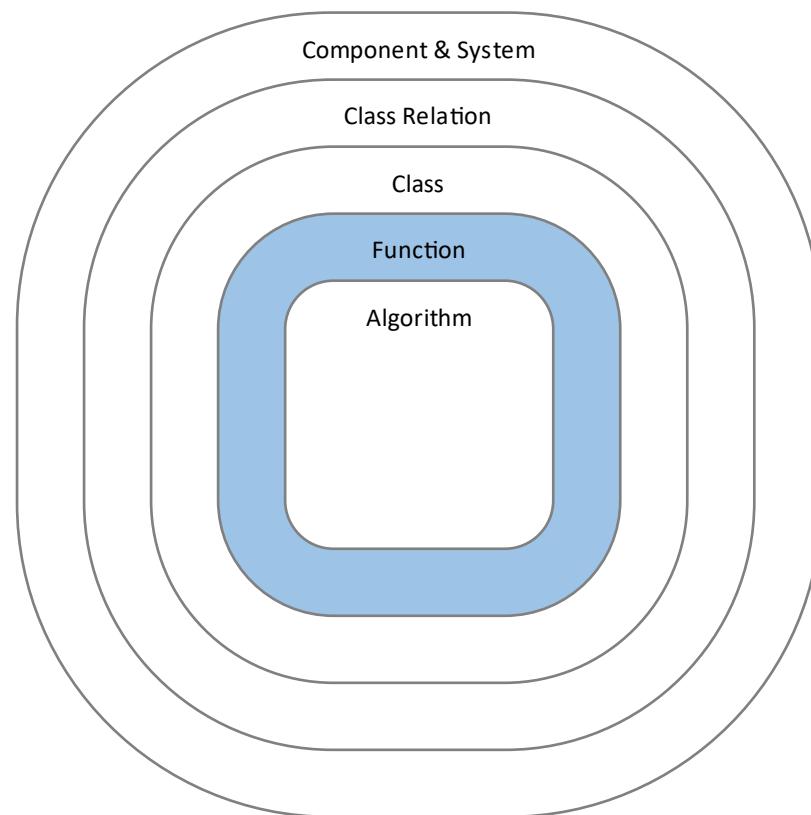
- Unique Row: No number (aside from 0) can be on a given row more than once.
- Unique Column: No number can be on a given column more than once.
- Unique Inside Square: No number can be on a given inside square more than once.

Please do the following:

1. Write the code to read a Sudoku board from a file.
2. Write the code to determine if any of the Sudoku rules are violated.
3. Display the results.

Modularization Design

Modularization is the process of subdividing a large or complex program into smaller components called functions. It is concerned with how these functions are defined and how they communicate with each other. Functions builds off algorithms in that each function contains an algorithm.



Structure Chart

Chapter 10

A structure chart is a graph that represents the functions in a program and how they call each other.

Functions are a fundamental tool to organize code into coherent units, making programs easier to understand and modify. This is certainly true when a program consists of five or ten functions. What happens when a program consists of hundreds, thousands, or even hundreds of thousands of functions? Modern programs are often this big or even bigger! How can one possibly keep track of everything? Structure charts were created to address this challenge.

A structure chart represents the functions in a program and how they call each other. It is a simple tree-like graph where the nodes are the functions of a program and the edges represent how the functions reference each other. Consider the following structure chart representing the display code of a list management application.

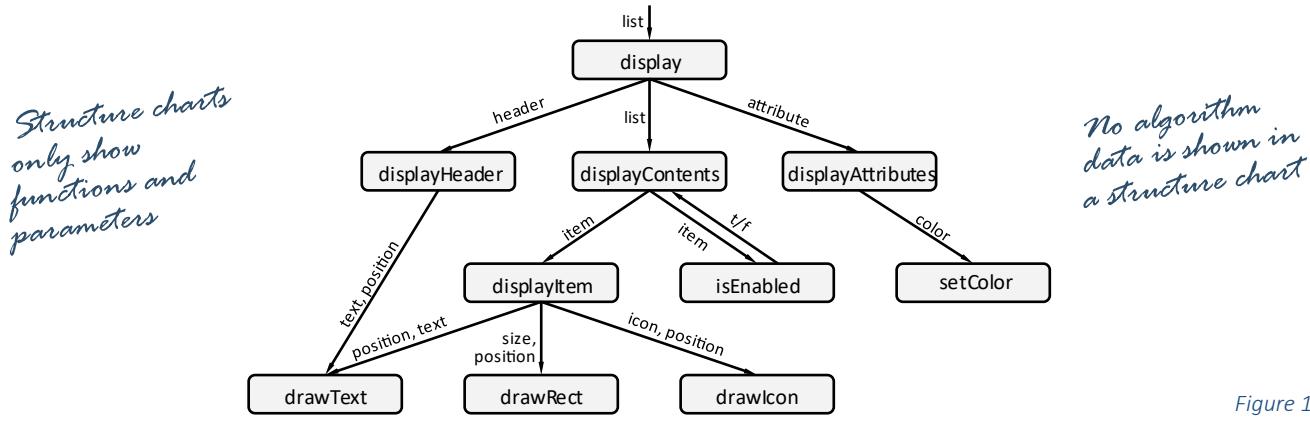


Figure 10.1:
Structure Chart

From this chart, we can see that `display()` calls three functions: `displayAttributes()`, `displayContents()`, and `displayHeader()`. We cannot tell the order or the number of times the functions are called; those details are captured with pseudocode or flowcharts. This structure chart also illustrates that `drawText()` gets invoked by `displayHeader()` and `displayItem()`, passing the same `position` and `text` parameters each time.

Structure charts have the following properties:

Property	Description
Use	Design and illustrate how functions call each other
Viewpoint	Development: showing the organization of the code
Strength	Easy to create, read, and translate into code
Weakness	Can get very large and complex with big projects

Viewpoints

It is easy to confuse flowcharts with structure charts. They are both graphical representations of a program, after all! This will get even more confusing as more tools are introduced later. Each of these tools represent software design from a different perspective. In many ways, this is like the design of a house. The blueprints, plumbing diagrams, and wiring diagrams complement each other, each representing a different aspect of the design. The same is true with flowcharts and structure charts. They complement each other; they do not compete.

In 1995, Philippe Kruchten introduced the “4+1” view model of software architecture. This model lists four “viewpoints” or perspectives on the software design problem:

Viewpoint	Description
Process	The verbs of the system. How algorithms work.
Development	How a program is partitioned and organized.
Logical	The nouns of the system. How data are stored.
Physical	The hardware and physical components.

Flowcharts and pseudocode are Process viewpoints

In Unit 0, we learned about flowcharts and pseudocode. Both are process viewpoints of the system. They describe algorithms and how tasks are carried out. In other words, they describe what goes on inside the functions of a program.

Class diagrams are Logical viewpoints

In Unit 1, we will learn about structure charts and data flow diagrams. They are development viewpoints, describing how code is organized and how the program is structured.

Structure Charts and Data Flow Diagrams are Development viewpoints

Units 2 and 3 will be about classes. Classes are essentially nouns, describing how data are stored. Class diagrams will be used to describe classes, a logical viewpoint.

Finally, in Unit 4, we will learn about component diagrams. These are also development viewpoints. In many ways, component diagrams are a fusion of structure charts and data flow diagrams. The difference, however, is that they work on a much larger scale.

Component Diagrams are also Development viewpoints

Selecting the Right Tool

Each viewpoint describes the system from a different perspective. If you are having a difficult time expressing your design ideas with a given tool, then you are probably using the wrong tool! The following guide will help you:

Scenario	Tool
Help me untangle a complex decision tree!	Flowchart
Help me figure out a complex algorithm!	Pseudocode
How are the functions organized?	Structure Chart
Where did the value in this variable come from?	Data Flow Diagram
What is this class supposed to do?	Class Diagram
What is the overall layout of the system?	Component Diagram

There are other tools as well. The sequence diagram and timing diagram are useful for unraveling parallelism and concurrency challenges. Entity relationship diagrams describe database tables and how they relate to each other. In addition to these mainstream tools, there are a plethora of special-purpose ones designed to help with specific programming challenges.

Structure Chart Elements

Structure charts are tree-like graphs consisting of nodes and edges. The functions of a program are the nodes, whereas the way that functions call each other is represented by the edges.

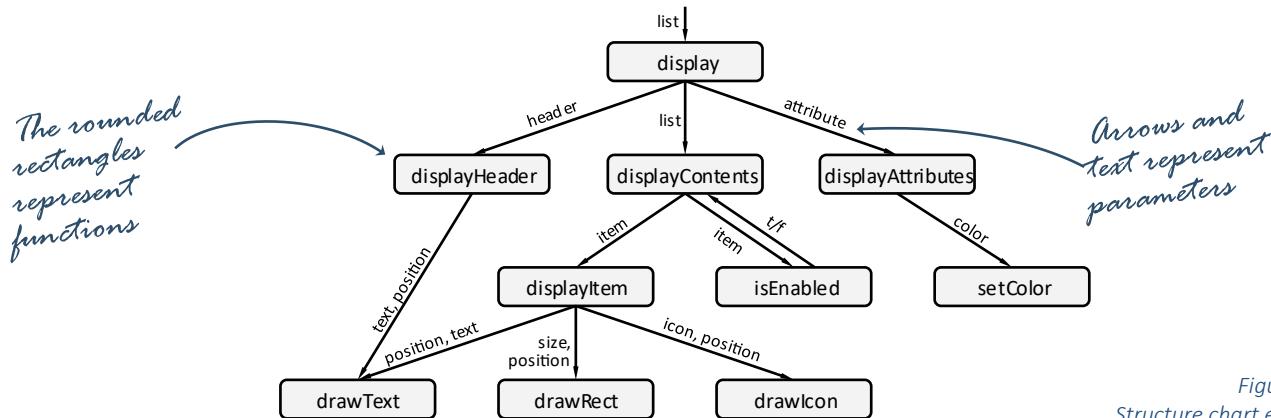


Figure 10.2:
Structure chart elements

Structure charts describe the relationship between the caller and the callee. The caller is the function that initiates a call to another function. In other words, there exists a function call to a callee from within a caller. In a well-designed program, all functions should have at least one caller except for main. If a function does not have a caller, then it is never used and perhaps should be removed from the project. In many cases, a single callee can be invoked by many callers.

Caller	Initiating a function call
Callee	Receiving a function call

Structure charts consist of only three symbols.

Symbol	Meaning
function name	Rounded rectangles represent functions
	Lines represent one function calling another
parameter name	Parameters passed between functions are labeled

Functions

Functions are represented as ovals or rounded rectangles in structure charts. The name of the function is the text contained therein. We do not represent variables, processes, or any other program constructs. That is because a structure chart is designed to only represent functions and their relationships.

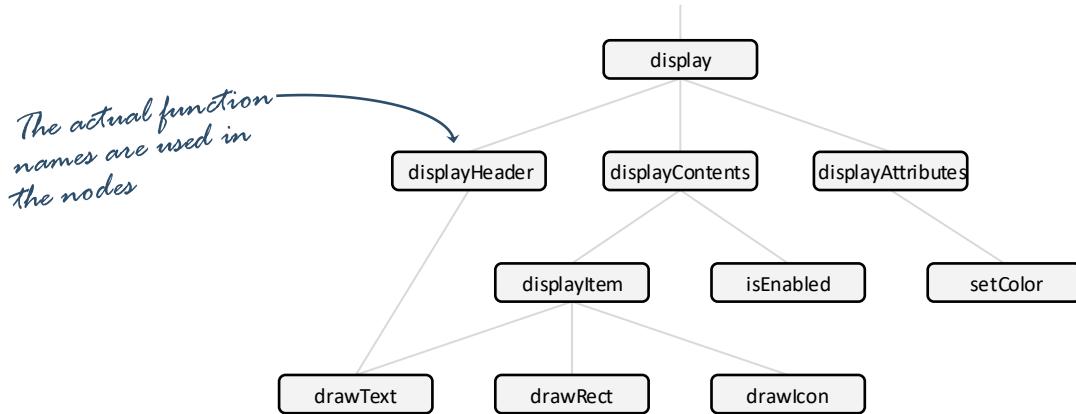
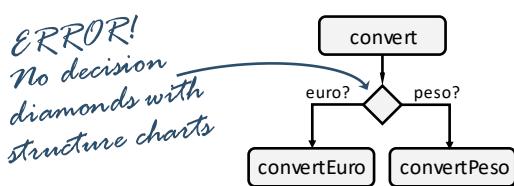


Figure 10.3:
Functions in a structure chart

Rule 10.1: The nodes of a structure chart are functions

Every node in a structure chart is a function. It is not an action, a process, or a collection of functions. If you are trying to represent a part of the program that is smaller than a function, then pseudocode or a flowchart would be a better tool for the job. If you are trying to represent a part of the program which encompasses many functions or large parts of the system, then a UML component diagram would be a better tool for the job.

Rule 10.2: Structure charts only represent how functions call each other



A common mistake is to try to turn a structure chart into a flowchart. If you find yourself adding decision diamonds or loops, then you are probably walking down the flowchart path. Recall that a structure chart is a development viewpoint, describing how code is organized. It is not a process viewpoint, describing how an algorithm works.

Figure 10.4:
Rule 10.2 – do not try to
represent an algorithm
with a structure chart

Lines

Lines are used in structure charts to indicate that one function calls another. By convention, the caller is on top and the callee is on the bottom. These lines correspond to the edges in the structure chart graph.

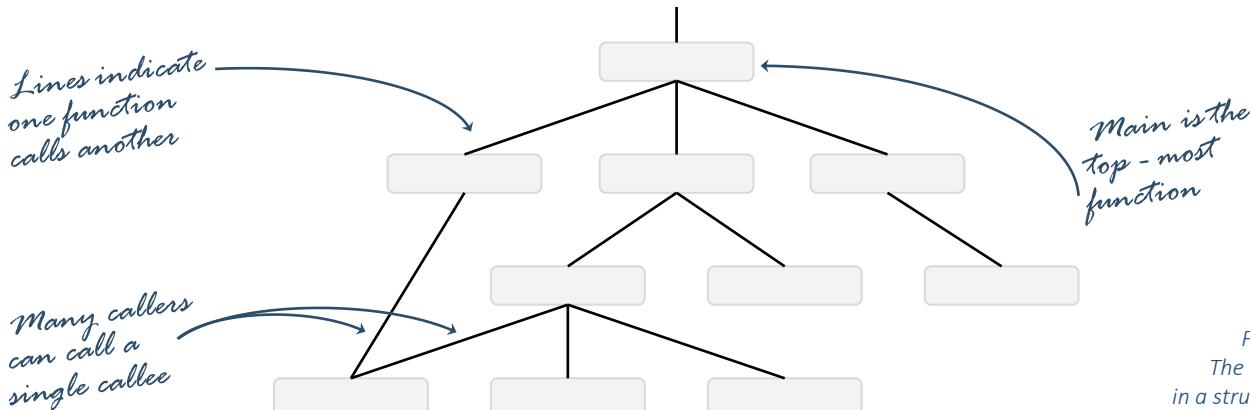


Figure 10.5:
The use of lines
in a structure chart

Rule 10.3: Program execution starts at the top of the graph

Most structure charts begin at the top of the page and work down. This means that the topmost function is `main()` or whatever the equivalent is in your programming language of choice. If we do not impose some order, then it is easy for the structure chart to become a confusing mess of functions and lines.

Structure charts often look like trees but, technically speaking, they are not. Trees are graphs where each node has one incoming edge and zero or more outgoing edges. This does not hold true with a structure chart because one function can be called from many different locations. Thus, a structure chart may not “branch out” like a tree.

Rule 10.4: When a caller invokes a callee, control returns to the caller when finished

Another common mistake is to try to make a structure into a flowchart. If, for example, the functions `callee1()`, `callee2()`, and `callee3()` need to be called in sequence, then it is tempting to depict a circuit where control flows from `callee1()` then to `callee2()` then to `callee3()`. This is a mistake! Control-flow is depicted with flowcharts rather than with structure charts. Instead, have a single caller first invoke `callee1()`. When `callee1()` is finished, then `caller()` would invoke `callee2()` and so on.

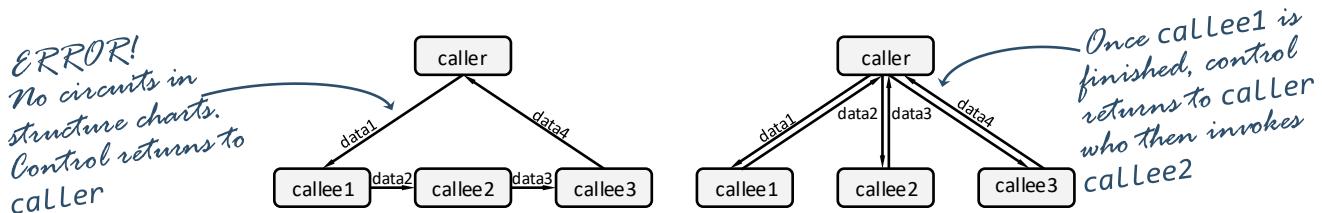


Figure 10.6:
Rule 10.4 – control
returns to the caller when
the callee is finished

Arrows and Parameters

Some simplified structure charts contain only functions and lines. Most find this level of detail insufficient to design a program or understand a given codebase. The missing piece is how data are passed between functions. To do this, we need two things: arrows and parameters. Arrows in structure charts are not used to represent which function is the caller and which is the callee. Instead, it indicates the direction a parameter is passed between functions. If there are no data passed between two functions, then a simple connecting line is used.

Figure 10.7: No arrows



If the caller sends data to the callee and nothing is returned, then an arrow leads from the caller to the callee. We also label the data passed between the functions by the name of the parameter.

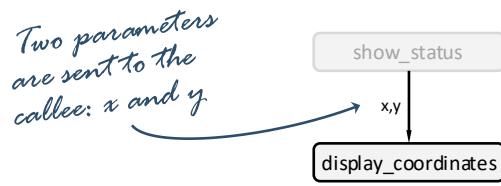
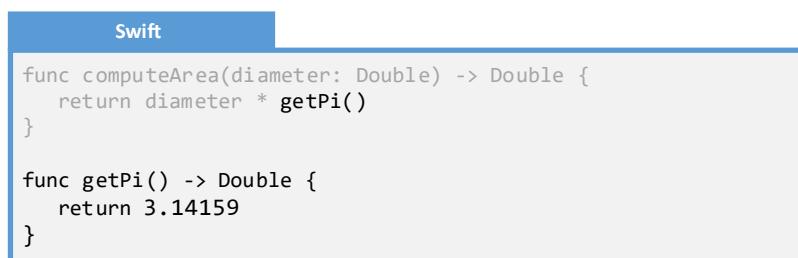


Figure 10.8: Arrow from caller to callee

Figure 10.9: Arrow from callee to caller

If only the callee returns data to the caller, then a single arrow leads from the callee up to the caller. As before, parameters are named.



Finally, if data are both passed to the caller and returned to the callee, then two arrows are used. Here, the arrows are parallel to each other and somewhat close together. Parameters are labeled in both directions.

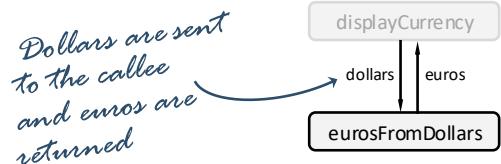


Figure 10.10: Arrows both from and to the callee

This fifth rule stems from how functions are implemented in most programming languages. Every function has a signature, which includes the name of the function, the input it expects, and the output it sends. Anything calling this function must provide the expected input and consume the resulting output. It clearly does not make sense to have one caller provide three parameters and a second caller provide one!

*ERROR!
caller1 sends
two parameters
while caller2
sends one*

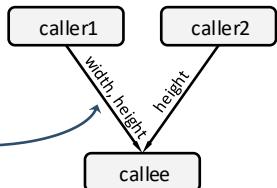


Figure 10.11:
The two callers use a
different number of
parameters

The first common mistake is for two different callers to invoke a single callee sending a different number of parameters. In the following example `caller1()` is sending two parameters to `callee()` whereas `caller2()` is sending one. There is an exception to this rule. Some languages have something called “default parameters.” This is a mechanism where, if the caller omits one or more parameters, then default values are provided. In this case, this structure chart would not be in error.

The second common mistake occurs when two callers send different data to a single callee. Here, the first is sending a textual name to the callee whereas the second is sending a numerical height. In the function declaration, the programmer needs to specify exactly what parameters are needed for a given function. The structure chart needs to reflect this reality. Note that some languages allow for “overloading”: having two functions with different signatures have the same function name. If this is the case, then this structure chart will not be in error.

*ERROR!
callerX is sending
a parameter
while callerY is
receiving one*

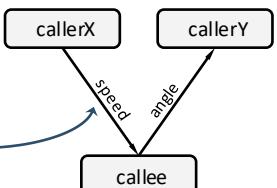


Figure 10.13:
One caller sending data,
one receiving it

A final common mistake is when one function expects the callee to return something but the second callee does not, or when one function sends the callee data but the second does not. In either case, the two callers are not agreeing upon the function signature of the callee. When reviewing a structure chart, check every function to verify that all the callers provide the same parameters. If one caller fails to provide one or more

parameters, then the potential for a bug exists. If one caller provides a different set of parameters than another, then perhaps two callees are needed instead of one. In each of these cases, it is easier to find and fix the mistake while analyzing the structure chart than it is when writing the code.

*ERROR!
callerA sends a
string while
callerB sends a
number*

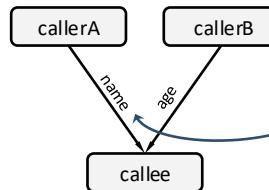


Figure 10.12:
Two callers are
sending different
types of parameters

Variations

Through the years, several standards and notations have been adopted for structure charts. The first came from 1979 with Larry Constantine and Edward Yourdon's popular Structured Design book. Here, they included some algorithm constructs such as loops and decisions in their structure charts. Today we generally avoid this, leaving flowcharts and pseudocode to represent them.

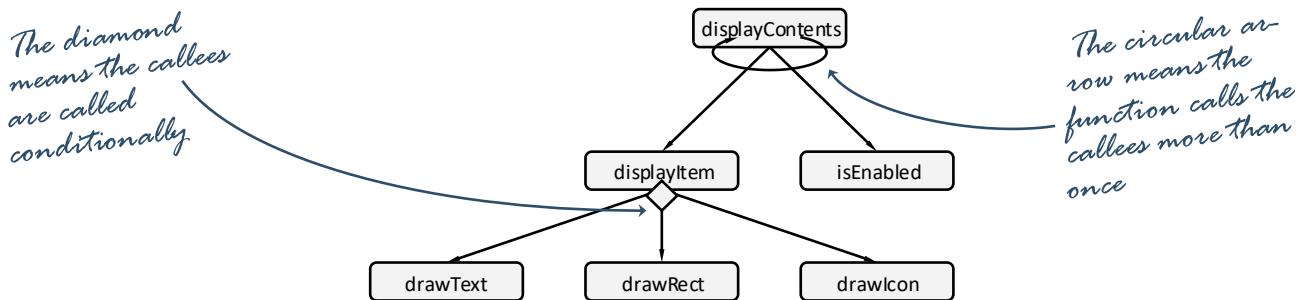


Figure 10.14: Constantine structure chart notation

In 1981, Sarah Brooks popularized another notation. This one uses a different way to describe the parameters of a program.

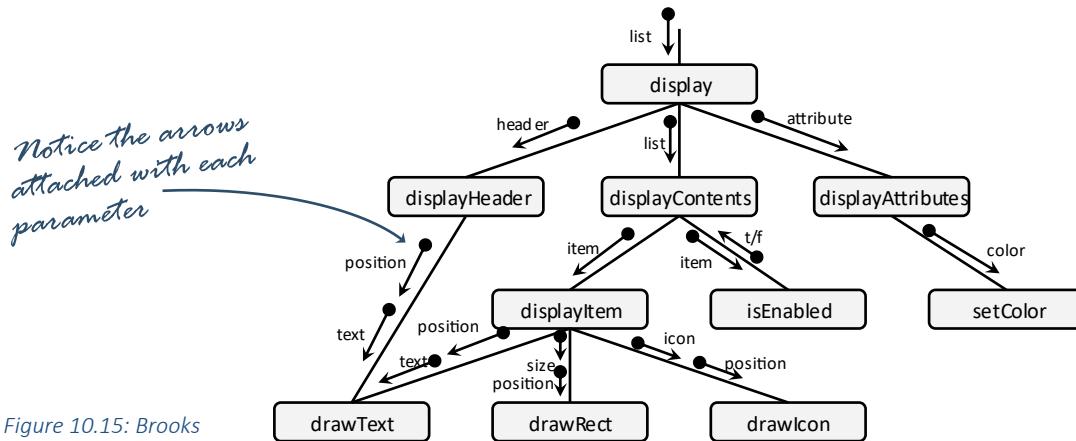


Figure 10.15: Brooks structure chart notation

Neither the Constantine nor the Brooks notation for structures are commonly used today due to their extra complexity and visual clutter.

Designing with a Structure Chart

Imagine a programmer wishing to create a program to maintain a grocery shopping list for a mobile application. We will start big. What are the main tasks that this program will do? It will read the old list from a file, display the list on the screen, allow the user to update the list, and then save the list back to the file. Even though we are unsure of how each of those tasks will be implemented, we can start by creating functions to represent them.

*First iteration
gives us only a
few functions*

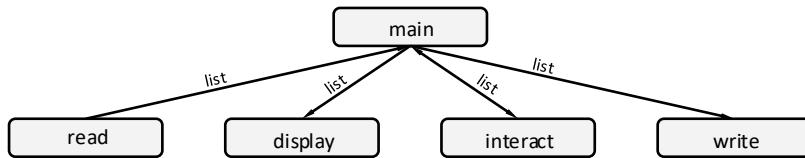
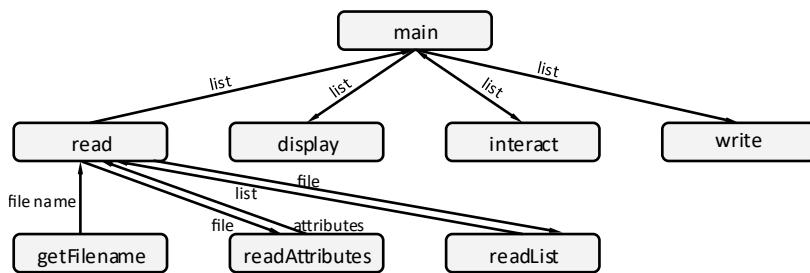


Figure 10.16:
First iteration

Now we will focus on the `read()` function. How will that work? It will need a file name, it will need to read in the attributes of the grocery list (creation date, owner, etc.), and then it will need to read in the list of records. Knowing no more than that, we will create functions representing these tasks and add them to our design.



*The second
iteration reveals
many details about
how we are to read
from a file*

Figure 10.17:
Second iteration

The `readList()` function appears to be the most challenging. What helper functions will it need to accomplish its task? It will probably need a `readItem()` function which reads a single list item from the file.

*The third iteration
shows us yet
another function:
one to read a
single record*

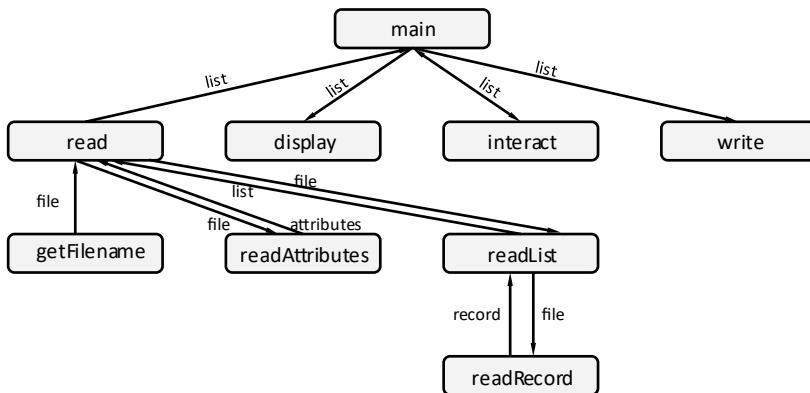


Figure 10.18:
Third iteration

This process continues until a roadmap exists containing the most important functions we will need to write for our grocery list program.

Examples

Examples 10.1: Modeling Existing Code

This first example will demonstrate how to create a structure chart to represent existing code. This is useful in helping the programmer understand a program.

Problem

Create a structure chart representing the following code.

```
JavaScript
function resetMachine() { ... }

function init() {
    ...
    resetMachine();
    update(document);
}

function update(doc) {
    updateDisplay(doc);
    updateMemory(doc);
    updateRegisters(doc);
}

function updateRegisters(doc) { ... }

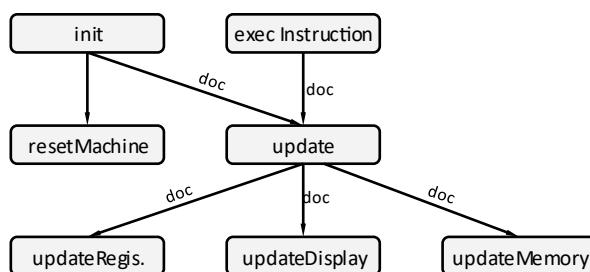
function updateDisplay(doc) { ... }

function updateMemory(doc) { ... }

function executeInstruction() {
    ...
    update(document)
}
```

Solution

Notice that this program is in JavaScript so there can be more than one entry point. From the code we have, there appear to be two: `executeInstruction()` and `init()`. Notice that we don't need to know many details about what goes on inside the functions to write a structure chart.



Examples 10.2: Designing a Program

This second example will demonstrate how to design a program with a structure chart.

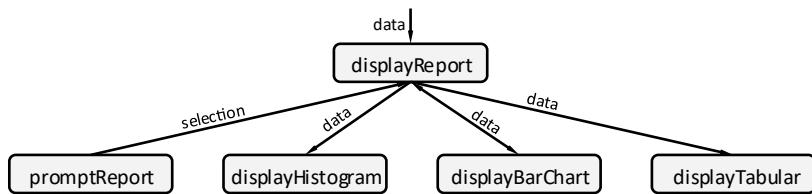
Problem

Design a program to display one of several financial reports to the user.

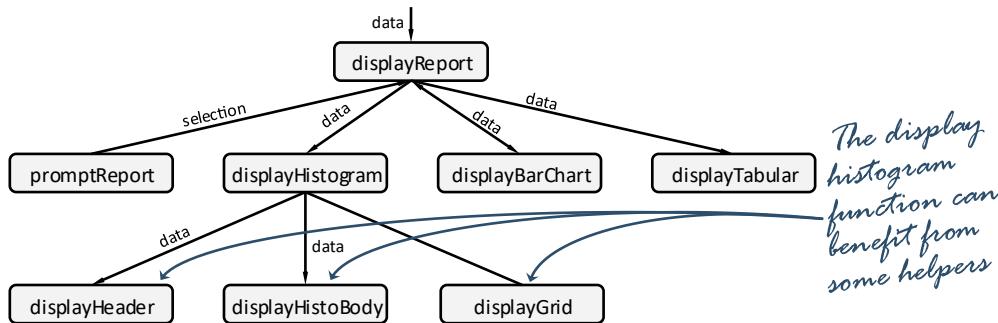
Solution

The first iteration will just represent the different report types and the prompt function asking the user which report she needs.

We start with three display functions and a prompt

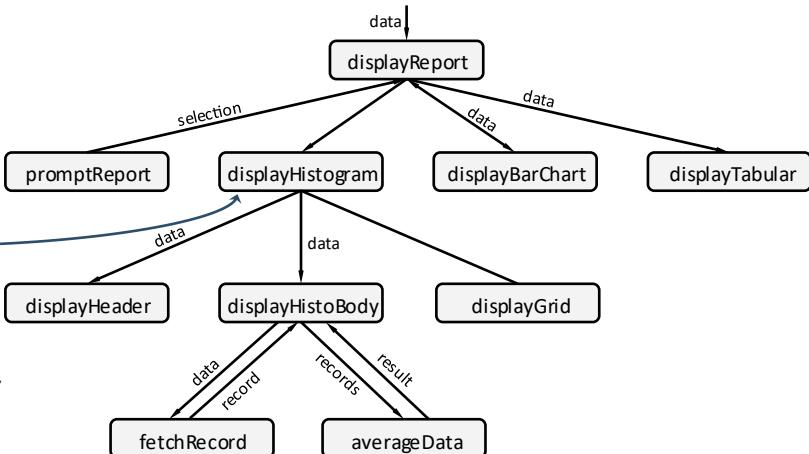


We will focus on the histogram report for the time being. This consists of a header, the actual data, and the grid on which the data are presented. Each will be a function.



One more iteration will focus on how the body of the histogram is displayed.

The histogram report is fleshed out enough that we can start writing pseudocode



Exercises

Exercises 10.1: Scenarios and Tools

For each of the following scenarios, select the best tool for the job.

Scenario	Tool
You would like to model the classes in a new program	
The business logic for a program is very complex involving many complex decision points	
You would like to figure out which functions are needed and how they call one another	
The algorithm to organize a collection of items is very complex, needing several loops	
You would like to model how user input flows through the system	
Who calls this function and with what parameters?	

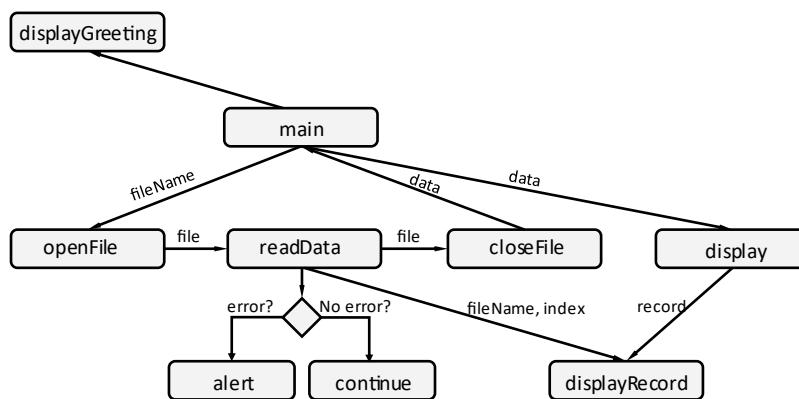
Exercises 10.2: Structure Chart Elements

Draw the various components of a structure chart:

Description	Drawing
Function A() calls function B() but no data are passed and there is no return value.	
There is a function called <code>computePay()</code> .	
The function <code>getIncome()</code> takes no parameters and returns a floating-point number: <code>income</code> .	
The function <code>displayCoordinates()</code> takes three parameters (<code>x</code> , <code>y</code> , and <code>z</code>). The output of this function is displayed on the screen.	
The function <code>displayInstructions()</code> takes no parameters and returns no data. It is called by three functions: <code>interact()</code> , <code>getCoordinates()</code> , and <code>setCoordinates()</code> .	
The function <code>computeBudget()</code> calls two helper functions: <code>computeIncome()</code> and <code>computeSpending()</code> .	

Exercises 10.3: Structure Chart Errors

Identify the errors in the following structure chart. If possible, list the rule number and description corresponding to each error.



Exercise 10.4: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Lines represent how functions call each other	
Ovals represent start/stop locations in the algorithm	
Structure charts represent the development viewpoint	
The labels next to arrows can represent decision options	
Diamonds represent decision points in the algorithm	

Problems

Problem 10.1: Budget

Create a structure chart matching the following draft of a Python program to display a simple monthly budget report:

```
Python
def getIncome():
    ...
    return income

def getExpenses():
    ...
    return expenses

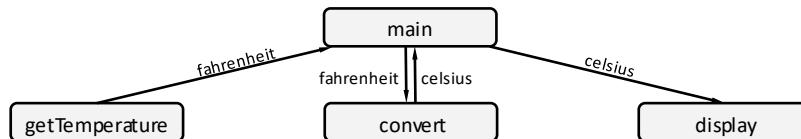
def computeTax(income):
    ...
    return tax

def displayBudget(income, expenses, tax):
    ...

income = getIncome()
expenses = getExpenses()
tax = computeTax(income)
displayBudget(income, expenses, tax)
```

Problem 10.2: Temperature Conversion

In the programming language of your choice, write the code corresponding to the following structure chart:



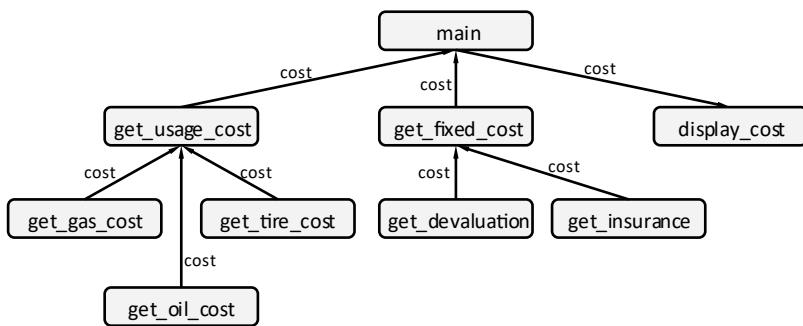
Problem 10.3: MadLib

Create a structure chart matching the following draft of a pseudocode algorithm to play the game MadLib:

Pseudocode
askQuestion(question) ... RETURN response
readFile(fileName) ... story[i] ← askQuestion(word) ... RETURN story
getFilename() ... RETURN fileName
playAgain() ... RETURN true
displayStory(story) ... RETURN
main() DO story ← readFile(getFilename()) displayStory(story) WHILE playAgain()

Problem 10.4: Cost of a Car

In the programming language of your choice, write the code corresponding to the following structure chart:



Problem 10.5: Tic-Tac-Toe

Create a structure chart for a program to satisfy the following scenario:

Tic-Tac-Toe, otherwise known as noughts and crosses, is a game played on a 3x3 grid where one player is represented with Xs and the other is represented with Os. The X player puts a mark in one of the nine grid locations. The O player then selects one of the eight remaining grid locations. The game continues until either one player gets three in a row or three-diagonally.

This game will read a board from a file, allow the user to play the game, and save a partially-completed game back to a file. The user will be prompted for his or her selection, then the program will then choose a best response to the user's move. In other words, the user will play against the computer.

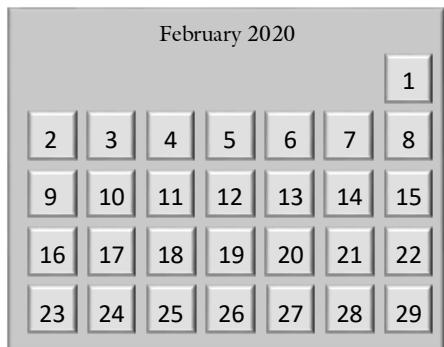
Hint: What functions will be needed to accomplish this task? What information will each function need in order to accomplish it?

Challenges

Challenge 10.1: Calendar

Create a structure chart to represent the part of a program which will display a calendar on the screen. The input to the main function will be a month and year, both integer parameters. The output of the function will be a graphical depiction of a calendar.

A personal finance program needs to display a calendar in a dialog from which the user will select the due-date for a bill. You are tasked with writing this code.



Hint: Your program will need to determine the day of the week of the first of the month. This will be determined by counting the days since the 14th of September, 1752 (which was a Thursday) and dividing by 7. It will need to know the number of days in each month, taking leap years into account.

Challenge 10.2: Asteroids

Consider the 1979 Atari game *Asteroids*. You may need to do some “research” to refresh yourself how this program works.

Create a structure chart representing this entire program.

Challenge 10.3: Sudoku

Create a structure chart representing the game of Sudoku:

Sudoku is a numbers game played on a 9x9 grid. The object of the game is to fill in the 9x9 grid while honoring certain constraints. The constraints are: 1) There is no more than one instance of a given number on a given row. 2) There is no more than one instance of a given number on a given column. 3) There is no more than one instance of a given number on an inside square (the 3x3 squares embedded in the 9x9 grid). 4) Every square can consist of a single digit between 1 and 9 exclusively, or can be blank.

Data Flow Diagram

A data flow diagram is a graph representing the movement of data between program entities.

Consider a programmer trying to fix a bug in a large and complex codebase. After an hour of investigation, she has isolated the malfunction to a single variable in an IF statement. In order to get to the root of the problem, she needs to track these data back to their source. How can she do this? The data transforms themselves many times as they pass through the program! This is where the data flow diagram (DFD) comes in.

A DFD is a graph representing the movement of data between program entities. It tracks data from the source where they enter the system, notes as the data are transformed through various processes, and illustrates how the data are eventually stored or sent back out of the system.

A DFD represents the movement of data between program entities

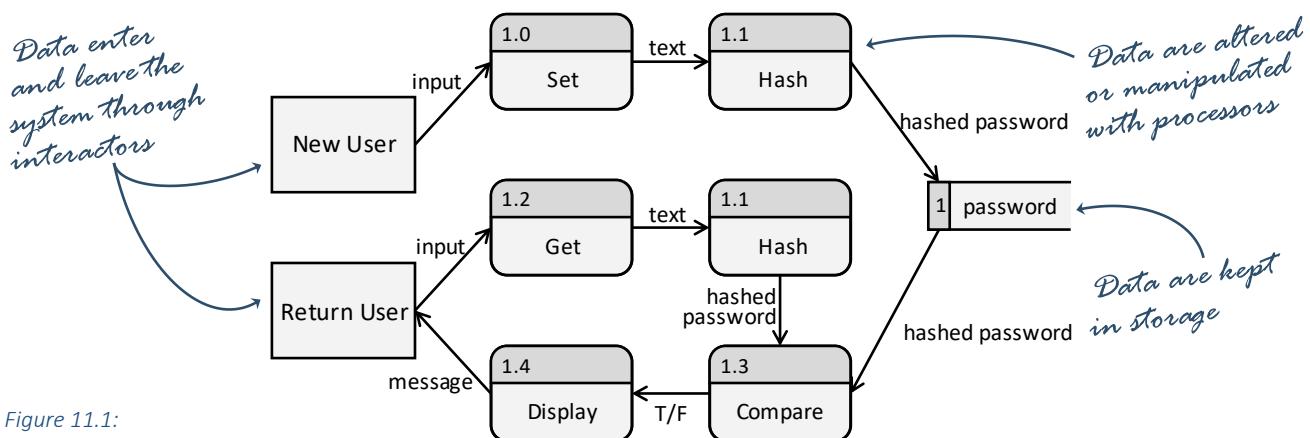


Figure 11.1:
Data flow diagram

In the above figure (Figure 11.1), we can easily see how data enter and leave the password field. We can see that all passwords originate from a new user. We can also see that passwords never leave the system. Only a *true/false* Boolean value is ever sent to the user. This type of information is difficult to ascertain from a structure chart or a flowchart, but is easy to visualize and verify with a DFD.

DFDs have the following properties:

Property	Description
Use	Design and illustrate how data flow through the system
Viewpoint	Development: showing the organization of the code
Strength	Easy to create, read, and translate into code
Weakness	Often gets confused with structure charts and flowcharts

Viewpoints

Each design visualization tool serves a different purpose in helping a developer design and evaluate a given program. The DFD is no exception. The first thing to realize about DFDs is that they are “development” viewpoints like structure charts. That means they describe how the system is organized rather than how the various pieces carry out their function. The difference between a DFD and a structure chart is that a structure chart describes how functions relate to each other, whereas DFDs describe how data moves through the system. To illustrate the relationship between the various tools, consider the following decision tree.

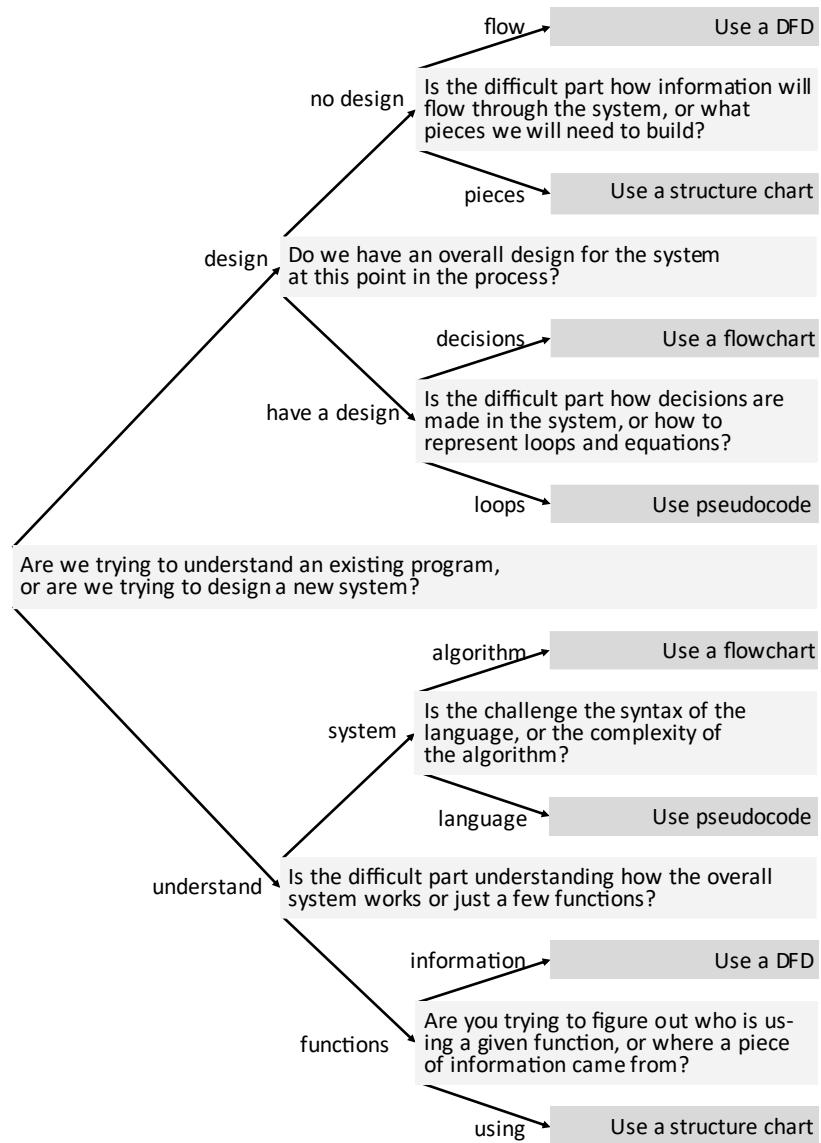


Figure 11.2:
Decision tree illustrating
how to select
a design tool

When designing a program, we first use the structure chart and the DFD to get a grasp of the overall system design. The algorithm design begins only after we completely understand what functions we will need and how data will move through the system.

When trying to fix a bug in an existing system, we use a DFD to understand where a given piece of data came from. We use a structure chart to understand who is calling a given function.

Data Flow Diagram Elements

As with the flowchart, pseudocode, and structure chart, the DFD is governed by several rules. The first and perhaps most important rule of the DFD is what problem it is designed to solve.

Rule 11.1: The DFD only represents how data moves through the system.

A DFD only represents how data enter, get transformed, get stored and leave the system

The DFD does not represent how data are altered or how processing happens. The flowchart and pseudocode are useful to that end. The DFD does not represent how the various parts of the system are organized. The structure chart handles that. Instead, the DFD only represents how data enters the system, where it gets transformed, where it gets stored, and how it eventually leaves the system. DFDs are created with five symbols, each governed by a collection of rules.

Symbol	Meaning
	Interactors are elements that create and consume data outside the system boundary. They can be users, external systems, or many other things. Note that if one considers a file or a database as external to the system, then they can be considered interactors.
	Flows are arrows representing the movement of data through the system. Flow arrows usually are labeled with the type of data they represent.
	Processors are things that act on data. They can transform data, set data, look at data, or move data between elements.
	Storage represents data at rest. These can be in a variable, a file, a database, or in a data structure.
	A subsystem is a portion of the system which is distinct in some way. It could be in a separate physical location or just be considered a different component.

Interactors

Interactors are elements that create and consume data. They are outside the system, providing input and accepting output. Examples of interactors include individual users, groups of people (such as administrators or sales representatives), organizations (such as a company), other systems (such as a web server), network entities (such as a Bluetooth device), or even a sensor (such as a camera or a GPS receiver). A key part of an interactor is that it is outside the system. Is a file “outside the system”? Well, it depends! If you are thinking only in terms of the application, then a file is outside the system and would be considered an interactor. If “the system” encompasses the computer, then the file would be a storage element within the system.

Interactors reside outside the system, creating and consuming data

Other common names for interactors are terminators, external entities, sources, and sinks. Interactors are represented as named rectangles in a DFD.

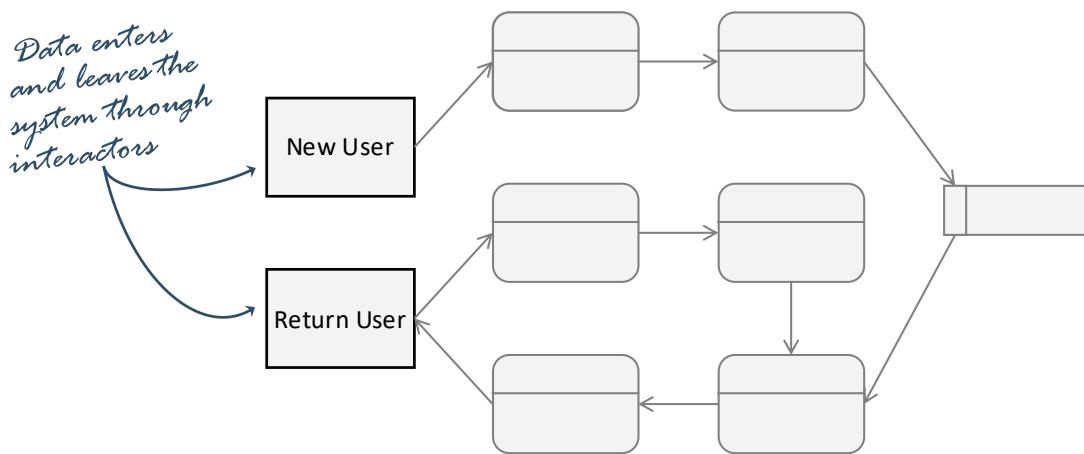


Figure 11.3:
Interactors

A few rules apply to interactors:

Rule 11.2: Interactors must be nouns

Interactors represent things, be they people or machines. They do not represent actions (verbs) or attributes (adjectives). If you are tempted to use a verb to describe an interactor, you probably want a processor instead.

Rule 11.3: Interactors are typically drawn on the edge of the DFD

To make it easier for people to understand your DFD, it is helpful to place interactors on the left edge or on the top of the DFD. If they are placed in the middle of the DFD or distributed randomly through the diagram, then they can be very hard to follow.

Rule 11.4: Interactors connect to processors through flow arrows

You cannot have one interactor connected directly to another or to a storage entity; they must go through a processor. Processors are verbs and one of their functions is to govern the movement of data.

Flow

Flows are arrows representing the movement of data through the system. In most applications it is necessary to describe the nature of the flow. This could be a parameter name, the format of the data being moved, or even a rough description of what is being moved.

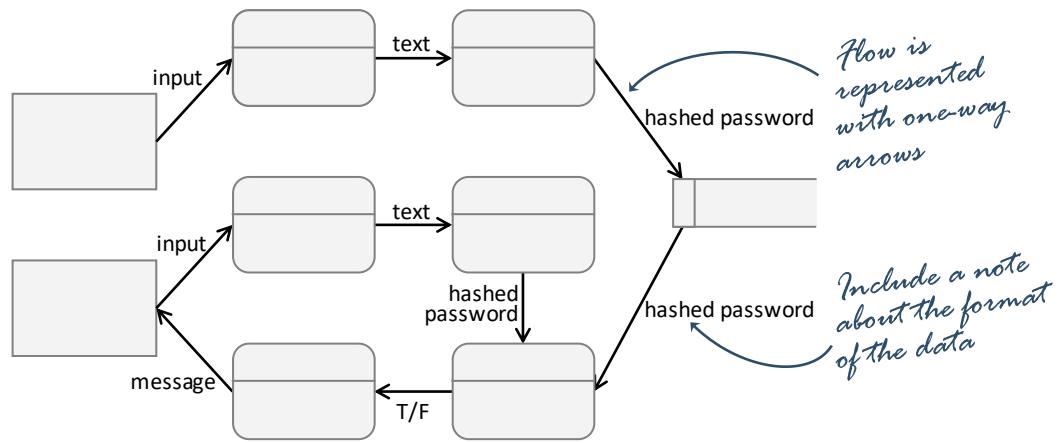


Figure 11.4:
Flow arrows

There are a few rules that apply to flow arrows:

Rule 11.5: There must be at least one processor connecting to each flow arrow

Two processors can be connected by flow. Data can leave a processor to enter an interactor or storage element. Data can leave an interactor or storage element to enter a processor. All these variations are OK. It is not possible for two interactors, two storage elements, or an interactor and a storage element to be connected by a flow arrow. There must be a processor mediating that interaction.

Rule 11.6: Flow arrows are one-way

If there is a two-way flow of data between elements in a DFD, then two flow arrows are utilized.

Rule 11.7: Flow arrows represent data movement

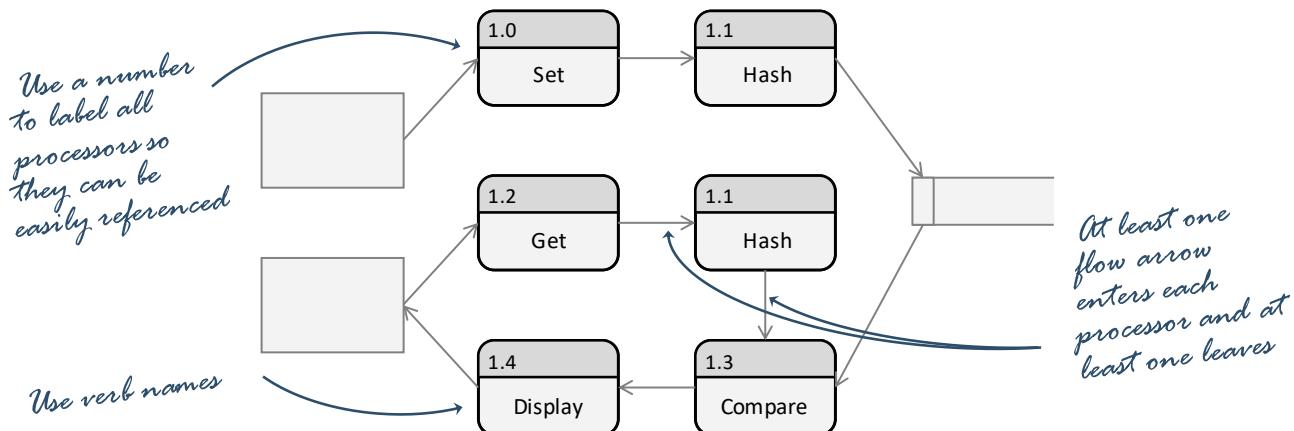
Flowcharts use arrows to represent movement of program control (specifically the movement of the instruction pointer). Structure charts use lines to represent that a function call is made. Class diagrams use arrows to represent inheritance and composition. DFDs, on the other hand, use arrows to represent the flow of data through the system. If you find yourself tempted to use a flow arrow for another purpose, you are probably using the wrong design tool.

Rule 11.8: Flow arrows are labeled

Unless by the usage or format of data associated with a flow arrow is painfully obvious, it is best to include a label. In many circumstances, the format of the data will enough (such as text, a stream, an integer, a grade, or a date/time). Other times, the use of the data is the important thing (such as username, input file stream, average grade, or creation date).

Processors

Processors are things that act on data. The scope and complexity of these actions can vary widely. For example, a processor could represent a simple mathematical expression converting feet to meters. It could represent a function which computes federal income tax from the user's income. It could even represent a host of complex actions such as generating a set of internet search results from a given search term. In each of these cases, it accepts at least one input element and produces at least one output result.



Fundamentally, processors perform one of four tasks:

Task	Description
Transform	Processors convert data from one format to another
Move	Processors move data from one part of the program to another
View	Processors look at data, typically for the purpose of making a decision
Set	Processors set data, usually from a computation with other data

Several rules apply to processors:

Rule 11.9: Processors have verb labels

Processors must have verb names because fundamentally they perform an action. If you are tempted to use a noun label, then it is probably a storage element or an interactor.

Rule 11.10: Processors are numbered

Processors have numerical labels in the top-left corner. This makes it easy to unambiguously reference them from other parts of the DFD or in other documentation. It also allows us to "zoom in." Please see the "Hierarchies" section in a few pages for more details.

Rule 11.11: At least one flow arrow must enter, and at least one must leave each processor

Data does not start or stop at a processor. It must come from somewhere and it must go somewhere. Note that you can have more than one inbound flow arrow (as is typically the case when a decision is made) and you can have more than one outbound arrow.

Figure 11.5:
Processors

Storage

Storage represents data at rest. These can be in a variable, a file, a database, or in a data structure. Other names for storage elements are data store, file, database, warehouse, and others. Note that storage items are within the system. If a file or database is considered to be outside the system, then it is called an interactor.

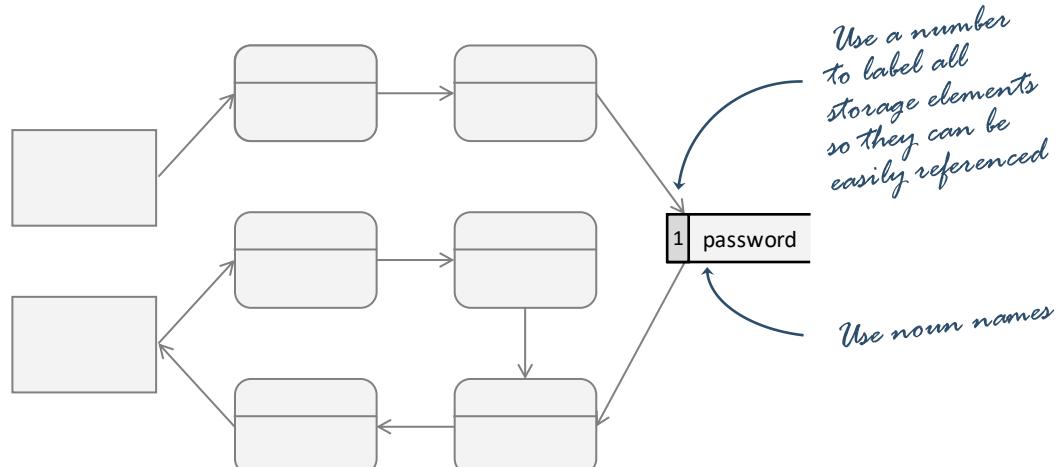


Figure 11.6:
Storage elements

There are a few rules associated with storage elements:

Rule 11.12: Storage elements are nouns

Storage elements refer to things: data at rest. Thus, they do not do anything; instead, they contain or represent something. In almost all cases, the labels associated with storage elements correspond to the actual variable name, file name, database table, or other program entity which it represents.

Rule 11.13: There should be an inbound and an outbound flow arrow for each storage element

This is not a hard-and-fast rule as the other DFD rules tend to be. In fact, there are some good reasons why a storage element might have only outbound flow arrows.

Normally a program will store data and then utilize those same data later in the program. A bug may exist, however, where a value is computed and then never used. DFDs facilitate finding such bugs.

In some cases, data are used but never set. In these cases, the data are hard-coded into the program and then presented to the user under certain circumstances. In most cases, we would not need to model this detail. However, if the program is designed to protect an asset (such as the password "rosebud" which is hard-coded into the program), then it would be worthwhile to include this detail in the DFD.

Rule 11.14: All flow arrows connecting storage elements must go through processors

If data are flowing into a storage element, they must come from a processor. If data are being retrieved from a storage element, then it is a processor which makes the request. You cannot have two storage elements connected by a flow arrow, and you cannot have an interactor directly connected to a storage element. It is the processor's job to mediate all interactions with storage elements.

System Boundaries

In large and complex systems, it is often helpful to include system boundaries in DFDs. For example, if data are to move from a client to a server, if the data are to load from a file into a program, or if the data are to travel from a phone to a car's speakers, then significant system boundaries exist. The symbol to use a system boundary like this is like that of processors.

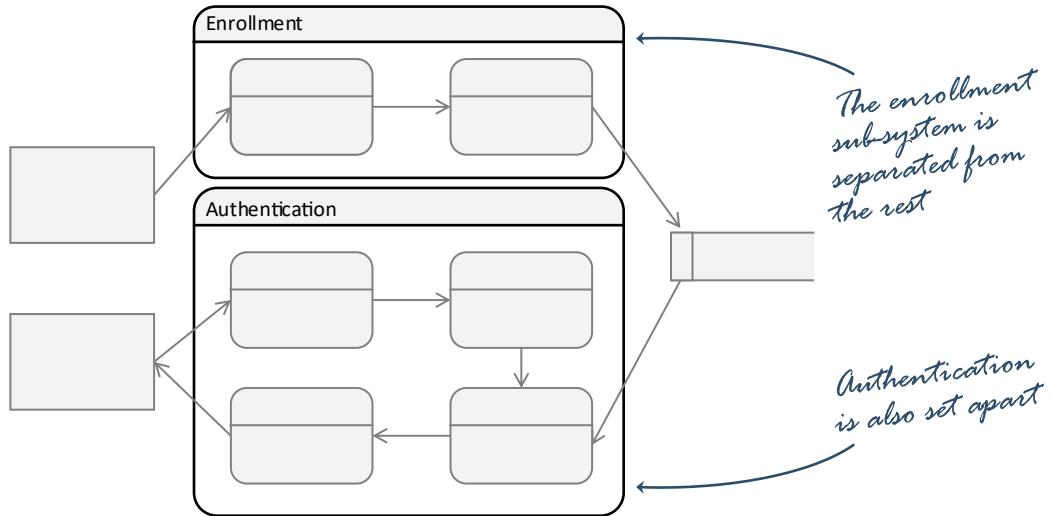


Figure 11.7:
System boundaries

The most common reasons why one would want to include system boundaries are the following:

1. There are classes in the design and the distinction between private and public is an important consideration in the system design.
2. The system involves moving data across distinct physical entities, such as more than one program or computer. It is very common to use system boundaries when networks are involved.
3. There are privacy or security concerns. Here, the system boundaries indicate varying levels of trust or access.

System boundaries can be nested (enclosing one subsystem with another) and can contain any number of elements. This is a useful tool for designing very large system. Another useful tool for representing and designing very large systems is a component diagram (please see Chapter 40 Tool: Component Diagram for more details).

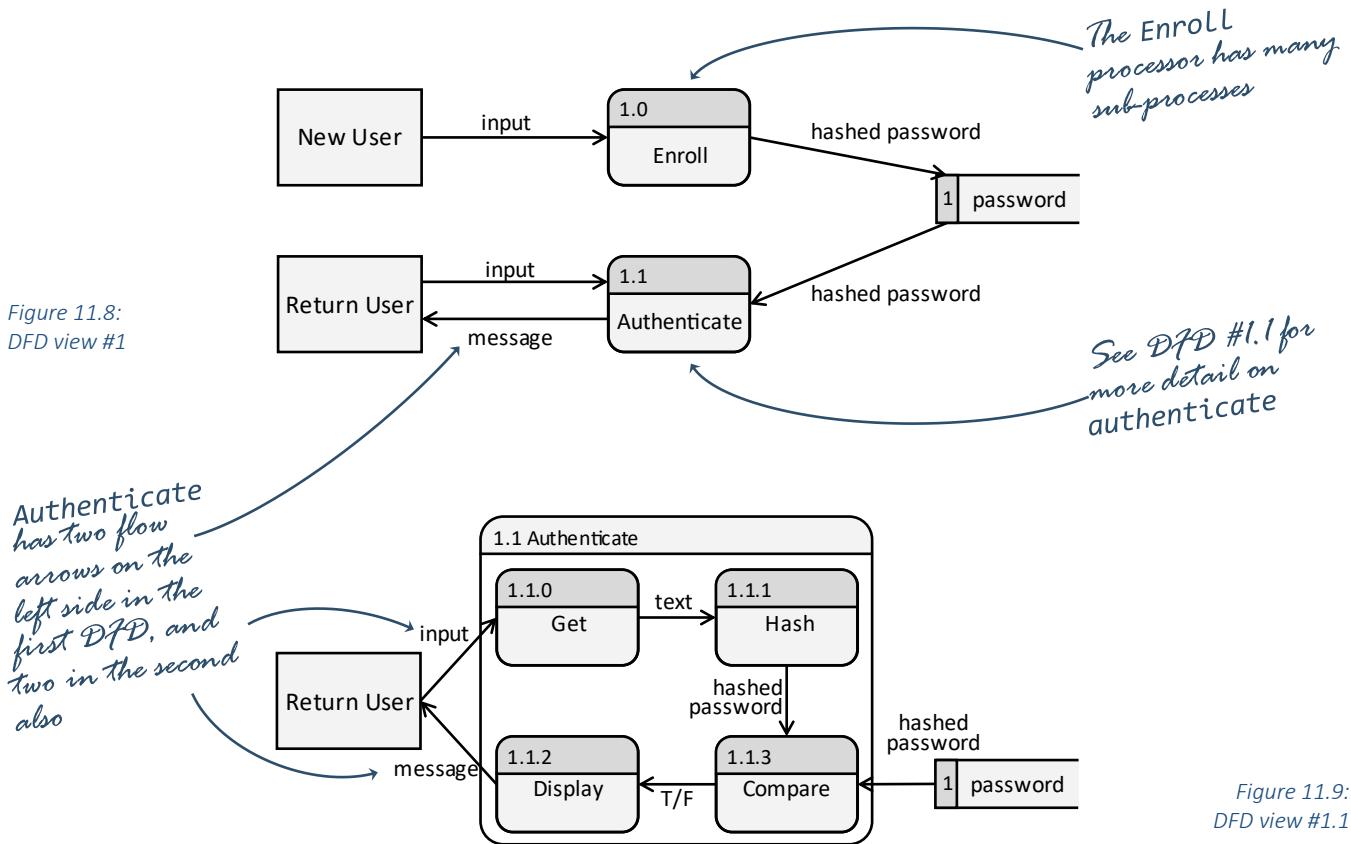
Hierarchies

It is difficult or impossible to model large and complex systems with a single DFD. There are simply too many elements to physically place them on one diagram, making the resulting DFD too complex for any human to make sense of it. As a result, most DFDs are hierarchies. They are done in the following way:

The first DFD, called “level 1” or “DFD #1,” presents a holistic system view. The dozen or so most important elements are present and every element represents a complex part of the system. Each element would then be labeled and another DFD would be created to “zoom in” on that aspect of the design.

To illustrate this process, the single DFD from before will be split into several:

Figure 11.8:
DFD view #1



DFD view #1 depicted in Figure 11.8 represents the high-level system view. Through this, we can understand the general flow of data through the system. If we want to learn more about the authentication process, we can go to DFD view #1.1 in Figure 11.9. Here, we can see the various transformations the data undergoes before a message is sent to the user.

It is important that view #1 and view #1.1 correlate. You will notice that processor 1.1 has two incoming flow arrows (hashed password and input) and one outgoing one (message). The same is true with view #1.1. If these do not agree, then at least one does not represent the true system implementation and the design will not work.

Large systems can have many, many, many sub views. It is not uncommon to have hundreds for a completely modeled system.

Variations

Flowcharts and structure charts have one dominant standard with several less-popular variations. Unfortunately, the same cannot be said for DFDs. There are three variations which are commonly employed today. The most common is the Chris Gane and Trish Sarson notation described in their 1977 book "Structured Systems Analysis: Tools and Techniques." This is the notation described in this chapter. Another popular notation was described by Edward Yourdon in his book "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design." The big differences are the circles for processors and open rectangles for storage elements.

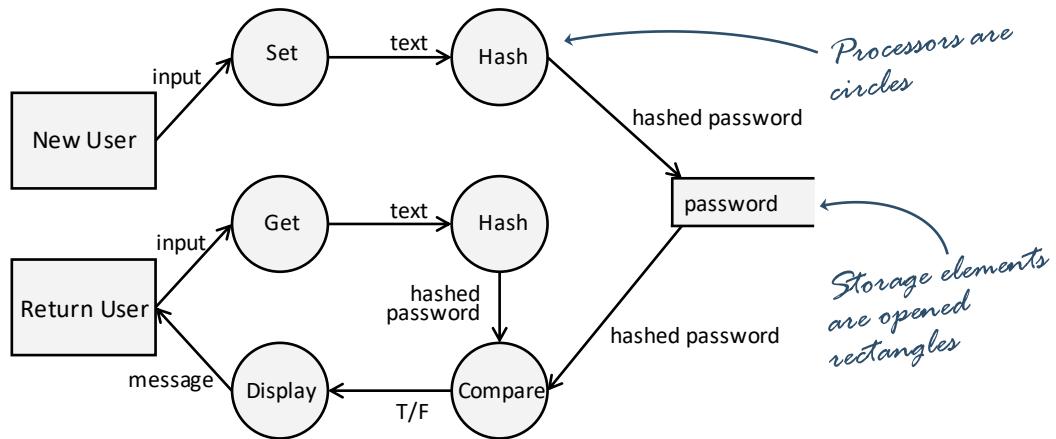


Figure 11.10:
Yourdon DFD notation

Tom DeMarco introduced another notation in his 1979 book "Structured Analysis and System Specification." This notation is commonly utilized by computer security teams to identify threats in a process known as threat modeling. It uses dotted lines for the program boundary (also known as a trust boundary) instead of a subsystem symbol described by Gane and Sarson. The storage symbol is also different: two horizontal lines.

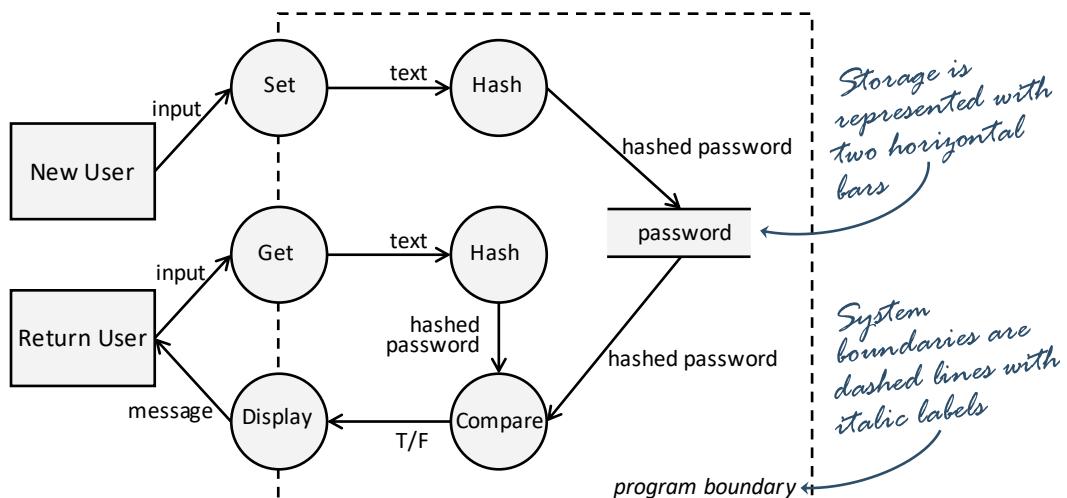


Figure 11.11:
DeMarco DFD notation

You will notice many similar elements between these three notations. The rules governing these variations are the same. Which will you choose to use? It is up to you! Most software engineers are familiar with all three variations.

Using DFDs

There are two primary uses for DFDs: to design new systems or to understand existing systems. Though we use the same tool for both, we create the tool and use it in entirely different ways.

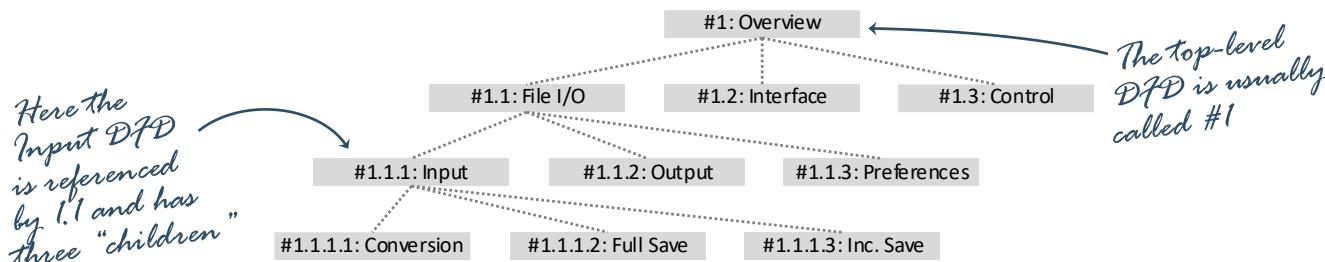
Designing with DFDs

To design a system using a DFD, you use the top-down strategy just as you do with a structure chart. The difference is what is defined as “top.” With a structure chart, one starts with `main()` and then works down to the functions that `main()` directly calls. When designing with DFDs, we don’t start with `main()` because functions themselves are not directly represented. Instead, we start with the interactors.

Outside-in design: start with the terminators and work towards a store

The first step in designing with a DFD is to list the system inputs and outputs. From here, we identify the level of processing that accepts the input and generates the output. The process is continued until we follow the data all the way to the data stores deep within the software. This approach is called “outside-in;” we start with the outside edges of the system and work towards the center.

DFDs are almost always created in hierarchies with many levels of details. The first view, normally called DFD #1, contains all the interactors of the system. In this high-level very abstract view: storage and processors are described only in general terms. DFD #1 serves as the blueprint for the entire system. From this starting point, subsequent views are identified. For example, the File I/O processor (labeled 1.1 in DFD #1) would be elaborated in DFD #1.1 to include a dozen or more processors and several storage elements.



With a small number of designers on a development team, it is not uncommon to work on the DFDs together as a group. Here, the team could follow a depth-first approach (Start with #1, then #1.1 and #1.1.1 and finally #1.1.1.1. Now backtrack and do #1.1.1.2) or a breadth-first approach (Start with #1, then do #1.1 and #1.2 and #1.3. Next, move on to the third level). Both approaches are equally effective.

The approach is usually quite different when the development team is large. Here, the entire team meets to discuss DFD #1 and the components of the system are delegated to the various teams. Each team would then work on their component independently. When a change is discovered that extends beyond the team’s component, a coordination meeting with the neighboring team would be needed. The larger the system redesign, the bigger the meeting. Of course, any change to DFD #1 would require the entire team.

Figure 11.12:
A hierarchy of DFDs

Understanding Existing Systems with DFDs

A DFD is a powerful tool to help a new programmer understand an existing system. Many organizations task new hires with documenting the existing system or subsystem by creating DFDs. This helps the new programmer grasp how everything works together.

To accomplish this, start with a given function or even a variable in the existing system. From that starting point, determine where the variable (or parameters if you are starting with a function) came from. With each step in the process, document your findings in a DFD. The process continues until the programmer identifies where the data entered the system or where the data left the system. This is called the “inside-out method.”

Inside-out analysis: start with a variable or data store in the system and work out towards a terminator

We often perform inside-out analysis in an ad hoc way when fixing a bug. We start with an assert, a program crash, or some other malfunction. From here, we trace the wayward variable back to the source. Things get complicated when the variable could come from multiple sources or undergoes multiple transformations. Soon our developer’s notebook becomes a jumbled mess.

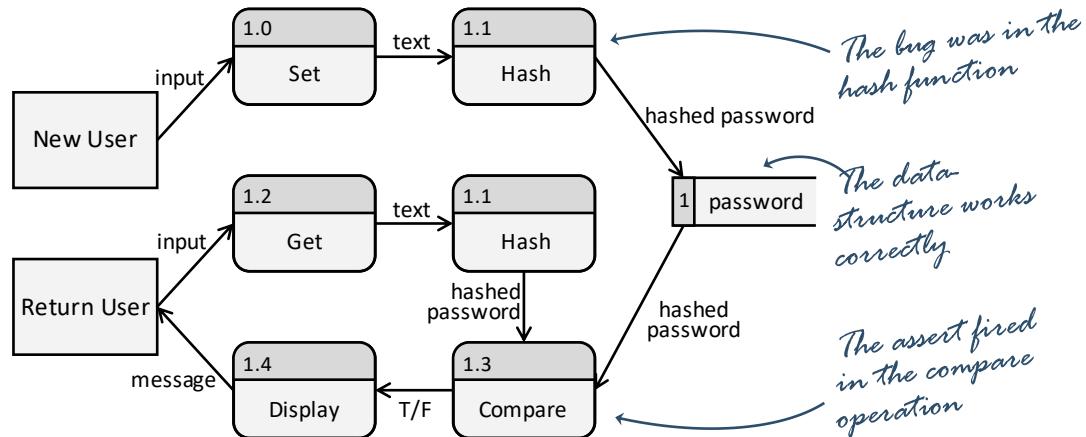


Figure 11.13:
Tracking down a bug
using a DFD

Back to our bug-fixing scenario, our programmer has discovered the problem in the compare operation represented in processor #1.3. Here, she can track where the incorrect value came from by following the flow arrows backwards. Using this process, she can discover that the problem originated in the hash function represented by processor #1.1. When trying to fix bugs like this, it is often necessary to track a piece of data back to the source. DFDs are designed to facilitate this process.

DFDs are developed from the outside-in for designing a system, and from the inside-out for discovering how an existing system works.

Examples

Examples 11.1: Modeling Existing Code

This first example will demonstrate how to create a DFD from existing code.

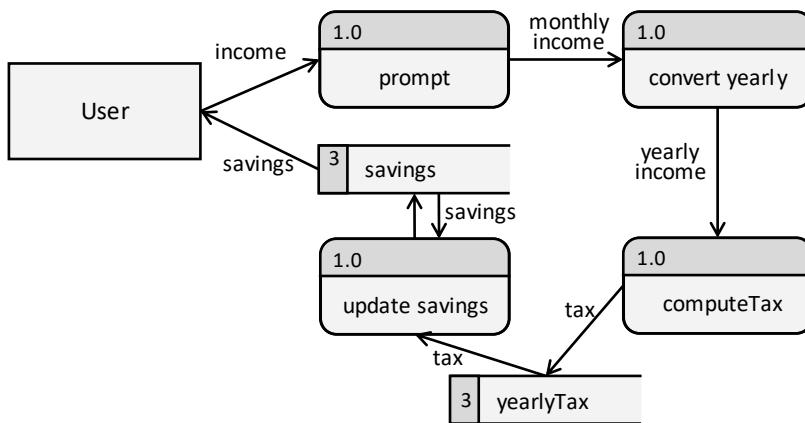
Problem

Create a DFD representing the following pseudocode.

Pseudocode
main() PROMPT for monthlyIncome yearlyIncome ← monthlyIncome x 12 yearlyTax ← computeTax(yearlyIncome) IF yearlyTax > \$10,000 PUT you might want to move to Monaco IF savings < yearlyTax PUT you are in debt ELSE savings ← savings - yearlyTax PUT savings ENDIF

Solution

Start with where data enters the system and follow it through the algorithm. In this case, data enter with `monthlyIncome`, get transformed to `yearlyIncome` and then to `yearlyTax`. Finally, the updated `savings` balance is displayed.



Examples 11.2: Designing a Program

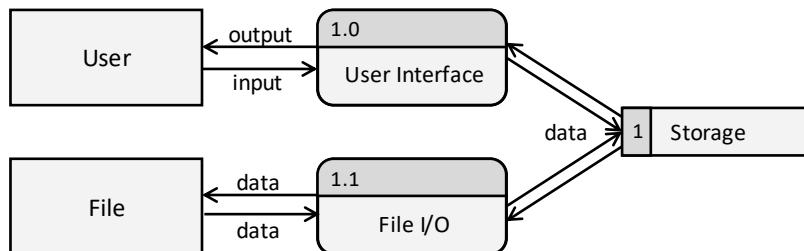
This second example will demonstrate how to design with a DFD.

Problem

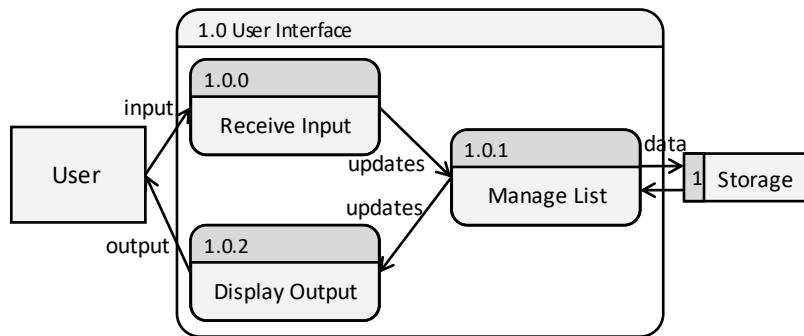
Create a DFD to represent how a to-do list mobile application will manage the user's data.

Solution

We will start with a simple system view containing the user interface, the file interface, the file data store, and the internal data store.



We will need to zoom in on how the user interface works. There are several components which need to be explored.



From here, we might want to explore the different types of input we expect from the user and how it will get transformed into useful data. This would require another DFD, probably labeled “1.0.0 Receive Input.” This process would continue until the system is sufficiently designed to begin writing code.

Exercises

Exercise 11.1: Identify the Symbol and Tool

For each of the following symbols, identify the design tool from which it comes and its use. The possible answers are: flowchart, structure chart, and DFD. Hint: a single symbol may belong to more than one tool.

Symbol	Tool and Usage

Exercise 11.2: Identify Parts of a DFD

Draw the symbols corresponding to the parts of a DFD.

Description	Symbol
A variable storing the date that a file was saved.	
Movement of an account balance from one part of the program to another.	
The user of the system.	
A function updating the bank account balance once interest is added.	
A file containing all the information related to a given user account.	
The bank's server.	

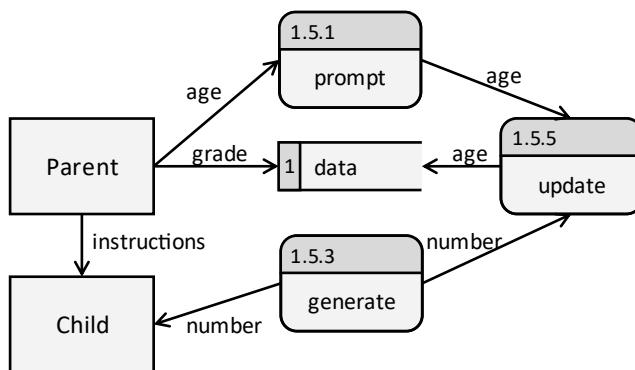
Exercises 11.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Processors have verb labels	
Interactors can connect to processors through flow arrows	
Storage elements can be connected with flow arrows	
DFDs represent how control moves through the system	
Flow arrows are always one-way	
Storage elements are nouns	
Interactors must be verbs	
Flow arrows never contain labels	
Processors are numbered	

Exercises 11.4: Errors

Identify all the errors in the following data flow diagram. If possible, list the rule number and description corresponding to each error.



Problems

Problem 11.1: DFD from Code

Create a DFD from the following code:

Python

```
balance = float(input("Your savings account balance: "))

# There is a minimum balance on this account
if balance < 200:
    fee = 20

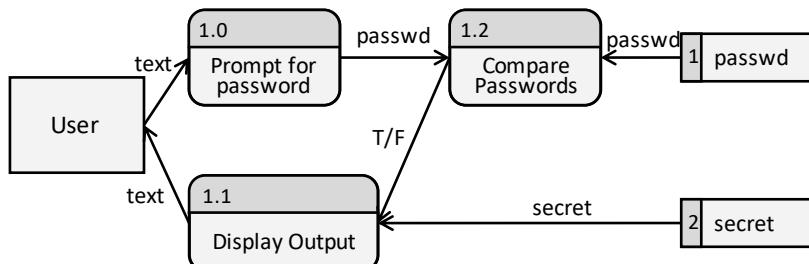
# Add some interest
interest = balance * 0.01

# User adjustments to the balance
deposit = float(input("Any deposits? "))
spending = float(input("What did you spend? "))

# Compute new balance and display it
new_balance = balance + interest + deposit - fee - spending
print 'The new balance is {0.2f}'.format(new_balance)
```

Problem 11.2: Code from DFD

In your programming language of choice, write the code corresponding to the following DFD:



Problem 11.4: DFD for Budget Program

Create a DFD for the following pseudocode.

Pseudocode

```
getIncome()
...
RETURN income

getExpenses()
...
RETURN expenses

computeTax(income)
...
RETURN tax

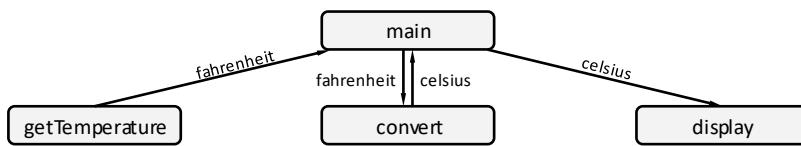
displayBudget(income, expenses, tax)
...

income = getIncome()
expenses = getExpenses()
tax = computeTax(income)
displayBudget(income, expenses, tax)
```

If you did Problem 10.1, then compare your structure chart.

Problem 11.4: DFD from Structure Chart

Create a DFD from the following structure chart:



Problem 11.5: Mad Lib DFD

Create a DFD matching the following draft of a pseudocode algorithm to play the game MadLib.

Pseudocode

```
askQuestion(question)
    PROMPT for question
    GET response
    RETURN response

readFile(fileName)
    READ word from file
    story[i] ← askQuestion(word)
    RETURN story

getFilename()
    PROMPT for filename
    GET filename
    RETURN fileName

playAgain()
    GET response
    RETURN true OR false

displayStory(story)
    PUT story

main()
    DO
        story ← readFile(getFilename())
        displayStory(story)
    WHILE playAgain()
```

Problem 11.6: Tic-Tac-Toe Design

Create a DFD for a program to satisfy the following scenario:

Tic-Tac-Toe, otherwise known as noughts and crosses, is a game played on a 3x3 grid where one player is represented with Xs and the other is represented with Os. The X player puts a mark in one of the nine grid locations. The O player then selects one of the eight remaining grid locations. The game continues until either one player gets three in a row or three diagonally.

This game will read a board from a file, allow the user to play the game, and save a partially-completed game back to a file. The user will be prompted for his or her selection, then the program will then choose the best response to the user's move. In other words, the user will play against the computer.

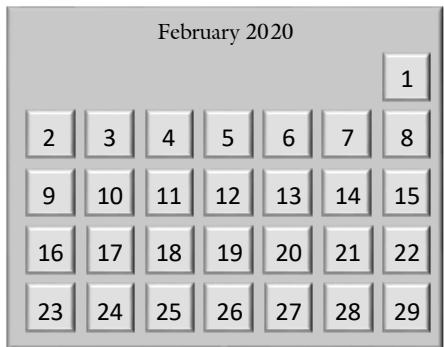
Hint: How will data enter the system? How will it be stored? What type of output will the system generate?

Challenges

Challenge 10.1: Calendar

Consider a program which will display a calendar on the screen. The input to the main function will be a month and year, both integer parameters. The output of the function will be a graphical depiction of a calendar.

A personal finance program needs to display a calendar in a dialog from which the user will select the due-date for a bill. You are tasked with writing this code.



Hint: Your program will need to determine the day of the week the first of the month falls on. This will be determined by counting the days since the 14th of September, 1752 (which was a Thursday) and dividing by 7. It will need to know the number of days in each month, taking leap years into account.

Create a DFD representing this entire program. If you did this challenge with structure charts (Challenge 10.1), compare and contrast the structure chart with the DFD. What types of details does one represent and the other does not?

Challenge 11.2: Asteroids

Consider the 1979 Atari game *Asteroids*. You may need to do some “research” to refresh yourself how this program works.

Create a DFD representing this entire program. If you did this challenge with structure charts (Challenge 10.2), compare and contrast the structure chart with the DFD. What types of details does one represent and the other does not?

Challenge 11.3: Sudoku

Sudoku is a numbers game played on a 9x9 grid. The object of the game is to fill in the 9x9 grid while honoring certain constraints. The constraints are: 1) There is no more than one instance of a given number on a given row. 2) There is no more than one instance of a given number on a given column. 3) There is no more than one instance of a given number on an inside square (the 3x3 squares embedded in the 9x9 grid). 4) Every square can consist of a single digit between 1 and 9 exclusively, or can be blank.

Create a DFD representing this entire program. If you did this challenge with structure charts (Challenge 10.3), compare and contrast the structure chart with the DFD. What types of details does one represent and the other does not?

Cohesion

Chapter 12

In order to know if we are using functions effectively in a program, we need to ask ourselves if each function is focused on a single task. We call this cohesion, the measure of the “internal strength” of a module. In other words, how much a function does one thing and one thing only? The four levels of cohesion are: strong, extraneous, partial, and weak.

A program is a collection of a very large number of individual statements. It is the combination of these statements that makes for working software – the magic of programming! As programs increased both in size and complexity, it became apparent that a way to organize these statements was needed. This was the genesis of functions (or modules or procedures). The term “modularization” is used to describe the process of subdividing a program into functions.

Modularization has several benefits: codebases are more understandable, contain fewer defects, are easier to alter and upgrade, and are more compact (contain less redundant code). Of course, the way programs are subdivided into functions makes a big difference in how understandable, bug-free, upgradable, and compact a program becomes. How, then, do we measure the quality of modularization?

Cohesion is a measurement of how well a unit of software represents one concept or performs one task

Shortly after functions were introduced into mainstream programming languages in the 1960s, Larry Constantine coined a term to capture the quality of our modularization efforts: cohesion. Cohesion is a measurement of how well a unit of software represents one concept or performs one task.

Being able to assess the cohesion level of a function and knowing how to optimize it represents a key skill to increasing the quality of modularization in our code.

Levels of Cohesion

Metrics like code size or execution time can be measured objectively and indeed even automatically. However, no program can tell us if two statements are related to a single task. This requires human judgement and intuition. Furthermore, we cannot obtain a linear metric to capture the cohesion measure of a function. Instead, there are a finite number of categories of cohesion which capture a broad spectrum of qualities and problems of functions. We call these categories the levels of cohesion.

Level	Description
Strong	The function performs one task completely.
Extraneous	There exists extra functionality.
Partial	The function does not complete a single task.
Weak	Both extraneous and partial.

Strong Cohesion

The strongest and most desirable level of cohesion is where all the code in a function is directed to one purpose. The formal definition of strong cohesion is this: all aspects of a function are directed to perform a single task, and the task is completely represented. There are two parts to this definition. The first is that the unit of software does nothing extra. Any extra code thrown into a function will forfeit its classification as strong cohesion. The second part of the definition is that the task or concept is completely represented. Anything that leaves part of the task undone or relies on the client to complete the work cannot be considered strong. Of course, every function should strive for strong cohesion. Observe that it does not matter how simple or complex the task is; it is strongly cohesive as long as only that task is being performed.

All aspects of a function are directed to perform a single task, and the task is completely represented

Consider the following function:

This function does one thing and one thing only:
Nothing extra, and nothing is missing

```
C#  
static double computePay(double hours, double wage)  
{  
    if (hours < 40.0)                      // regular pay  
    {  
        return hours * wage;  
    }  
    else                                     // overtime  
    {  
        return (wage * 40.0) +  
                (wage * 1.5 * (hours - 40.0));  
    }  
}
```

Figure 12.1:
Strong cohesion

Observe how `computePay()` does one thing and one thing only: it computes pay given an employee's hourly wage and number of hours worked. This task is completely accomplished; no other work is needed in order to finish this task.

It is always our goal to write functions and methods that exhibit strong cohesion regardless of the circumstance. This is true if the function performs a highly complex task taking hundreds of lines of code, and is true if the function performs a trivial task requiring only a single line of code. Cohesion is a metric independent of task complexity.

Note that occasion sometimes requires a function that exhibits a weaker form of cohesion. This is never the ideal and is always to be looked upon as a compromise. For example, consider the scenario when a product is nearing completion and is about to be sent to the customer. As is frequently the case, a defect is found during the last weeks of development. There are many ways in which this defect can be fixed, the most elegant of which requiring a new function or modification of an existing function utilized by many parts of the program. The cost of verifying either of these options is expensive, taking many hours to complete and incurring considerable technical risk. A third option is also available: modifying a different function which would weaken its cohesion but simplify validation. In a case like this, the development team will probably choose the third option because loss of cohesion is compensated by other factors.

Extraneous Cohesion

The first weak form of cohesion is extraneous: there exists something unnecessary in the unit of software. A political analogy would be a “rider” on a bill and a sports analogy would be a “bench warmer” on a team. The formal definition of extraneous cohesion is: at least one part of a function is not directed towards a single task. However, the principle task is completely represented. Streamlined software should have no extraneous functionality. An example of an extraneous compute-tax function would be one that correctly performs the calculation and then asks the user what to do with the refund. Any time the word “and” is used to completely describe a unit of software, it is probably extraneous cohesion. Consider the following function:

At least one part of a function is not directed towards a single task. However, the principal task is completely represented

```
C#  
static double computePay(double hours, double wage)  
{  
    if (hours > 60.0)          // display error message if worked  
    {                          // than the maximum amount of time  
        Console.Out.WriteLine("WARNING: Special permission ");  
        Console.Out.WriteLine("is required.\n");  
    }  
    if (hours < 40.0)          // regular pay  
    {  
        return hours * wage;  
    }  
    else                      // overtime  
    {  
        return (wage * 40.0) +  
               (wage * 1.5 * (hours - 40.0));  
    }  
}
```

Notice how this F statement and error message are not directly related to the task at hand

Figure 12.2:
Extraneous cohesion

This function completely accomplishes the task of computing the pay for a worker. Unfortunately, it also does something that is not directly related to the task at hand. In this case, it warns if too much work is reported. As a rule, a function designed to “compute” should not also “display.”

Any time a function has code that is not directly related to the task at hand, there is a good chance that the function has extraneous cohesion (or worse!). Fortunately, the fix is very easy: move the extra code to a more appropriate location.

Partial Cohesion

Another weak form of cohesion is partial, where a task is left incomplete. In other words, additional data or work needs to be stored or completed elsewhere for the concept or task to be completed. Note that partial is not below extraneous in the hierarchy but rather a peer. One does not improve a partial class by making it extraneous. Instead, one finishes the job it was designed to do. The formal definition of partial cohesion is this: all aspects of a function are directed to perform a single task, but the task is not completely represented by the function. Partial cohesion is particularly difficult for the client of a system because the onus is on the client to figure out how to finish the job. It is also difficult for the author of the software because, in order to thoroughly test the system, all possible implementations that complete the task need to be discovered. Any time a description of a system necessitates a detailed description of the context in which it is used, that system is a candidate for partial cohesion. Consider the following function:

All aspects of a function are directed to perform a single task, but the task is not completely represented by the function

The diagram shows a C# code block with two functions: computeOvertimePay() and computeNormalPay(). A handwritten note on the left says 'This function does not handle overtime at all' with an arrow pointing to the first function. A handwritten note on the right says 'This function only computes overtime pay' with an arrow pointing to the second function.

```
C#
// Determine an employee's pay
// WARNING: Call computeNormalPay() if hours is less than 40
static double computeOvertimePay(double hours, double wage)
{
    Debug.Assert(hours >= 40.0);
    return (wage * 40.0) + (wage * 1.5 * (hours - 40.0));
}

// Determine an employee's pay
// WARNING: Call computeOvertimePay() if hours is more than 40
static double computeNormalPay(double hours, double wage)
{
    Debug.Assert(hours <= 40.0);
    return hours * wage;
}
```

Figure 12.3:
Partial cohesion

Here, each of the two functions accomplishes part of the task at hand. Anyone using one of these functions will also have to use the second to get the job done. There are two ways to fix this problem and make the function(s) strongly cohesive: either combine the functionality into a single function or create a wrapper function that calls both components.

With both parts covered, this function makes the other two more cohesive

The diagram shows a C# code block for a computePay() function that handles both normal and overtime pay. A handwritten note on the right says 'With both parts covered, this function makes the other two more cohesive' with an arrow pointing to the brace at the end of the if-else block.

```
C#
// Determine an employee's pay
static double computePay(double hours, double wage)
{
    if (hours < 40.0)
    {
        return computeNormalPay(hours, wage);
    }
    else
    {
        return computeOvertimePay(hours, wage);
    }
}
```

Figure 12.4:
Strong cohesion

Weak Cohesion

At least one part of a function is not directed towards performing a single task. Additionally, the task is not completely represented by the function

The worst form of cohesion is weak. One should never design for weak cohesion; it is a state that is to be avoided. The formal definition of weak cohesion is this: At least one part of a function is not directed towards performing a single task. Additionally, the task is not completely represented by the function. In other words, weak cohesion is a combination of extraneous and partial. In theory, one should never come across weak cohesion. Alas, if only this were true! Consider the following function:

```
C#
// Determine an employee's pay
// This function also tabulates total pay
// WARNING: Call computeNormalPay() if hours is less than 40
static double computePay(double hours,
                        double wage,
                        ref double totalPayEarned)

    // error if we try to compute regular pay
    Debug.Assert(hours >= 40.0);

    // compute overtime pay
    double pay = (wage * 40.0) +
                (wage * 1.5 * (hours - 40.0));

    // tabulate total pay earned to be reported elsewhere
    totalPayEarned += pay;

    return pay;
}
```

*Partial:
Where is the code to compute non-overtime pay?*

*Extraneous:
What does this code have to do with computing pay?*

Figure 12.5:
Weak Cohesion

This function exhibits extraneous cohesion. We can tell first because the function comment block contains the word “also.” A subsequent inspection of the code will reveal the code to configure output for money. Since this function does not display anything, the code clearly does not belong here.

This function exhibits partial cohesion because it only produces correct output if the employee’s wage is not less than 40 hours. In this case, the program will display an error message on the screen and still produce erroneous output.

Since this function is both extraneous and partial, it can be classified as weakly cohesive. It appears that the programmer threw code together hoping it would work, rather than properly designing the function. On the surface, it might seem that the best approach from this point is to add the missing functionality and remove the extraneous parts. In practice, a better approach is to start the design process from scratch with cohesion in mind.

Variations

The concept of cohesion as we know it was first identified by Larry Constantine in 1968. He called this the “structured theory of programs,” though it was not until 1974 that the term “cohesion” was used to describe it. Originally cohesion was defined as “the measure of the interrelatedness of features located in the external interfaces of a class,” though it was later simplified to “a function should do one thing and one thing only.”

Classical cohesion is described on a seven-part taxonomy. This taxonomy was never meant to be a scale or metric of quality, but rather a way to classify or justify the reason why a function exists. The seven levels of cohesion are the following:

Level	Description
Coincidental	There is no meaningful relationship among the elements in a module.
Logical	Elements fall into some general category because they all do the same kind of thing.
Temporal	Some logical relationship exists and the elements are related by time. That is, the temporally bound elements are executed in the same time period.
Procedural	Performs multiple sequential functions, where the sequential relationship among all the functions is implied by the problem or application statement.
Communicational	Elements are related by a reference to the same set of input and/or output data.
Sequential	The output data (or results) from one processing element serve as input data for the next processing element.
Functional	All the elements are related to the performance of a single function.

Constantine’s levels of cohesion are commonly used today to evaluate modularization design quality, though they are often mistakenly applied as a scale. For example, sequential is deemed to be better than procedural, so one can strengthen the cohesion of a function by adding dependencies between statements. Generally, engineers seek to reduce dependencies rather than increase them.

Note that Constantine’s coincidental level of cohesion is very similar to the modern definition of weak cohesion. Similarly, Constantine’s functional level of cohesion is very similar to the modern definition of strong. There is no correlation between the middle five of Constantine’s levels of cohesion with the middle two of the modern levels.

Strengthening Functions

Assessing the level of cohesion of a function can be difficult. The software designer needs to understand the purpose of the function as well as the task of every line of code contained therein. As difficult as that is, it is harder still to design functions that are strongly cohesive.

Unfortunately, there is no single foolproof technique that we can follow which will always yield perfectly cohesive functions. A certain amount of perspective, brainpower, and judgement are required. That being said, all is not lost! There are a couple of techniques that we can follow which will increase the odds that our code is strongly cohesive.

Best Practice 12.1 Choose function names carefully

Recall the line from William Shakespeare's play Romeo and Juliet:

What's in a name? That which we call a rose by any other word would smell as sweet;

The Bard of Avon is saying, in effect, that the name we apply to something is really not that important. In the context of this play, Juliet is implying that Romeo's last name means nothing and they should be together regardless of their family feud. Unfortunately, this sentiment is counterproductive in software engineering.

Be intentional and systematic about the way you name a programming construct; it is vitally important

The name we bestow upon a variable, a function, a class, a file, or indeed any programming construct is vitally important. Not only does it allow us to find the construct amongst a sea of distractors, but it also informs us of its use. We should be very precise and descriptive in our name selection. We should also be very regular.

What does it mean to be regular? It means that if we are to have two similar variables in different functions, they probably should have the same name. It means that if we are to have two functions that perform similar tasks, they should have similar names.

It also means that the differences between similar constructs should be made clear in our name selection. For example, if I were to have three index variables in a single function, the similarities between the variables names as well as their differences should be represented in their names.

This brings us to the subject at hand: using the name selection process of a function to help us design more cohesive functions. Early in the design process, we should be very intentional about the names we choose for functions and variables. We should decide upon the name of a function long before we figure out what goes into it or even how we will go about writing the code. Too many programmers wait until the end of the design or coding processes to finalize the name of their functions. As a result, they never settle in their minds what the function is all about. This makes it very difficult to make the function cohesive.

Finalize the name of each function as early in the design process as possible. Make sure the name completely captures the function's intended purpose, and make sure the name is understandable. Then, as you write the code in the function, make sure every line of code contributes to this unified purpose and, when you are done, make sure the purpose is completely fulfilled.

iRow	Index into a row of a grid
iColumn	Index into a column of a grid
iPlayer	Index into an array of players

Comments, by definition, have no impact on compiled code. Despite this, they are an invaluable resource for making code more cohesive and for finding cohesion errors. This is because comments help programmers document their understanding of what code does. To see how this works, we need to recognize a unit of programming smaller than a function and larger than a statement: a block.

If one relates a programming language to a spoken language, it is easy to see the parallel between a sentence and a programming statement. Both are collections of words/keywords conforming to a fixed syntax designed to represent a simple idea. It is also easy to see the parallel between an essay and a function. One is a collection of sentences designed to convey the author's point and the second is a collection of statements designed to perform an action. Essays are frequently broken into paragraphs: units of writing related to a central theme, idea, or point that are smaller than the overall essay. These are usually set apart from one another with indentations, a new line, or spacing. The programmer's analogy to a paragraph is a block: a collection of related statements within a function or method designed to represent a task.

It is considered good programming style to set apart blocks in code with a blank line and a comment. This comment should, in just a few words, describe what the block is meant to accomplish. Reading all the block comments in a function should nicely summarize what the function is meant to accomplish. Consider the following code:

```

First block,  
the code  
dealing with the  
D: and M: prefix
Third block,  
dealing with  
decimal input
Second block,  
dealing with  
hexadecimal

```

```


    // Read a memory address from a text string. Formats include:
    // 0xe1 39 D:86 M:23
    function addressFromText(s)
    {
        var output = 0x00;

        // check for M: or D: prefix for special memory addresses
        if (s.charAt(0) == "M" && s.charAt(1) == ":") {
            s = s.substr(2, s.length); // no offset for M:
        } else if (s.charAt(0) == "D" && s.charAt(1) == ":") {
            s = s.substr(2, s.length);
            output += sizeMemory; // add display offset for D:
        }

        // the value is in hexadecimal. Convert it to numbers
        if (s.charAt(0) == "0" && s.charAt(1) == "x") {
            s = s.substr(2, s.length); // skip the 0x at the start
            output += parseInt(s, 16); // base 16
        }

        // the value is in decimal, convert it from text to numbers
        else {
            output += parseInt(s, 10); // base 10
        }
    }

    return output;
}


```

Figure 12.6:
Block comments

Each block should be cohesive, related to the block comment preceding it. Each function should be cohesive, being a summation of the block comments. Thus, block comments are a steppingstone for summarizing large functions and identifying statements or blocks which do not belong.

Examples

Examples 12.1: Partial Cohesion

This example will demonstrate how to recognize partial cohesion.

Problem

Identify the level of cohesion in the following function:

C++

```
// input: "Schmitt, John Jacob Jingleheimer"
// output: {"Schmitt", "John", "Jacob Jingleheimer"}
FullName parseFullNameFromText(const char * input)
{
    FullName name;

    // last name
    for (; *input != ','; input++)
        name.last += *input;

    // consume comma and space
    assert(input[0] == ',' && input[1] == ' ');
    input += 2;

    // first name
    for (; *input != ' '; input++)
        name.first += *input;

    // consume the space
    assert(*input == ' ');
    input++;

    // grab the rest of the text
    name.other = input;
    return name;
}
```

Solution

On the surface, this may appear to be strong cohesion. It does one thing and one thing only. We can see that the program only generates a name from the provided name, and it does that task completely. Or does it?

Notice that the function assumes that the input string is in a certain format. If the input string is empty, missing the comma or space, missing a first name, or even missing the middle name, then it will malfunction. In debug mode, it will assert. In the release mode, it will walk off the end of the string and crash. We therefore expect the caller to do work to prepare the input string for our function. Any time we rely on another part of the program to do part of our task, we are a good candidate for partial cohesion. Now, it is not guaranteed in this case. It might be that the name input function pre-sanitized the text, so it is always in the expected format. It is unlikely in this case, however. This is, therefore, partial cohesion.

Examples 12.2: Extraneous Cohesion

This example will demonstrate how to recognize extraneous cohesion.

Problem

Identify the level of cohesion in the following function:

Swift

```
// did element1 and element2 hit each other
func hitDetection(element1: inout Element,
                  element2: inout Element) -> Boolean {
    // advance our position through inertia
    element1.position = computePosition(element1.position,
                                         element1.velocity);

    // if the distance is less than the diameter, then hit!
    var distance = computeDistance(element1.getVelocity(),
                                    element2.getVelocity());
    if distance < element1.getSize() + element2.getSize() {
        element1.kill();
        element2.kill();
        return true;
    }

    return false;
}
```

Solution

The first step is to identify the purpose of the function. The comment at the beginning of the function is aligned with the name of the function: a simple hit-detection algorithm. Specifically, we wish to see if `element1` and `element2` hit each other in a video game.

Next, we will look at the block comments. The first block claims that the following statement relates to advancing the position of `element1` for inertia. The following statement seems to do exactly that, so the block has strong cohesion.

The next block determines if two elements hit each other. It does this by computing the distance between them and seeing if the result is less than the diameter of the two elements. One could argue that killing the two elements would weaken the cohesion level of the function, but perhaps that is how hit detection occurs in this program. Thus, the second block has strong cohesion.

The final step is to relate the blocks to the overall purpose of the function. The second block is related directly to the function purpose, but the first is not. Computing inertia for one of the two elements is extraneous.

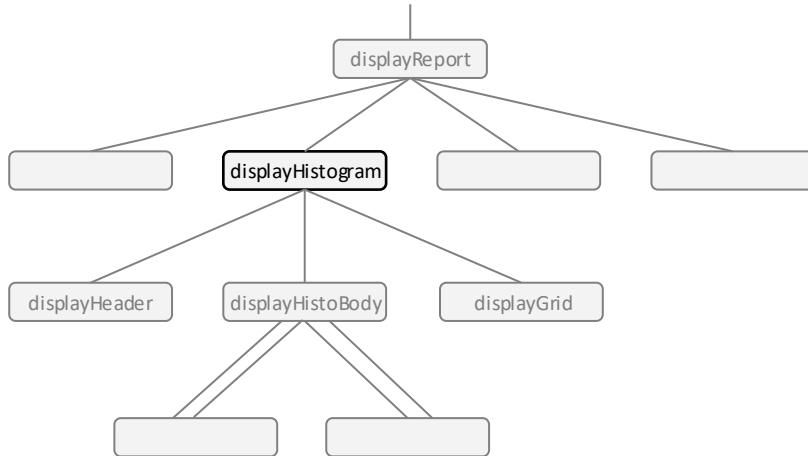
Since this function completely carries out hit detection, it cannot be partial, so it must be strong or extraneous. Since the function has extraneous code, it must be extraneous or weak. It therefore follows that it is extraneous.

Examples 12.3: Designing for Cohesion

This example will demonstrate how to design a program to maximize cohesion.

Problem

Optimize the cohesion for the following program. Focus on the `displayHistogram()` function:



Solution

From the above structure chart, we can see that `displayHistogram()` is called from `displayReport()` and calls three functions: `displayHeader()`, `displayHistoBody()`, and `displayGrid()`. We desire for `displayHistogram()` to exhibit strong cohesion. Since the function is not written yet, we cannot analyze the code to see whether it hits that mark. The criteria for strong cohesion are the following:

Strong Cohesion: All aspects of a function are directed to perform a single task, and the task is completely represented

First, does `displayHistogram()` do anything unrelated to what which its name suggests? The calling function suggests not; `displayReport()` apparently is asking `displayHistogram()` to display one special type of report. The children of `displayHistogram()` also appear to be related to the task at hand.

Second, does `displayHistogram()` completely carry out the intended task? There seems to be three components to displaying a histogram on the screen: draw a header, draw the grid underlying the data, and place the data itself on the chart. Since functions exist to perform all of these tasks, it seems that the task will be completely carried out.

By looking at these two aspects of the structure chart with respects to `displayHistogram()`, it is apparent that the function is set up to be cohesive. If we follow this process for all the functions in the structure chart, we should be able to tell whether the design will be cohesive. It is easier to fix cohesion problems at the structure chart level rather than in the source code.

Exercises

Exercise 12.1: Define the Level of Cohesion

From memory, recite the definition of each level of cohesion.

Level	Definition
Weak	
Extraneous	
Partial	
Strong	

Exercises 12.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Cohesion is a measure of how much code is in a function	
Weak cohesion maps closely to Constantine's coincidental cohesion	
Cohesion is a modularization metric	
Strong cohesion maps closely to Constantine's sequential cohesion	
Weak cohesion is the worst combination of partial and extraneous	

Exercise 12.3: Identify the Level of Cohesion

Identify the level of cohesion for the following function.

VB

```
Public Function IsLeapYear(ByVal year As Integer) As Boolean
{
    If year Mod 4 <> 0
        Return False

    ElseIf year Mod 100 <> 0
        Return True

    ElseIf Return year Mod 400 = 0
        Return True
    Else
        Return False
    End If
End Function
```

Exercise 12.4: Identify the Level of Cohesion

Identify the level of cohesion for the following function.

Kotlin

```
fun daysInYear(year: Int): Int {
    if (year < 0)
        println("You need to buy a new calendar!")

    if (isLeapYear(year)) {
        return 366
    } else {
        return 365
    }
}
```

Exercise 12.5: Identify the Level of Cohesion

Identify the level of cohesion for the following function.

Pseudocode

```
getIncome
    GET income

    PUT prompt for taxes paid
    GET taxesPaid

    oweTaxes ← computeTax(income) - taxesPaid
    RETURN oweTaxes
```

Problems

Problem 12.1: Cohesion of Pseudocode

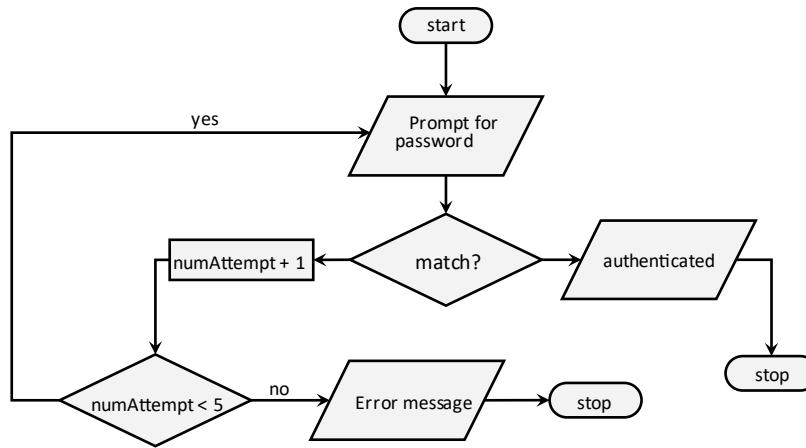
Identify the level of cohesion for the following pseudocode.

Pseudocode

```
computeTax (income)
    brackets ← readTaxTable(taxBracket.txt)
    FOR bracket ← brackets
        IF bracket.low ≤ income < bracket.high
            tax ← bracket.initial + bracket.rate ×
                  (income - bracket.low)
    RETURN tax
END
```

Problem 12.2: Cohesion of Flowchart

Identify the level of cohesion for the following flowchart.



Problem 12.3: Cohesion of Python Function

Identify the level of cohesion for the following Python function.

Python

```
def encode(input):
    keys  = [">", "<", "&", "\n", ","
    values = ["&gt;", "&lt;", "&amp;", "&13;", "&comma;"]
    dictionary = dict(zip(keys, values))

    for word in input:
        if word in dictionary.keys():
            output += dictionary[word] + " "
        else:
            output += word + " "
    return output
```

Problem 12.4: Cohesion of C++ Method

Identify the level of cohesion for the following C++ method.

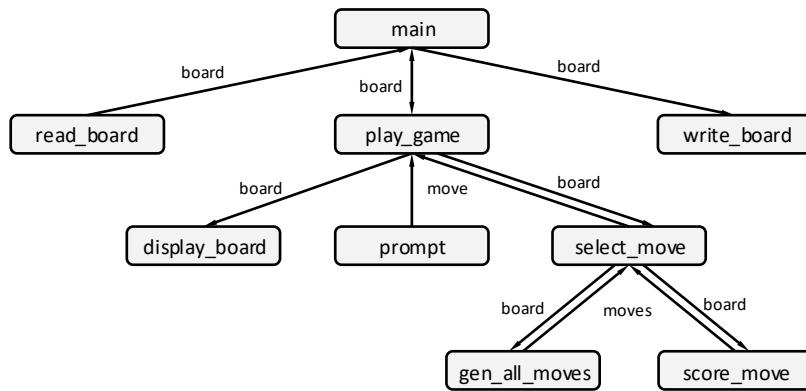
C++

```
// execute every job in the jobs list and tally up the scores
void Jobs::executeJobs(Assign & assign)
{
    // go through the list of all the jobs for this assignment
    // and execute them if they are ready
    vector<Job*>::iterator iter;
    for (iter = list.begin(); iter < list.end(); iter++)
        // if this job is active, then execute it
        if ((*iter)->isActive())
            (*iter)->execute(assign, *this, fVerbose);

    // tally up the scores and put them in the total
    assert(assign.getStatus() >= 0 &&
           assign.getStatus() <= stLast);
    scoresTotal[assign.getStatus()]++;
}
```

Problem 12.5: Tic-Tac-Toe Design

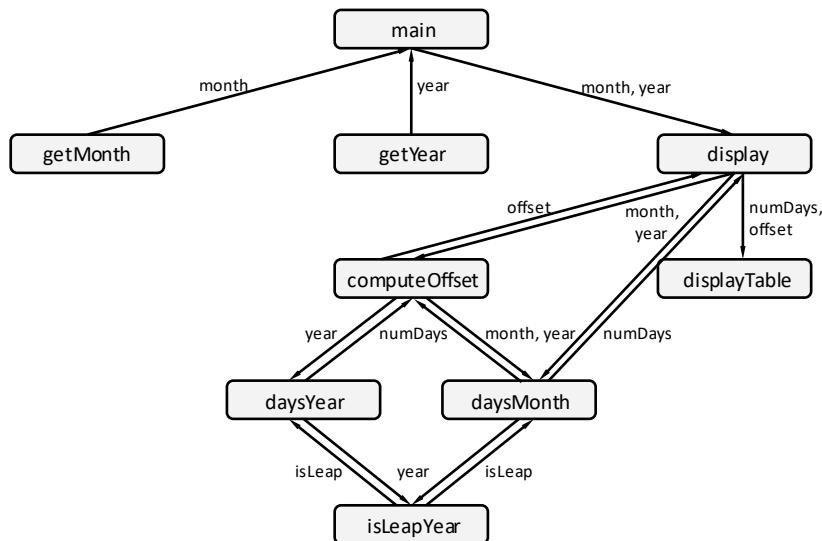
Consider the following structure chart for a program which displays a calendar on the screen. If you have done Problem 10.5, please use that structure chart.



Determine the level of cohesion for every function in the structure chart. If any function is less than strongly cohesive, refactor the design.

Problem 12.6: Calendar Design

Consider the following structure chart for a program which displays a calendar on the screen. If you have done Challenge 10.1, please use that structure chart.



Determine the level of cohesion for every function in the structure chart. If any function is less than strongly cohesive, refactor the design.

Challenges

Challenge 12.1: Sudoku

Consider the game Sudoku.

Sudoku is a numbers game played on a 9x9 grid. The object of the game is to fill in the 9x9 grid while honoring certain constraints. The constraints are: 1) There is no more than one instance of a given number on a given row. 2) There is no more than one instance of a given number on a given column. 3) There is no more than one instance of a given number on an inside square (the 3x3 squares embedded in the 9x9 grid). 4) Every square can consist of a single digit between 1 and 9 exclusively, or can be blank. This program will read a user-specified board from a file, allow the user to interact with the game while enforcing the rules, and then write an unfinished board to the file.

Create a structure chart for the game of Sudoku. If you have already created a structure chart for this in Challenge 10.1, please use that one. Determine the level of cohesion for every function in the structure chart. If any function is less than strongly cohesive, refactor the design.

Challenge 12.2: Yahtzee

Consider the dice game Yahtzee:

Yahtzee is a multi-round game of dice. The user throws five dice each turn and tallies points on a scorecard. Each category has a set of rules, describing the set of numbers on the dice necessary to complete the category. Every round requires a category to be filled, even if the current roll yields zero points for that category. The game is continued until every category has a score.

Create a structure chart to describe a program that allows a single user to play the game Yahtzee. For each function in the structure chart, identify the corresponding level of cohesion. If any function has less than strong cohesion, rework the design until the cohesion weakness is fixed.

Challenge 12.3: Card Game

Create a structure chart to describe a program that plays a card game of your choice. For each function in the structure chart, identify the corresponding level of cohesion. If any function has less than strong cohesion, rework the design until the cohesion weakness is fixed.

Coupling

Chapter 13

Coupling is a measurement of the complexity of the interface between units of software, representing how a function interacts with the rest of the program rather than what the function does.

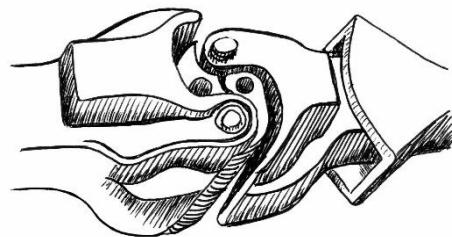
Coupling is a measurement of the complexity of the interface between units of software

program. Coupling is a measurement of the complexity of the interface between units of software. In other words, how much does the programmer need to know to successfully use a given function?

The name “coupling” is derived from the interface between railroad cars: the coupler. The design we use today was invented by Eli Janney in 1873. The coupler simplifies the interface between railroad cars to a single, well-defined connection. The train engineer doesn’t need to worry about the design of the cars or what they haul; they will all connect to his train if they use the standard coupler. This metaphor applies also to software. Ideally our functions will be as easy to connect to a program as a railcar is to a train.

There are two metrics of function quality: what goes on inside a function and how functions connect. Cohesion is the first metric, describing the purpose of a function and the code contained therein. The second metric describes how the function interacts with the rest of the

Figure 13.1:
Railcar coupler



Levels of Coupling

Coupling is difficult to measure objectively with counting metrics (such as the number and type of parameters passed between function) because counting metrics miss a key aspect of coupling’s definition: complexity. Any valid coupling metric needs to take into account the difficulty for a programmer in understanding an interface, how difficult the interface is to use, and how much the programmer needs to know in order to use the function the way it is intended. Based on this observation, seven levels of coupling have been identified.

Level	Description
Trivial	No information interchange exists.
Encapsulated	Parameters are convenient and always valid.
Simple	Parameters are easy to select and interpret.
Complex	At least one parameter is difficult.
Document	A rich language is utilized.
Interactive	Nontrivial dialog.
Superfluous	Any unnecessary parameter.

Trivial Coupling

Trivial is the weakest and best form of coupling. Here, the client of a unit of software needs to provide no information and receives no information from another unit of software.

There is no information interchange between units of software

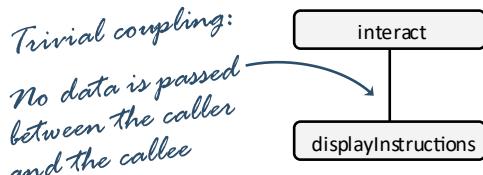


Figure 13.2:
Trivial coupling

Trivial coupling is defined as when there is no information interchange between units of software. In other words, one unit may instantiate, call, or activate another, but no information is passed. Similarly, no information can be gleaned from the timing of the function call. An example would be a function with no return value and no parameters. We represent trivial coupling with a structure chart with a simple line between the caller and callee, no arrows and no parameters passed.

A common question that is asked about trivial coupling is: can a function do any useful work if there is no input and no output? There are several ways to answer this question. First, trivial coupling refers to parameters passed between functions, not to input and output. A function can take user input and display results back to the user but still pass no parameters. Second, many common operations require no system interaction. Examples include pausing, waiting for an event to happen, and displaying instructions. Perhaps the third answer is the most relevant here. It is not appropriate for most functions to reach the trivial level of coupling. Unlike the cohesion metric where all functions should strive to be strongly cohesive, it is simply not possible for the vast majority of all functions to have trivial coupling.

It is not possible for the vast majority of all functions to have trivial coupling

Another common question pertains to object-oriented programming: does a class method taking no input and having no return parameter classify as trivial coupling? This question is complex. First, note that almost all methods have access to the member variables of the class (with the exception of static methods). Programming languages accomplish this by passing a hidden parameter to member functions. This hidden parameter is called `this` or `self` in various languages. Due to the presence of this hidden parameter, are all methods excluded from trivial coupling? The answer boils down to how the method interacts with the member variables. If a method utilizes a member variable, then it is the same thing as if the member variable was passed as a parameter. The only difference is that the caller is not responsible for providing the input parameters or interpreting the results. In this case, the method would be considered encapsulated. If, on the other hand, the method does not utilize any of the member variables, then the method is considered trivially coupled.

If a class method utilizes a member variable, then it is as if that member variable is passed as a parameter

A final question is: how does the use of global variables impact coupling? The same rules apply here as they do for class methods. If a function looks at a global variable or sets a global variable, then it is the same thing as being passed a parameter. However, if a global variable exists in the program and the function does not utilize it, then it is as if the global variable does not exist. In other words, creating a global variable does not automatically degrade the level of coupling of every function in the program.

Encapsulated Coupling

All the information exchanged between units of software is in a convenient form and is guaranteed to be in a valid state

Encapsulated is a very weak form of coupling defined by all the parameters being in a trusted and accessible state. In other words, there are logical checks in place to ensure that no invalid data are sent between modules. This level makes no reference to the degree of complexity of the data nor the number of data items passed between modules. A function exhibits encapsulated coupling when all the information exchanged between units of software is in a convenient form and is guaranteed to be in a valid state.

To see the value of encapsulated parameters, consider how to represent Boolean data in the C programming language. Lacking a native Boolean data type, programmers use integers where 0 is taken to mean *false* and any other value is taken to mean *true*. This makes it possible to have two *true* values which are not equal. For example, `value1` could be 1 and `value2` could be `0xffffffff`, and both would be considered *true*, but they would fail the equivalence test. If the programmer is not aware of this implementation detail, then a bug may exist that is difficult to find. This function has simple coupling.

C++ solves this problem with a native Boolean data type, thereby relieving the user from having to deal with unequal Boolean values. The compiler ensures that the value of a `bool` is either `true` or `false`, corresponding to 1 or 0. One could set `value1` to `0xffffffff` but it would contain 1 in memory. This function would have encapsulated coupling because the client would not have to worry about sending invalid data and the function itself would not need to validate the client's parameters.

Another example of a data type facilitating encapsulated coupling is the `Date` type in VB. It is much easier for the caller and callee when a `Date` object is passed between functions rather than forcing both to interpret a date string or some other format.

Figure 13.5:
Advantage of an
encapsulated Date

VB

```
Dim dateBasic As Date = #5/1/1964 4:00 AM#
Dim dateVB    As Date = #1/1/1991#
displayTwoDates(dateBasic, dateVB)
```

The VB system
natively supports
date objects

There are several ways in which a parameter can be said to be validated and accessible. Boolean data are always validated because the value is either *true* or *false*. Enumerations are also validated for most languages because the compiler ensures that only enumerated values are allowed. With modern languages and modern programs, the most common way to achieve the validated status is to use a class whose methods contain checks to guarantee data validity. Just about any interface can be simplified with the introduction of a class which handles data representation and validation.

Figure 13.3:
Simple coupling because
data validation is needed

C

```
int isEqual(int value1, int value2)
{
    assert(value1 == 0 || value1 == 1);
    assert(value2 == 0 || value2 == 1);
    return (value1 == value2);
}
```

Figure 13.4:
Encapsulated coupling

C++

```
bool isEqual(bool value1, bool value2)
{
    return (value1 == value2);
}
```

Simple Coupling

All the information exchanged between units of software is easy to select, interpret, and validate

Another weak form of coupling is simple, meaning that data can be selected, interpreted, and validated easily. Parameters consisting of built-in data types such as integers or characters are often simple, assuming that the use of the parameter is easily specified. Simple coupling is defined as all the information exchanged between units of software being easy to select, interpret, and validate.

The definition of simple coupling makes no reference to the number of parameters passed between functions. There is a caveat to this rule. If a parameter is unnecessarily passed between functions, then the coupling level is superfluous.



Figure 13.6:
Structure chart of
simply coupled functions

An example of simple coupling for a video game would be a `displayNumberOfLives()` function. This function would allow the logic of the game to decrement the number of lives if the player made a mistake or increment the number of lives if a bonus was achieved. It is trivial to select, interpret, and validate this `lives` parameter.

Figure 13.7:
Simple coupling

```
Swift
func displayNumberOfLives(lives: Int) {
    if lives >= 0 && lives < 100 {
        print("You have \(lives) lives");
    }
}
```

Easy to validate,
but still needs to
be done

Figure 13.8:
Encapsulated coupling

Of course, the design could be improved. If a class was used to represent the number of lives, then it would be impossible to mistakenly pass the value -12 or to leave the number uninitialized. This change would move the coupling level to encapsulated.

```
Swift
func displayNumberOfLives(lives: Lives) {
    print("You have \(lives.num) lives");
}
```

The class handles
validation for us

Finally, one could move the method closer to trivial if we had a `decreaseNumberOfLives()` function instead, absolving the caller from having to keep track of the number of lives.

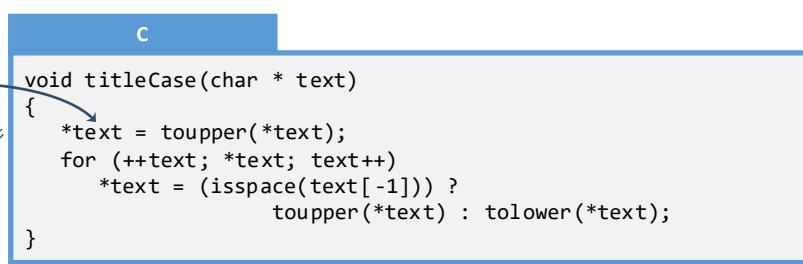
Complex Coupling

A tight or bad degree of coupling is complex. Here, the participants need to know a great deal about the parameters in order to make a connection. The greater knowledge the participants need to have and the more work the participants need to accomplish to make sure the connection is done correctly, the more complex the level of coupling. The definition of complex coupling is that at least one piece of information is nontrivial to create, validate, or interpret. While it might be worthwhile to enumerate sublevels of complex coupling, it is enough to say that all are bad and should be avoided.

At least one piece of information is nontrivial to create, validate, or interpret

information is nontrivial to create, validate, or interpret. While it might be worthwhile to enumerate sublevels of complex coupling, it is enough to say that all are bad and should be avoided.

One example of complex coupling would be a C string. Here, the participants need to be aware of the set of valid characters (are unprintable characters valid?), the inclusion of a null character to indicate the end of the string, and other limitations imposed by the string's usage. While all of these are easily managed by an experienced programmer, the onus is on the programmer to know these things and perform the checks to make a connection.

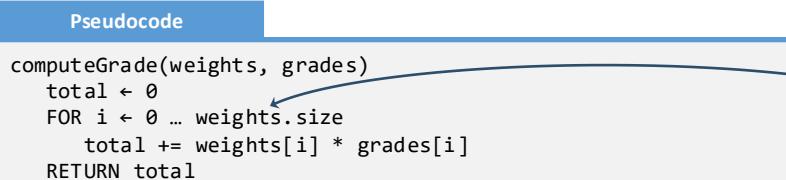


```
C
void titleCase(char * text)
{
    *text = toupper(*text);
    for (++text; *text; text++)
        *text = (isspace(text[-1])) ?
            toupper(*text) : tolower(*text);
}
```

Figure 13.9:
Complex coupling

In a second example, consider a function that computes the total grade for a student given the grades and weights for various categories. If the weights do not add up to 1.0 or if the categories are not in the expected range, then we have a bug. For this to function correctly, the onus is on the client to provide valid data and the function needs to perform extensive data validation. This is complex coupling.

Figure 13.10:
Complex coupling



```
Pseudocode
computeGrade(weights, grades)
    total ← 0
    FOR i ← 0 ... weights.size
        total += weights[i] * grades[i]
    RETURN total
```

The weights need to add up to 1.0 or this will not work

There is a certain temptation for you the experienced programmer to use complex coupling unnecessarily. This happens when you select a data structure or other data type that you have just invented or is very familiar. It is easy to say, "Well, anyone can figure that out," when you have just mastered it! The problem is that you now demand that users of your function must achieve the same degree of mastery in order to use your code. In other words, don't ask yourself, "Can another programmer figure this out?" Instead, ask yourself, "Why am I making another programmer figure this out?"

The solution to complex coupling is encapsulation. Hide the complexity of the parameter in a class; provide instead safe and convenient interfaces.

Document Coupling

Another tight level of cohesion is document. Here, data conforming to some language are passed from a producer to a consumer. Note that the producer could be the caller or the callee, depending on the direction of information flow. The important thing to note is that both the producer and the consumer need to fully understand the intricacies of the shared language. The definition of document coupling is that at least one piece of information contains a rich language including syntactic and/or semantic rules.

At least one piece of information contains a rich language including syntactic and/or semantic rules

Perhaps document coupling is best explained by example. Consider a program executing a shell script. Here, the producer is the program and the consumer is the operating system (OS) interpreter. The producer needs to create a document (the shell command in this case) that is then interpreted by the OS. Because any valid shell command could be sent, the coupling is extremely complicated. Furthermore, any change to the command language would require substantial modification to both the producer and the consumer.

The diagram shows a Python script within a blue-bordered box. The code imports subprocess and uses it to run a grep command on files matching a pattern involving a variable username. It then reads the output and prints each line. Two arrows point from handwritten text to specific parts of the code: one from 'There is a rich syntax for shell commands' to the subprocess module, and another from 'The function check_output() accepts a rich language' to the check_output() call.

```
Python
import subprocess

command = "grep " + username + " *.txt"
output = subprocess.check_output(command)
for line in output.stdout.readlines():
    print line
```

Figure 13.11:
Document coupling

Sometimes document coupling can be subtle and difficult to identify. For example, consider a function that interprets error codes so a user-friendly error message can be displayed on the screen. This function would take only an integer as a parameter. How complex could an integer be? As it turns out, rather complex. In this case, the integer is broken into two fields, being the major code and the minor code. Each of these components then maps to various strings. Since both the producer of the code and the function that interprets the code (the consumer) need to completely understand the code syntax for the interchange to function properly, this would qualify as document coupling.

The diagram shows a Python function lookupCode() within a blue-bordered box. It takes an integer code as input and returns a string error message. The code uses bitwise operations to separate the code into status and error fields, then looks up each in dictionaries statusMapping and errorMapping. A handwritten note on the right says 'Both the caller and the callee need to know all of the codes'. An arrow points from this note to the statusMapping dictionary.

```
Python
def lookupCode(code) :
    error  = code & 0xffff0000
    status = code & 0x0000ffff
    statusMapping = { 100 : "Continue",
                     101 : "Switching Protocols",
                     200 : "OK",
                     202 : "Accepted",
                     204 : "No Content",
                     404 : "Not found",
                     ...
                     500 : "Internal Server Error" }

    return statusMapping.get(status) + errorMapping.get(error)
```

Figure 13.12:
Document coupling using
only an integer

Interactive Coupling

Interactive coupling involves an ongoing communication between components, not a one-time message event. Interactive coupling is usually characterized by a session where conversations begin, the participants react to each other, and conversations are terminated. There are usually syntactic and semantic rules that need to be followed to correctly maintain the conversation. The definition of interactive coupling is that there exists a communication avenue between units of software involving nontrivial dialogs, sessions, or interactions.

There exists a communication avenue between units of software involving dialogs, sessions, or interactions

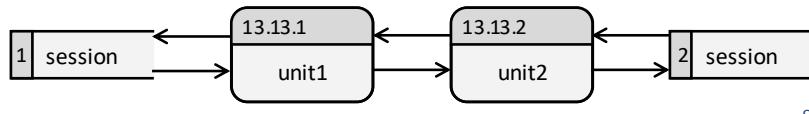


Figure 13.13:
Data flow diagram
showing interactive coupling

An example of interactive coupling is a socket connection where two parties send messages and confirm receipt. For this to work, multiple messages must be sent and results carefully checked.

There are
multiple steps
in the process
of setting up a
Berkeley socket

```
Perl
# initiate the socket
socket(SOCKET, PF_INET, SOCK_STREAM, getprotobynumber('tcp'));
setsockopt(SOCKET, SOL_SOCKET, SO_REUSEADDR, 1);

# connect to port 8281
bind(SOCKET, pack_sockaddr_in(8281, inet_aton("localhost")));
listen(SOCKET, 5);

# accepting a connection, send a message, and close it
my $client;
while ($client = accept(NEW_SOCKET, SOCKET)) {
    my $name = gethostbyaddr($client_addr, AF_INET );
    print NEW_SOCKET $message;
    close NEW_SOCKET;
}
```

Figure 13.14:
Interactive coupling

In most cases, interactive coupling also involves document coupling. The messages being passed between entities often involve complex syntax and semantics. Common examples of this are the TCP/IP handshake and the SSL connection.

While networking examples are certainly easy to come by, there are a host of others as well. Creating a graphics context to perform on-screen drawing is such an example.

Superfluous Coupling

The tightest and therefore worst level of coupling is superfluous, defined as at least one piece of data or information being passed between units of software unnecessarily. This is the only level of coupling that refers to the amount of data passed between modules. Data passing between modules is only bad if that data are not necessary. There are two important parts to this definition: what is unnecessary and what is data/information passing. The unnecessary component is completely domain specific. For example, if a module has read/write access to an asset when read-only is required, then the write aspect is unnecessary and the coupling can be classified as superfluous.

At least one piece of data or information is passed between units of software unnecessarily

The “passing” component of the definition is far more complex. Consider the range of scope for a full-featured high-level language (from small to large) in the table below. There are legitimate uses for each of these scope levels in many applications. However, if a function, class, component, or system utilizes a scope larger than is necessary, then superfluous coupling exists. The levels of scope are the following:

Scope Level	Accessible to ...
Block	... only a small number of statements
Local Variable	... all statements in a single function
One-way Parameter	... two functions, writable by only one of them
Two-way Parameter	... two functions, writable by either of them
Function Static	... all instances of a function
Private Member	... all methods in a single classes
Inherited Member	... all methods in multiple related classes
Public Member	... any who have access to an object
Class Static	... all objects of the class
Global	... the entire runtime environment
File System	... any program on the system
Protected Network	... all entities on a private network.
Global Network	... the world!

The degree of superfluity depends on the number of scope levels beyond that which is necessary that was utilized in a given application. The most common causes for superfluous coupling are the following:

- **Unnecessary Parameter:** Passing a parameter into a function or returning a value from a function when none is needed. This often happens when the parameter was necessary before but is no longer needed. Here, we have “one-way parameter” scope when “local variable” scope would be more appropriate.
- **Unnecessary Two-way Parameter:** This happens when an input parameter is made by-reference for performance reasons (so a copy does not need to be made). Thus, the scope is “two-way parameter” (otherwise known as by-reference) when “one-way parameter” (otherwise known as by-value) would be more appropriate. The fix is to make it one-way by adding the constant modifier.
- **Unnecessary Private Member:** A member variable should only be used if it is part of the state of the class. Sometimes programmers use a “private member” variable when “local” would be more appropriate.
- **Unnecessary Global Variable:** There are very few reasons for making a variable global. Think twice before using one; the result is almost certainly superfluous coupling.

Variations

Larry Constantine, the first to identify the concept of cohesion, was also the first to identify the concept of coupling. He defined coupling as “the number and type of inter-modular interconnections” and later as “the strength of interconnections.” The levels of coupling defined by Constantine and later by Myers were the following:

Level	Description
Common	A group of modules that reference a global data structure
External	A group of modules that are not content, or common-coupled and reference a homogeneous global data item
Control	One module explicitly controls the logic of the other
Stamp	Two functions refer to the same non-global data structure
Data	The modules directly communicate with one another and all interface data between the modules are homogeneous data items
No Direct	A function takes no parameters and returns no data

Unlike the original levels of cohesion, Constantine’s levels of coupling are designed to be a quality metric. The higher levels of coupling (Common, External, and Control) are tight coupling and considered bad, whereas Stamp, Data, and No Direct are considered good. Some researchers have extended the original coupling levels to include object-oriented concepts. The most commonly used are the following:

Level	Description
Interaction	Whether two modules share member variables
Content	Whether two classes are related by association
Inheritance	Whether two classes are related by inheritance
Class	Whether a method of a class invokes a method of another class
Object	Whether there exists a state dependency between two objects during execution

Simplifying Interfaces

There is no single methodology to design functions and programs with loose coupling. Instead, a perspective change is required. One must continually ask, "What can I do to make this function easier to use?"

Best Practice 13.1 Loose coupling starts with the structure chart; ask coupling questions with every arrow

Modularization design should always start with a structure chart. This not only helps the programmer visualize design ideas, but it also helps him or her ask important modularization questions. To illustrate this process, consider the following draft of a structure chart representing the part of a financial program pertaining to displaying reports to the user. We will focus on the `displayHistogram()` function.

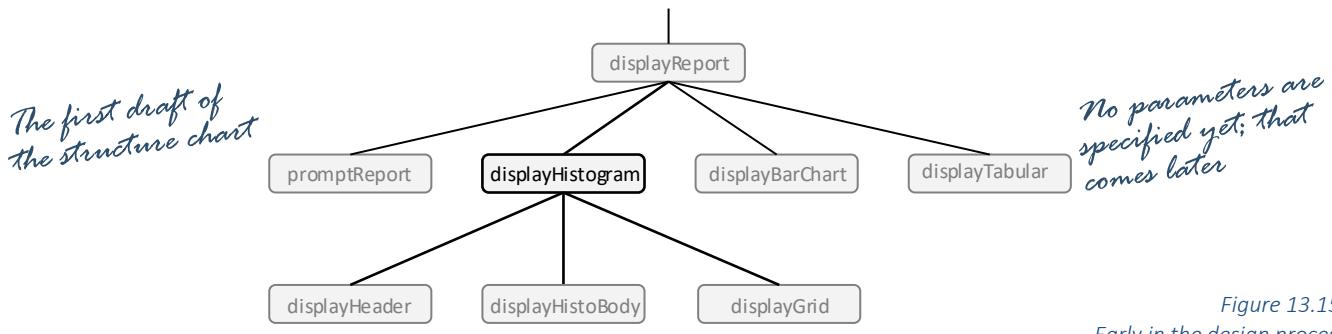


Figure 13.15:
Early in the design process

When looking at this function from a coupling perspective, we need to know what kind of information `displayHistogram()` needs to fulfill its purpose and what kind of information it will pass on to its children. At this point, it is not uncommon to discover that our current layout of functions requires excessive amounts of data to

be passed. A draft of the parameters into `displayHistogram()` and the parameters that it passes onto `displayHistoBody()` reveals that a large amount of content is just passing through the function. We are asking `displayReport()` or perhaps the function that calls it to fetch all the account data we need before we even decide if it will be necessary for our report. We can do better.

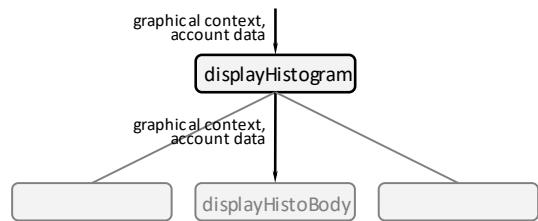


Figure 13.16:
First pass at specifying
parameters

With our second draft, we will attempt to simplify the parameter passing and have the caller do less work. We can accomplish this by passing only the account number and graphical context to the `displayHistogram()` function and having `displayHistoBody()` fetch the account data from the account number. This has several advantages. First, it is now much easier to call the display functions; we only need an account number and the graphical context. Second, we move the fetching code closer to the graphing code. As a general rule, the closer we put these items, the less likely that we will end up fetching data we don't need.

We will finish the design process by moving through the rest of the functions in our structure chart, analyzing each from a coupling perspective.

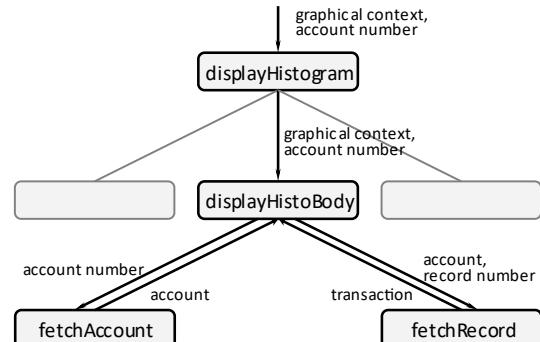


Figure 13.17:
A new design with
looser coupling

More often than should be the case, programmers are forced to use complex interfaces when working with external components. This sometimes even happens when working with new code in our own codebases. The reasons for this are legion, but they often boil down to this: the author of the interface did not anticipate the needs of your particular application. In cases such as these, what is to be done?

In order to protect the quality of our code, we need to shield our program from the complexity of external influences. This can be accomplished with a technique called a façade. A façade is a layer bridging the gap between two incompatible or disjoint parts of the program. Its job is to translate data from one format into another. A façade can help provide simple coupling between the new code and the rest of the system by shielding the system from the complexities of the new code. It can also simplify the coupling in the new code by shielding it from the complexities of the rest of the system.

If an interface with another component is complex, then have one function handle that interface. It will then communicate with the rest of the program using a simpler interface. Do not pass on the complexity, which requires many functions to have tight coupling. Instead, isolate that complexity in just one or two functions.

Examples

Examples 13.1: Document Coupling

This example will demonstrate how to recognize document coupling.

Problem

Identify the level of coupling in the following function:

C++

```
// input: "Schmitt, John Jacob Jingleheimer"
// output: {"Schmitt", "John", "Jacob Jingleheimer"}
FullName parseFullNameFromText(const char * input)
{
    FullName name;

    // last name
    for (; *input != ','; input++)
        name.last += *input;

    // consume comma and space
    assert(input[0] == ',' && input[1] == ' ');
    input += 2;

    // first name
    for (; *input != ' '; input++)
        name.first += *input;

    // consume the space
    assert(*input == ' ');
    input++;

    // grab the rest of the text
    name.other = input;
    return name;
}
```

Solution

We can rule out trivial because there is a parameter. It is not encapsulated because the parameter is of a primitive data type that does not provide any guarantee of validation. It is not simple because the client needs to understand details of C strings to use the `input` parameter. It may be complex or document. It cannot be interactive because there is no state or session maintained between function calls. Finally, the `input` parameter is utilized so it is not superfluous.

One could argue that this is complex because details of C strings need to be completely understood to consume the `input` variable. Note from the comment how the string is in a specific format with commas and spaces. This qualifies as language including syntactic and/or semantic rules. The function thus exhibits document coupling.

Examples 13.2: Interactive Coupling

This example will demonstrate how to recognize interactive coupling.

Problem

Identify the level of coupling in the following code:

```
PHP
<?php
// Establish a connection with the database
$connection = mysqli_connect($serverName, $userName,
                             $passWord, $databaseName);

// Issue the SQL Query
$sql = "SELECT player, score FROM ScoreBoard";
$result = mysqli_query($connection, $sql);

// Display the results
if (mysqli_num_rows($result) > 0) {
    while($row = mysqli_fetch_assoc($result)) {
        echo $row["player"] . " " . $row["score"] . "<br>";
    }
}

// Close the connection
mysqli_close($conn);
?>
```

Solution

Notice that there are multiple steps involved in making the SQL connection. First, the connection is established through the `mysqli_connect()` function call. Next, the query is sent through `mysqli_query()`. After that, the results are parsed through `mysqli_num_rows()` and `mysqli_fetch_assoc()`. The final step is to close the connection through `mysqli_close()`. This interaction is complex, involves many steps, and involves maintaining state (through `$connection` and `$result`). Therefore, this is interactive coupling.

Examples 13.3: Designing for Coupling

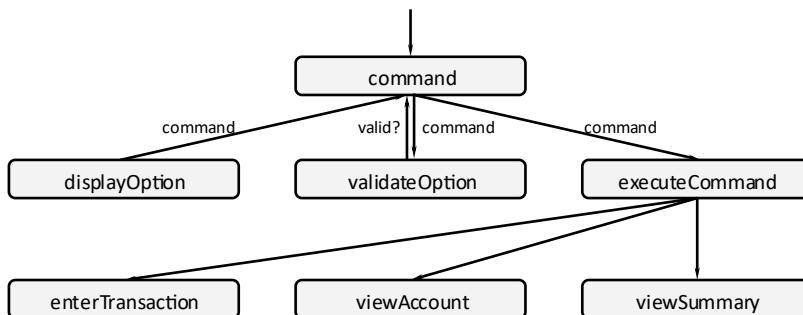
This third example will demonstrate how to design a program to simplify coupling.

Problem

A program displays a selection of options to the user:

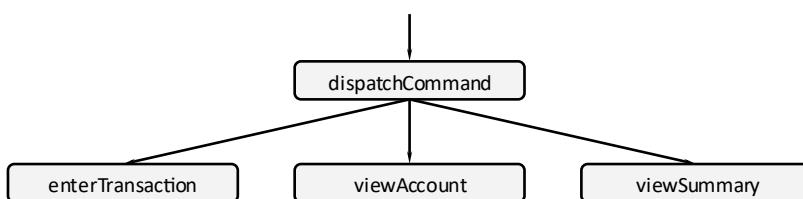
```
Output
Select your option:
1 ... Enter transaction
2 ... View account details
3 ... Display account summary
4 ... Change to a different account
5 ... Balance account
6 ... Connect to financial institution
7 ... Quit
>
```

Our initial design prompts the user for an option and then calls the appropriate function in the following way:



Solution

The problem with this design is that three functions (`displayOption()`, `validateOption()`, and `executeCommand()`) all have to completely understand the command language. If an option is added, removed, or changed, then all three functions would need to be updated. This is an example of document coupling. We can simplify the interface by combining these three functions into `command()`. Note that this will not reduce the cohesion of `command()` because it still does one thing and one thing only: dispatch a command.



Exercises

Exercise 13.1: Define the Level of Coupling

From memory, recite the definition of each level of coupling.

Level	Definition
Trivial	
Encapsulated	
Simple	
Complex	
Document	
Interactive	
Superfluous	

Exercises 13.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Coupling is a measure of how many parameters are passed between functions	
Trivial coupling maps closely to Constantine's No Direct coupling	
Coupling is a modularization metric	
Simple coupling is more desirable than encapsulated coupling	
Interactive coupling requires each client to maintain session state	

Exercise 13.3: Identify the Level of Coupling

Identify the level of coupling for the following function.

C

```
int promptForAge(int age)
{
    printf("What is your age? ");
    scanf("%d", &age);
    return age;
}
```

Exercise 13.4: Identify the Level of Coupling

Identify the level of coupling for the following function.

PHP

```
<?php
function display_coordinates($row, $column)
{
    echo "($row, $column)";
}
?>
```

Exercise 13.5: Identify the Level of Coupling

Identify the level of coupling for the following function.

VB

```
Sub displayInstructions()
    Console.WriteLine("Select your option:")
    Console.WriteLine("    1 ... Enter transaction")
    Console.WriteLine("    2 ... View account details")
    Console.WriteLine("    3 ... Display account summary")
    Console.WriteLine("    4 ... Change to a different account")
    Console.WriteLine("    5 ... Balance account")
    Console.WriteLine("    6 ... Connect to financial bank")
    Console.WriteLine("    7 ... Quit")

End Sub
```

Problems

Problem 13.1: Coupling of Is Punctuation

Identify the level of coupling for the following C# function.

C#

```
// Determine if a given word is a punctuation mark
public bool isPunctuation(string word)
{
    char[] punct = {',', '.', '!', '?', ';', ':'};

    foreach (char punctuationMark in punct)
        if (punctuationMark == word[0])
            return true;
    return false;
}
```

Problem 13.2: Coupling of Symbol Type

Identify the level of coupling for the following C++ function.

C++

```
int symbolType(char word[])
{
    if (word[0] == ':')           // determine if it's special
    {
        if (word[1] == '!')       // treat it like a new line
            return 1;
        else if (word[1] == '<') // treat it like an open quote
            return 2;
        else if (word[1] == '>') // treat it like a close quote
            return 3;
        else if (word[1] == '.') // treat it like a period
            return 4;
        else if (word[1] == ',') // treat it like a comma
            return 5;
    }

    else
        return 6;                // leave it unchanged
}
```

Problem 13.3: Coupling of Play Again

Identify the level of coupling for the following JavaScript function.

JavaScript

```
// Determine if the user wants to quit the game or play again
function playAgain() {
    var response = prompt("Do you want to quit?", "no");
    return response;
}
```

Problem 13.4: Coupling of Word Wrap

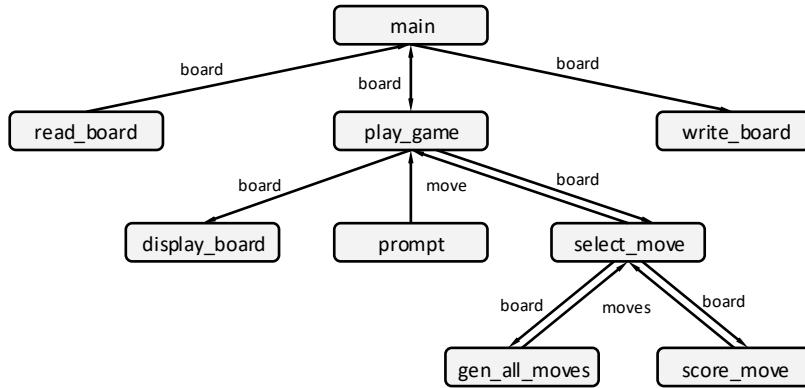
Identify the level of coupling for the following pseudocode function.

Pseudocode

```
wordWrap(text)
nColumn ← 0
FOR i ← 0 ... text.size
    SWITCH text[i]
        CASE newline
            nColumn ← 0
        CASE space
            nColumn++
            iLastSpace ← i
        CASE tab
            nColumn ← (nColumn / sizeTab + 1) × sizeTab
        DEFAULT
            if nColumn > sizeLine && iLastSpace
                text[iLastSpace] ← newline;
                nColumn ← 0
            else
                nColumn++
```

Problem 13.5: Tic-Tac-Toe Design

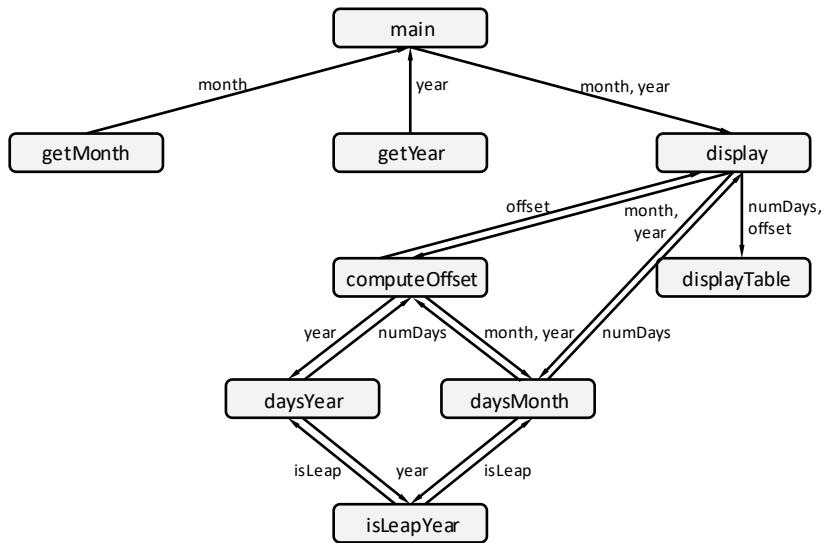
Consider the following structure chart for a program which plays the game of Tic-Tac-Toe. Each parameter is a built-in data type (a board is an array of characters, a move is an integer address for a square). If you have done Problem 10.5, please use that structure chart.



Determine the level of coupling for every function in the structure chart. If any function could be improved, then refactor the design.

Problem 12.6: Calendar Design

Consider the following structure chart for a program which displays a calendar on the screen. Each parameter is a simple integer. If you have done Challenge 10.1, please use that structure chart.



Determine the level of cohesion for every function in the structure chart. If any function is less than strongly cohesive, then refactor the design.

Challenges

Challenge 13.1: Sudoku

Create a structure chart for the game of Sudoku. If you have already created a structure chart for this in Challenge 10.1 or Challenge 12.1, please use that one.

Sudoku is a numbers game played on a 9x9 grid. The object of the game is to fill in the 9x9 grid while honoring certain constraints. This program will read a user-specified board from a file, allow the user to interact with the game while enforcing the rules, and then write an unfinished board to the file.

Determine the level of coupling for every function in the structure chart. If any function is worse than simple coupling, refactor the design.

Challenge 13.2: Yahtzee

Create a structure chart for the game of Yahtzee. If you have already created a structure chart for this in Challenge 12.2, please use that one.

Yahtzee is a multi-round game of dice. The user throws five dice each turn and tallies points on a scorecard. Each category has a set of rules, describing the set of numbers on the dice necessary to complete the category. Every round requires a category to be filled, even if the current roll yields zero points for that category. The game is continued until every category has a score.

Determine the level of coupling for every function in the structure chart. If any function is worse than simple coupling, refactor the design.

Challenge 13.3: Card Game

Create a structure chart to describe a program that plays a card game of your choice. For each function, identify the corresponding level of coupling. If any function is less than simple coupling, rework the design until the coupling problem is fixed.

Challenge 13.4: SMTP

Research the SMTP e-mail protocol. What level of coupling would be required when two entities communicate through this protocol? If you were to design this protocol from scratch, how could you simplify it so it exhibits looser coupling?

Test Case

Chapter 14

A test case is a set of actions used to verify that a product, feature, or module behaves as expected. It is a single run-through of the application designed to determine whether a given input will produce the expected output.

Imagine a test pilot walking up to a newly designed aircraft after having been informed that the most rudimentary checks have not been conducted. He would never get into that airplane! Imagine a patient preparing to receive a new pacemaker when she overhears the doctor saying that it has not been tested. She would never allow that to happen! Testing is an essential step with every system of even the most moderate degree of complexity. Why, then, do we often skip this step with something as complex as software?

A test case is a set of actions used to verify a product, feature, or module behaves as expected

An essential component of any test plan, be that testing conducted by the developer as the code is being written or conducted by a quality assurance engineer before the product is delivered to the client, is the identification of test cases. A test case is a set of actions used to verify a product, feature, or module behaves as expected. It is a single run-through of the application designed to determine whether a given input will produce the expected output.

Name	Symbol for username
ID	#219
Priority	High
Requirement	#2: The username control shall accept only letters, numbers, and underscores
Area	Authentication
Pre-Condition	None
Input	Username: "@" Password: "passw0rd"
Steps	1. Boot the application 2. Press the [Login] button 3. Enter "@" in the username edit control 4. Enter "passw0rd" in the password edit control 5. Press [Login]
Output	Error: "No symbols allowed in a username"

Test case identification should be done at all phases of the software development cycle. It should be done at the requirements elicitation phase during the first interactions with the client. It should be done during design phase as different options are explored. It should be done during code writing as the programmer identifies edge cases and assumptions inherit in the code. Finally, it should be done at the quality assurance phase when the final product is prepared for the client. Teams that value quality continually seek to increase it whenever possible. Discovery of a good test case is valued in organizations such as these.

Quality

The purpose of test cases is to discriminate between normal behavior and deviant, between functional software and a bug. While some types of defects (such as a crash) are easy to spot, most of the interesting defects are far more subtle. The first step in any testing process is to precisely define quality in the context of the project.

According to the International Standards Organization (ISO), quality is “the totality of features and characteristics of a product or service that bears its ability to satisfy stated or implied needs.” In other words, quality is fitness for use by the client. If the software satisfies the client’s needs, then it is a quality product. From this definition we can see that quality is a relative metric expressed in terms of the client’s needs rather than an absolute metric that universally applies to all systems.

Quality is the degree in which something is fit for use by the client

Before beginning to define quality for your project, it is useful to explore the various aspects of quality that may influence a product. It is rare for all of these aspects to equally relevant for a given product; it is up to the development team to see which are important and which can safely be ignored.

Quality Component	Definition
Correctness	The extent in which a given input produces the expected output
Fault Tolerance	The degree in which the program can continue to function in the face of errors or unexpected input
Efficiency	The amount of resources required by the system to function
Usability	The amount of effort required of users to learn, understand, prepare input, and interpret system output
Supportability	The degree of effort required to keep the system working
Security	The capacity of the system to provide confidentiality, integrity, and availability assurances
Interoperability	The ability of the system to collaborate with other systems
Reliability	The capacity of the system to produce output to the required level of precision

In those cases when the development team is contracted to fulfill a rigid set of requirements, quality identification is very straightforward: one only must fulfill the contract. Unfortunately, there have existed many systems which match the specification and fulfill all the requirements yet were not “fit for use.” The reason is this: it is nearly impossible to create a complete and correct requirements list before the client has any direct interaction with the system.

Quality identification is the job of the entire development team and is an ongoing process

Most development scenarios require deep client involvement in the quality specification and requirement elicitation process. The details and techniques of this process are beyond the scope of this book. It is sufficient here to say that quality identification is the job of the entire team and is an ongoing process. At all stages of the software development process, quality determinations should be documented and verified with the client. This is important because, again, quality is defined as fitness for use by the client. If the client is not satisfied with the product, then there is a quality problem regardless of what the engineers think!

Identifying Test Cases

In many ways, a software engineer coming up with test cases is like a scientist coming up with experiments. In the former case, the engineer is looking for information that will tell her about the quality of the system. In the latter case, the scientist is looking for information that will tell him about how the universe works. In both cases, they create a series of experiments, each designed to provide insight about their field of study. When going about the process of identifying test cases, it is useful to think of yourself as a scientist. You are always seeking the highest quality information at the lowest possible cost. In effect, you are creating experiments.

An engineer identifying test cases is like a scientist defining an experiment

Meta-Metrics

A metric is a standard through which quality determinations are made. In many ways, that is what test cases are: metrics. As we develop test cases, we are creating rulers from which we measure the software. How, then, do we determine the quality of the rulers themselves?

Meta-metrics determine the quality of metrics

A meta-metric is a metric from which we measure metrics. Perhaps this is best explained by example. A track coach needs to identify who can jump the furthest, needing a technique which is fast, reliable, and precise. The chosen metric is a long piece of string: when an athlete jumps, a knot is tied in the string at the corresponding distance. While this may work to a degree, there are some obvious flaws: the string stretches, tying a knot is time-consuming, and each knot shortens the string. The meta-metrics of speed, reliability, and precision are not met!

There are three meta-metrics with which all test cases must meet: validity, reliability, and efficiency. Any test case failing one of these meta-metrics needs to be refined.

Validity

Validity refers to the degree in which our metric is measuring the right thing. Back to our track example, our coach would now like to know which athlete can run 100 meters the fastest. Wanting to give every athlete an equal chance, the coach decides to judge them based on effort. Every student will run 100 meters and the coach will ask them if they “gave it their all.” Note that this is not a valid measure of speed, it is a measure of effort! This metric fails a basic validity test. A more valid measure would be to time how long it takes for them to run 100 meters.

Validity is the degree in which our metric measures the right thing

When choosing test cases, it is important to always measure the right thing. If asked to test how well an edit control can handle invalid or malicious data, make sure you create a test case that includes invalid and malicious data.

Best Practice 14.1 Make every test case traceable to a requirement

If you cannot trace a test case directly to a requirement, stakeholder need, or user scenario, then it is likely that test case has validity challenges. The more difficult it is to make that connection, the more you should be concerned.

Reliability

Reliability is how much noise there is in the measurement

Reliability, also known as accuracy, is how much noise there is in the measurement. If you were to make the same measurement twice, how different would the two values be? Back to our track example, our track coach decided to measure the students on elapsed time for 100 meters of running rather than on effort. However, on the day the test was conducted, some athletes forgot their running shoes and others just ate a large lunch. Learning of this, the coach conducted another test the next day. When comparing the results, the top-10 runners were different between the two days. Clearly, there are reliability issues with this metric.

Best Practice 14.2 Be very specific about the test case's pre-conditions

Recall that a pre-condition is the setup we do before the actual test is run. This could include the input and state of the system variables. It could also include the environment in which the system is running. For example, a performance metric may be unreliable if it measures elapsed time to complete a task when the pre-condition failed to mention the need to stop all background processes from running. Make sure that the test case's pre-conditions are specific enough that the test will always be run in the same environment.

Best Practice 14.3 Avoid generalities in test cases

A test-case can have reliability challenges when the instructions say things like "enter some text." The phrase "some text" may mean the word "hamster" to one tester and the entirety of the Old Testament to another. A more reliable test case would specify exactly the text to be entered. This is sometimes called "canned data."

Efficiency

Efficiency refers to the time, effort, or money required to obtain a measurement. Back to our track example, the coach decides to measure how long it takes the athletes to run 100 meters (a valid metric) by having them run the distance ten times (a reliable metric). The problem with this approach is that the cost of obtaining the measurement is too great. It will be hard on the coach and even harder on the athletes. There must be a better way!

Efficiency is how much time or effort is required to obtain the measurement

Best Practice 14.4 Reduce and simplify steps whenever possible

Do not describe in two words what can be expressed with one. It is tempting to go with the first test case you come up with. However, it is worthwhile to spend a moment or two honing the test case, reducing unnecessary steps and simplifying things whenever possible. What started as a convoluted and niche case may turn into something mainstream if you refine it enough.

Best Practice 14.5 Automate whenever possible

While it is acceptable to have a few manual tests in your list of test cases, it is better if everything is automated. Write code to do your heavy lifting.

Best Practice 14.6 Beware of false positives, false negatives, unhelpful logs, and verbose logs

If your test cases are buggy, then you might find yourself tracking down many false leads. This can waste your time and cause you to overlook important issues.

Techniques

Given that test cases must be valid, reliable, and efficient, how does one identify test cases? There is no single algorithm one can follow which will produce great test cases every time. Creativity and imagination are required at every step. Thankfully there are a few techniques which have been proven effective.

Requirements Can Become Test Cases

If you are working on a project with a well-defined set of formal requirements, then these requirements can be the launching point for test cases. Of course, every test case derived directly from a requirement is likely to not have validity challenges! Many projects unfortunately do not have formal requirements or, worse yet, have no requirements at all. In cases like these, what is to be done? The answer: write them. To demonstrate how this can be done, consider the following conversation between the stakeholder and the software engineer:

Requirements are so useful in the test case generation process that it is usually worthwhile to write them if none exist

Stakeholder:	We need a login screen.
Software Engineer:	A username and password edit control?
Stakeholder:	Yes, and a [Login] button.
Software Engineer:	What is the size limit on the username?
Stakeholder:	Between 1 and 16 characters.
Software Engineer:	Are there limitations with spaces, symbols, or numbers?
Stakeholder:	Good question. Only letters, numbers, and underscores.
Software Engineer:	What about the passwords?
Stakeholder:	Up to 20 characters; anything but spaces are allowed.

From this conversation, the software engineer derived the following requirements:

-
- #1: The username control shall accept between 1 and 16 characters
 - #2: The username control shall accept only letters, numbers, and underscores
 - #3: The password control shall accept between 1 and 20 characters
 - #4: The password control shall accept letters, numbers, and symbols
-

It is worthwhile for the engineer to then read these requirements back to the stakeholder to make sure that nothing is misrepresented. Now, from these requirements, some test cases are easy to identify:

Name	Pre-Condition	Inputs	Outputs	Rationale
Short Name	Empty both fields	UserName: empty	Disabled [Login] button	Empty names are invalid
One Letter Name	Empty both fields	UserName: "A"	Enabled [Login] button	Small names are valid
Symbol Name	Empty both fields	UserName: "@"	Error message	Symbols are invalid
Space Name	Empty both fields	UserName: ""	Error message	Spaces are invalid
Typical Name	Empty both fields	UserName: "JohnDoe"	Enabled [Login] button	Typical username

It may seem like extra work to generate requirements, but it is not. The software engineer needs to fully understand what the stakeholder has in mind. The earlier in the process this occurs, the less wasted time. Notice that written requirements facilitate writing test cases. You can even think of them as a steppingstone in the test case writing process.

Scenarios Can Become Test Cases

Some user needs are best captured with scenarios or use cases. These are simple stories or descriptions of how the user interacts with the system. An example scenario of a user interacting with a financial package might be the following:

Sally is reviewing her budget on the Budget Report screen. Unsure of why her grocery bill is larger than expected, she clicks on that number. A window pops up listing all her grocery purchases for the month. "Oh yeah," she says, remembering the dinner party she threw last week. She closes out of the pop-up window and the budget report and continues her work.

From this scenario, we can see several things the software must do.

-
- #1: The budget report must show the planned spending for the month
 - #2: The budget report must show the actual spending for the month
 - #3: All the actual spending numbers must be clickable
 - #4: Clicking on a spending number must instantiate a purchase window
-

We could probably come up with a dozen or more items from this scenario. As with requirements, several test cases corresponding to this scenario follow:

Name	Pre-Condition	Inputs	Outputs	Rationale
Empty Budget	No budget specified	Report > Budget	Empty budget column	Empty budgets should work
Empty Spending	Budget, no spending	Report > Budget	Empty spending column	Empty spending works
Simple Budget	2 budget fields 2 spending	Report > Budget	2 items in budget, spending	Simple
Space Name	Empty both fields	User Name: ""	Error message	Spaces are invalid
Typical Name	Empty both fields	User Name: "JohnDoe"	Enabled [Login] button	Typical username

Error Conditions Can Become Test Cases

Of course, a program must function properly in the face of valid data. It is also necessary for a program to function properly in the face of error conditions. As with requirements and scenarios, it is best to start with a complete and correct set of error conditions for any function you may be working with. This is seldom the case in practice. It usually falls to the software engineer to identify and record these error conditions. For example, consider a function that converts a GPA into a letter grade. Since $0.0 \leq \text{GPA} \leq 4.0$, there are two sets of error conditions: everything above 4.0 and everything below 0.0.

Every error condition should get at least one test case

Equivalence classes are sets of input that have the same characteristics

Notice that all negative numbers are treated the same. This is called an equivalence class: the set of values that have the same characteristics. In other words, we can treat -5.0 and -5.1 the same; we

don't need to test them both. For this example, there are three equivalence classes or equivalence partitions: Too high (∞ to 4.00001), valid (4.0–0.0), and negative (-0.00001 to $-\infty$). We want to make sure that we cover each equivalence class at least once.

Name	Pre-Condition	Inputs	Outputs	Rationale
Too High	None	5.5	Error	Above 4.0
Valid	None	3.0	B	$0.0 \leq \text{GPA} \leq 4.0$
Negative	None	-5.0	Error	Below 0.0

Boundary Conditions Can Become Test Cases

Boundary conditions are those locations on the border between equivalence classes. Back to our GPA example, we have established that there are three equivalences classes: too high, valid, and negative. These are illustrated below:



Figure 14.1:
Boundary conditions

Notice that there are four boundary conditions: very high numbers, 4.0, 0.0, and very low numbers. To properly test boundary conditions, we want to test just above each condition, the condition itself, and just below. Since we cannot go any higher than the maximum floating-point number of 3.403×10^{38} and cannot go any lower than -3.403×10^{38} , there are 10 boundary conditions to check:

Name	Inputs	Outputs	Rationale
Max float	3.403×10^{38}	Error	Anything larger than 4.0, even much larger, is an error
Just below max float	3.402×10^{38}	Error	This might be different from "Max float"
Just above 4.0	4.00001	Error	This should be different from 4.0
4.0	4.0	A+	$0.0 \leq \text{GPA} \leq 4.0$. Notice that 4.0 is legal
Just below 4.0	3.99999	A+	$0.0 \leq \text{GPA} \leq 4.0$
Just above 0.0	1.175×10^{-38}	F	$0.0 \leq \text{GPA} \leq 4.0$
0.0	0.0	F	$0.0 \leq \text{GPA} \leq 4.0$. Notice that 0.0 is legal
Just below 0.0	-1.175×10^{-38}	Error	Even a number slightly below 0.0 is an error
Just below min float	-3.402×10^{38}	Error	< 0.0
Min Float	-3.403×10^{38}	Error	Even the smallest value is still negative and thus an error

An exhaustive set of test cases will explore each side of all boundary conditions for the input.

Best Practices

Aside from the standard places to look for test cases, there are a couple best practices:

Best Practice 14.7 Don't assume that current behavior is the expected behavior

Look at the system with a critical eye. Just because it has "always been done this way" does not mean that it is right. Do some research to see what the client or the user expects.

Best Practice 14.8 Think BAOE (Basic, Alternate, Options, and Exceptions) when writing test cases

The basic flow is the mainstream scenario, constituting the most common user experience. This must work flawlessly. Alternate flows are variations and nonstandard scenarios. Each of these will probably be explored by the user on occasion. Options represent settings that the user may manipulate. Not only should each option be represented in a test case, but all relevant combinations should be as well. Finally, exceptions represent errors. Each error condition should be represented with a test case.

Documenting Test Cases

When writing a function, the programmer typically knows how it works and what types of input is expected. At this point, it is easy to manually run through the necessary test cases and be reasonably sure that everything is covered. There is a problem with this approach: what happens if the tests need to be run again in a few weeks or months? What happens if the programmer is not present? What happens if the functionality is changed and the new programmer needs to know how the old code was expected to perform? In other words, we need to be able to replicate a test. This is why documentation is so important.

Documentation can be as simple as a few comments above a function or as complex as a formal legally binding document. Whatever the format, we need to document our testing to ensure we have reliable and replicable results.

Basic

The most basic way to document test cases is to simply create a two-column table: input on the left and output on the right. While this is enough for very simplistic test cases, most need a couple of extra fields.

Column Name	Use
Name	The name of the test case, usually just a few words
Pre-Condition	The state of the system immediately before the test is run
Input	Input into the system
Output	The expected output

An example of a basic test case is the following:

Name	Symbol for username
Pre-Condition	None
Input	Username: "@" Password: "passw0rd"
Output	Error: "No symbols allowed in a username"

Even when creating the most basic documentation of test cases, make sure that they are written with enough detail and clarity that anyone will conduct the test the same as you. Anything less and we threaten the reliability of our test cases.

Complete

In most cases, considerably more documentation is required than is needed for the basic test cases. The most common extra fields include the following:

Column Name	Use
ID	Each test case should have an ID so it can be uniquely referenced
Requirement	To be valid, each test case should cite a specific requirement
Area	What part of the code does this test case exercise?
Priority	Distinguish most important test cases from less important ones

An example of a complete test case is the following:

Name	Symbol for username
ID	#219
Priority	High
Requirement	#2: The username control shall accept only letters, numbers, and underscores
Area	Authentication
Pre-Condition	None
Input	Username: "@" Password: "passw0rd"
Steps	1. Boot the application 2. Press the [Login] button 3. Enter "@" in the username edit control 4. Enter "passw0rd" in the password edit control 5. Press [Login]
Output	Error: "No symbols allowed in a username"

Remember that the goal of documenting test cases is to increase reliability, so they can be replicated by another programmer without your intervention. Anything that helps with that goal is worthwhile, anything that distracts should be avoided.

Software Solutions

Test cases can easily multiply until they are difficult to manage. In cases like that, it might be necessary to use a more robust system to manage them. Many teams use a spreadsheet where each column is a field in the test case table and each row is an individual test. This technique allows the team to quickly identify the test cases associated with a given feature or requirement. It also allows them to quickly identify the highest priority tests if the entire system needs to be reverified.

There are also software testing suites that facilitate the creation, documentation, and reporting of test cases. The specifics of these tools is beyond the scope of this textbook, but most software engineers work with several variations during the course of their career. An example of using a spreadsheet to store test cases is the following:

*Figure 14.2:
Test cases managed
by a spreadsheet*

Name	Prior.	Requirement	Area	Pre-Condition	Input	Steps	Output
Max float	136	3 #214 Error	GPA	Navigate to GPA dialog	3.403×10^{38}	Enter input and press Submit	Error
Just below max float	137	3 #214 Error	GPA	Navigate to GPA dialog	3.402×10^{38}	Enter input and press Submit	Error
Typical High	138	2 #214 Error	GPA	Navigate to GPA dialog	5.5	Enter input and press Submit	Error
Just above 4.0	139	3 #214 Error	GPA	Navigate to GPA dialog	4.00001	Enter input and press Submit	Error
4.0	140	1 #213 Valid	GPA	Navigate to GPA dialog	4.0	Enter input and press Submit	A+
Just below 4.0	141	2 #213 Valid	GPA	Navigate to GPA dialog	3.99999	Enter input and press Submit	A+
Typical Valid	142	1 #213 Valid	GPA	Navigate to GPA dialog	3.0	Enter input and press Submit	B
Just above 0.0	143	2 #213 Valid	GPA	Navigate to GPA dialog	1.175×10^{-38}	Enter input and press Submit	F
0.0	144	1 #213 Valid	GPA	Navigate to GPA dialog	0.0	Enter input and press Submit	F
Just below 0.0	145	3 #214 Error	GPA	Navigate to GPA dialog	-1.175×10^{-38}	Enter input and press Submit	Error
Typical Negative	146	2 #214 Error	GPA	Navigate to GPA dialog	-5.0	Enter input and press Submit	Error
Just below min float	147	3 #214 Error	GPA	Navigate to GPA dialog	-3.402×10^{-38}	Enter input and press Submit	Error
Min Float	148	3 #214 Error	GPA	Navigate to GPA dialog	-3.403×10^{-38}	Enter input and press Submit	Error

Examples

Example 14.1: Is Prime

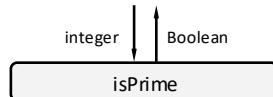
This example will demonstrate how to create test cases for a simple function.

Problem

A prime number is defined the following way:

A prime number is a whole number greater than 1 whose only factors are 1 and itself

Create test cases for a function that takes an integer as a parameter and returns whether the number is prime:



Solution

One equivalence partition is those numbers known to be prime. Only four numbers are needed in this set: 2, 3, 29, and 2147483647.

Name	Inputs	Outputs	Rationale
Prime 2	2	True	Smallest prime should certainly be tested
Prime 3	3	True	Small prime
Prime 29	29	True	Any prime would work here
Prime Max	2,147,483,647	True	Largest prime number in a 32-bit integer

Another equivalence partition is those numbers known not to be prime.

Name	Inputs	Outputs	Rationale
Not Prime 4	4	False	Smallest composite (non-prime)
Not Prime 21	21	False	Small composite
Not Prime Max	2,147,483,646	False	Largest composite that is a 32-bit integer

Two boundary conditions should be checked: 0 and 1.

Name	Inputs	Outputs	Rationale
Not Prime 0	0	False	Zero is always unique
Not Prime 1	1	False	Specifically not prime

Negative numbers, whether the absolute value is prime or composite, are not prime.

Name	Inputs	Outputs	Rationale
Negative 1	-1	False	-1 is a special case
Negative 2	-2	False	2 is prime but -2 is not
Negative 4	-4	False	Opposite smallest composite number
Negative 21	-21	False	Opposite a composite
Negative 29	-29	False	Opposite a prime

Example 14.2: Convert Grade

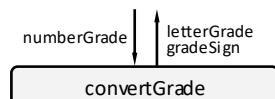
This example will demonstrate how to create test cases for a simple function.

Problem

A given class has the following mapping from number grade to letter grades:

Range	Grade
100% - 93.00%	A
92.99% - 90.0%	A-
89.99% - 87.00%	B+
86.99% - 83.00%	B
82.99% - 80.00%	B-
...	

Create test cases for a function that takes an integer as a parameter and returns a letter grade and a grade sign:



Solution

One equivalence partition is numbers in the middle of the range: 95%, 91%, 88% etc.

Name	Inputs	Outputs	Rationale
Typical A	95.00	A	Typical A
Typical A-	91.00	A, -	Typical A-
Typical B+	88.00	B, +	Typical B+
...			

Another equivalence partition is those on the boundary: 100%, 93%, 90% etc.

Name	Inputs	Outputs	Rationale
A+ boundary	100.00%	A	Highest possible
A to A- boundary	93.00%	A	On A to A- boundary
A- to B+ boundary	90.00%	A, -	On A- to B+ boundary
...			

Next, test the error conditions: negative numbers and greater than 100%.

Name	Inputs	Outputs	Rationale
Little too high	100.01%	Error	Above 100% is a problem
Little too low	-1.175 x 10 ⁻³⁸	Error	Just slightly below 0%
Way too low	-3.403 x 10 ³⁸	Error	Smallest possible floating-point number

Finally, make sure that the various flavors of F do not have signs.

Name	Inputs	Outputs	Rationale
No F+	59.0%	F	No F+ please
No F-	51.0%	F	No F- please

Exercises

Exercise 14.1: Define the Meta-Metrics

From memory, recite the definition of each meta-metric.

Level	Definition
Validity	
Reliability	
Efficiency	

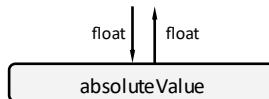
Exercise 14.2: Scenarios

In each of the following scenarios, determine if the test suffers from validity, reliability, or efficiency issues.

Scenario	Type of Defect
A test is created that measures execution time, namely how long it takes to run. This can be impacted by the other programs on the system.	
A test is created that requires the programmer to click through 3 dialogs and enter 15 different values in various locations.	
A test meant to measure algorithmic efficiency actually measures memory usage.	
The same test was run five times, getting one answer twice and another answer the remaining three time.	
Each test takes a minute to execute, making the entire test suite take over an hour.	
A test does not map to any requirement.	

Exercise 14.3: Types of Test Cases

There is a function which computes the absolute value of a number.



For each of the following test cases, identify if it is a requirement/scenario, error condition, or boundary condition:

Input	Type of test case
1.175×10^{-38}	
-32.0	
3.403×10^{38}	
0.0	
-1.175×10^{-38}	
NaN (not a number, invalid floating-point number)	
32.0	
-3.403×10^{38}	

Exercises 14.4: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Every error condition should get at least one test case	
Quality is the job of the quality assurance engineer, not the programmers	
Quality is roughly defined as fitness for use	
Test case identification should be done at all phases of the software development cycle	
Test cases should explore all sides of boundary conditions	

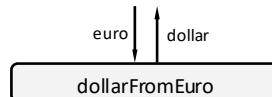
Problems

Problem 14.1: Dollar from Euro

The exchange rate between dollars and euros is the following:

€1.00 = \$1.13

Create test cases for a function that takes a euro as a parameter and returns the corresponding US dollars. Pay special attention to boundary conditions.



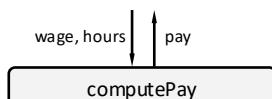
In the language of your choice, implement this function and run it through your test cases. If you found any defects, run the tests again to make sure there have been fixed.

Problem 14.2: Compute Pay

Overtime is computed the following way:

Condition	Pay
0 hours → 40 hours	wage x hours
More than 40 hours	(wage * 40) + wage * 1.5 * (hours - 40)

Create test cases for a function that takes wages and hours as parameters and returns the pay the employee can expect. Pay special attention to error conditions and boundary conditions.



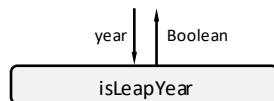
In the language of your choice, implement this function and run it through your test cases. If you found any defects, run the tests again to make sure there have been fixed.

Problem 14.3: Is Leap Year

A leap year is computed the following way:

According to the Gregorian calendar, which began use in 1753, years evenly divisible by 4 are leap years, with the exception of centurial years that are not evenly divisible by 400. Therefore, the years 1700, 1800, 1900 and 2100 are not leap years, but 1600, 2000, and 2400 are leap years.

Create test cases for a function that takes a year as parameters and returns whether the year is a leap year:

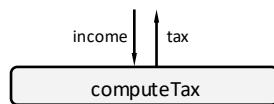


Problem 14.4: Compute Tax

Taxes are computed the following way:

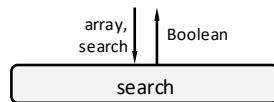
Income range	Tax is
\$0 – \$15,100	10% of amount over \$0
\$15,100 – \$61,300	\$1,510 plus 15% of amount over \$15,100
\$61,300 – \$123,700	\$8,440 plus 25% of amount over \$61,300
\$123,700 – \$188,450	\$24,040 plus 28% of amount over \$123,700
\$188,450 – \$336,550	\$42,170 plus 28% of amount over \$188,450
\$336,550 – no limit	\$91,043 plus 35% of amount over \$336,550

Create test cases for a function that takes income as parameters and returns the associated tax burden:



Problem 14.5: Search

Create test cases for a function that determines whether a given number is in an array. The first parameter is a sorted array of integers. The second parameter is the search value, also an integer. The function will return `true` if the number is found and `false` otherwise.



Challenge

Challenge 14.1: Insertion Sort

A sort is an algorithm which converts an unordered array of values into a sorted array. For each element in the sorted array a_i , we know that $a_i \leq a_{i+1}$. The following algorithm performs the insertion sort:

Pseudocode

```
sortInsertion(array, n)
    FOR iPivot ← n - 2 ... 0
        valuePivot ← array[iPivot];
        iInsert ← binarySearch(array, valuePivot,
                               iPivot + 1, n - 1)
        iInsert--
        FOR iShift ← iPivot ... iInsert
            array[iShift] ← array[iShift + 1]
        array[iShift] ← valuePivot
```

Create test cases for a function that takes an array as a parameter (including the length of the array if that is not provided in your language of choice), and modifies the array so it becomes sorted. Implement this function in the programming language of your choice. Next, test your function using your generated test cases. If any bugs are found, fix them and run your test cases again.

Challenge 14.2: Fibonacci

The value for a given number in the Fibonacci sequence can be computed by adding the previous two values in the sequence.

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n - 1) + F(n - 2) & \text{if } n > 1 \end{cases}$$

If you completed Challenge 08.1: Fibonacci, please use the algorithm you developed there. Otherwise, use the following algorithm:

Pseudocode

```
fibonacci(n)
    IF n ≤ 1
        RETURN n
    ELSE
        RETURN fibonacci( n - 1 ) + fibonacci( n - 2 )
```

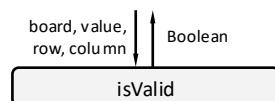
Implement this function in the programming language of your choice. Next, test your function using your generated test cases. If any bugs are found, fix them and run your test cases again.

Challenge 14.3: Sudoku

Consider the game of Sudoku:

Sudoku is a numbers game played on a 9x9 grid. The object of the game is to fill in the 9x9 grid while honoring certain constraints: 1) there is no more than one instance of a given number on a given row. 2) there is no more than one instance of a given number on a given column. 3) there is no more than one instance of a given number on an inside square (the 3x3 squares embedded in the 9x9 grid).

We are particularly interested in the function which determines whether a given value is legal in a given square on the grid:



Create test cases for this function. Implement this function in the programming language of your choice. Next, test your function using your generated test cases. If any bugs are found, fix them and run your test cases again.

Chapter 15 Driver

A driver is a simple function or program designed to test another function or program by generating input parameters and validating output parameters. This makes it possible to easily run through a large number of test cases for the purpose of finding defects or determining the level of code quality.

A driver is code used to test a function or another part of the system

When an automobile manufacturer wishes to test a new engine, it does not immediately drop it into an existing car and then fire it up. Instead, the engine is installed on a testing rig called a dyno (short for dynamometer). This dyno can fully exercise the engine through a wide variety of simulations while measuring torque, power, temperature, fuel consumption, and a myriad of other parameters. Not only can this testing be done more safely and cheaply than is possible in an actual car, a wider range of parameters can be tested. In fact, it is possible to run the engine through situations far beyond what an actual car will ever encounter. The software equivalent of a dyno is called a driver.

A driver is a simple function or program designed to test another function or program by generating input parameters and validating output parameters. In other words, it facilitates running through a large number of test cases in a very efficient way.

```
C#  
Console.WriteLine($"Wage      Hours      Pay      Test Name");  
Console.WriteLine($"-----+-----+-----+-----");  
  
float pay = computePay(0.00, 0.00);  
Console.WriteLine($"{0.00}      0.00      {pay}      Zeros");  
  
pay = computePay(5.00, 1.00);  
Console.WriteLine($"{5.00}      1.00      {pay}      One hour");  
  
pay = computePay(5.00, 1.00);  
Console.WriteLine($"{5.00}      2.00      {pay}      Two hours");  
  
pay = computePay(5.00, 1.00);  
Console.WriteLine($"{5.00}      40.00      {pay}      Full time");
```

Wage	Hours	Pay	Test Name
0.00	0.00	0.00	Zeros
5.00	1.00	5.00	One hour
5.00	2.00	10.00	Two hours
5.00	40.00	200.00	Full time

Figure 15.1:
Driver and resulting output

Types of Drivers

There are two broad categories of drivers: manual drivers and automation. Manual drivers are simple user interfaces added to the code making it easy for the tester to run through a large number of test cases. They serve to isolate the function under test from the rest of the program and make it easier to localize bugs. Automation, on the other hand, runs without user intervention. It automatically executes test cases and reports the results.

Best Practice 15.1 Use manual drivers early in the development process

When the team is early in the development process and the code is perhaps not even completely written, manual drivers are usually the best choice. They are very easy to set up and offer the highest degree of flexibility. There gets to be a point in time, however, when you find yourself using the same test cases again and again. At that time, it might be beneficial to convert manual drivers into automation.

Best Practice 15.2 Use automation after the code is written and the function or program is nearly code complete

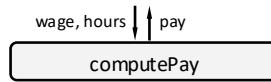
When the code is complete and the team is prepared to run through all these test cases on the function under test, it might be time to write automation. Here, all the test cases are encoded into the driver and can be executed with a single command.

There are five categories of drivers, presented below in increasing order of sophistication:

Driver Type	Description
Ad Hoc	A few lines of code used to test a function
Commandeer	Code added to <code>main()</code> to test a function
Driver Function	Permanent code written to test a specific function
Automation	A driver function working without human intervention
Test Suite	Code used to execute many automation functions

To see how this works, consider a function that computes pay based on hours worked and overtime. The structure chart for this function is the following:

Figure 15.2:
Structure chart of
`computePay()`



The first thing to notice about this function is that there are two parameters: `hours` and `wage`. This means that our driver will need to provide these two values in every test case. The second thing to notice is that there is an output parameter: `pay` which comes through the return mechanism. The driver will also need to provide a mechanism to validate the output. We call `computePay()` the function under test. The aim of the driver, regardless of the mechanism utilized, is to find bugs and determine the level of quality of the function under test. This will be demonstrated with an ad hoc driver, a commandeer driver, a driver function, automation, and integration into a test suite.

A function under test is the code that a driver is designed to exercise

Ad Hoc Driver

The easiest way to test a function is to write a simple loop prompting the user for the input parameters of a function, passing those parameters on to the function, and then displaying the results. This is often accomplished in a WHILE loop.

To exercise `computePay()`, our function under test, it is only necessary to prompt for the input parameters and display the return value. The ad hoc driver might appear as follows:

Figure 15.3:
Driver for `computePay()`

```
Python
while True:
    wage = float(input("Wage: "))           # Simplest prompts
    hours = float(input("Hours: "))          # for the 2 params
    pay = computePay(wage, hours)            # Feed data in
    print("\t\t", pay)                      # Display results
```

There are a few things to notice about this code. First, we simply prompt the user for the two input parameters that `computePay()` requires. Of course, if this function required one or three parameters, the number of prompts would vary.

The second thing to notice is that the output is displayed on the screen. In this case, the output is indented by a couple of tabs so it is easier to visually distinguish it from the input prompts. It should take no more than a minute to write the code.

Now we wish to run through a few test cases to validate that our program works as expected. Our test cases are the following:

Name	Inputs	Outputs	Rationale
Zeros	Wage: 0.00 Hours: 0	0.00	Zeros in, zero out
1 hour	Wage: 5.00 Hours: 1	5.00	\$5.00 an hour for an hour
2 hours	Wage: 5.00 Hours: 2	10.00	2 hours
40 hours	Wage: 5.00 Hours: 40	200.00	Full time
41 hours	Wage: 5.00 Hours: 41	207.50	1 hour of overtime
No wage, high hours	Wage: 0.00 Hours: 30	0.00	No wage, no pay
No wage, overtime	Wage: 0.00 Hours: 41	0.00	No wage, no pay
Low wage, no hours	Wage: 1.00 Hours: 0	0.00	No work, no pay
High wage, no hours	Wage: 1000 Hours: 0	0.00	No work, no pay
Negative	Wage: -1.00 Hours: -1	Error	Must be positive

The following output is generated for the first three of these test cases. Here, user input is represented as underlined.

Figure 15.4:
Driver output

```
Output
Wage: 0.00
Hours: 0          0.00
Wage: 5.00
Hours: 1          5.00
Wage: 5.00
Hours: 2          10.00
```

Notice that easy it is to go through the test cases that were previously identified.

Commandeer Driver

A commander driver is code at the start of main to facilitate testing

Isolating a single function with a driver may seem simple in principle, but in practice it may be more complicated. It is most often the case that we need to test a function that is already integrated into a large program. This can be accomplished with a commander driver.

In most languages, program execution begins with the `main()` function or something equivalent. We can always hijack `main()` by inserting our driver code at the beginning. For example, consider the following code that does a bit of initialization required for our function to work:

```
Kotlin
fun main(args : Array<String>) {
    // read the user file and initialize everything
    UserFile(args[0]).initialize()

    // main loop
    interact(UserFile)

    // quit
    UserFile().save()
}
```

Figure 15.5:
Code before driver is
added

We can insert our driver code at the beginning of this function right after the `initialize()` method is called. Because our driver has an infinite loop, we never reach the end of the program. That is OK; we will remove the drive code once we are sure our function works as expected. In fact, this is an important component of drivers: they are for debugging and should never be in client-viewable production code.

```
Kotlin
fun main(args : Array<String>) {
    // read the user file and initialize everything
    UserFile(args[0]).initialize()

    val reader = Scanner(System.`in`)
    while (true) {
        var wage:Float  = reader.nextFloat()
        var hours:Float = reader.nextFloat()
        var pay:Float = computePay(wage, hours)
        println(pay)
    }

    // main loop
    interact(UserFile)

    // quit
    UserFile().save()
}
```

Driver code
added at the
beginning of
our program

Figure 15.6:
Driver code added to test
`computePay()`

Driver Function

In many cases, ad hoc and commandeer driver code is removed shortly after the function is completed and tested. This means we must rewrite the driver code from scratch if we want to retest the function. What a waste of time!

When working on a particularly complex program, it is not uncommon to want to save the driver code. We do this by putting the driver code in a separate function that we can later call. Usually, in the course of writing a function, the driver function is created at the same time. This both gives the author some confidence that the function will work and gives another programmer the ability to quickly retest the function if a change needs to be made.

When creating a plethora of driver functions, we run the risk of bloating the code with many functions that are not used in the final project. We also want the driver to have no impact on the production code. In other words, the driver should do no harm. One method we can use to avoid this risk is to surround the function with pre-compiler directives. This will ensure the code is not compiled unless we specifically ask for it.

A driver function is a manual driver placed in a function making it easy to retest a given function

The diagram shows a C++ code block with handwritten annotations. The code is enclosed in a blue box and labeled 'C++' at the top. It includes a pre-processor directive #ifdef TEST_COMPUTE_PAY, a documentation block, and a void computePayTest() function. The function contains a while loop that prompts for wage and hours, then tests the computePay() function with cout statements. A handwritten note on the left says: 'We must define TEST_COMPUTE_PAY in order to run this driver function'. A handwritten note on the right says: 'The function is called here, with the output nicely formatted'.

```
C++
#ifndef TEST_COMPUTE_PAY
/*
 * COMPUTE PAY TEST
 * A simple driver function to exercise computePay()
 */
void computePayTest()
{
    while (true)
    {
        // prompt for wage
        float wage;
        cout << "Wage: ";
        cin >> wage;

        // prompt for hours
        float hours;
        cout << "Hours: ";
        cin >> hours;

        // test the function
        cout << "computePay(" 
            << "wage:" << wage << ','
            << "hours:" << hours << ") = "
            << computePay(wage, hours) << endl;
    }
}
#endif // TEST_COMPUTE_PAY
```

Figure 15.7:
Driver code added to test
computePay()

When this driver function is written, we only need to define TEST_COMPUTE_PAY and then call computePayTest() to run through our test cases again.

Best Practice 15.3 When possible, put driver code in a separate file.

It is usually a good idea to put driver code in a separate file so there is no chance that it could find its way into customer-facing code.

Automation

Running through a large suite of test cases can be tedious. It is time consuming, easy to make mistakes, and difficult to log efficiently. If only there were a way to make this process easier! Automation is the process of using software to simplify the process of testing programs. Here, a function or a program runs through a collection of test cases.

Automation is a function used to automatically execute test cases

Remember our compute pay example from earlier in the chapter? Instead of asking the user to manually run through all the test cases, we can simply feed the test cases into the function under test and validate the return value with an assertion. The programmer is then notified of an error if anything works unexpectedly.

The diagram shows a C++ code snippet for automating the `computePay()` function. The code is enclosed in a blue box and labeled "C++". It includes an `#ifdef DEBUG` block containing a comment block and a test function `void computePayTest()`. The test function contains several `assert` statements validating various wage and hours inputs. A handwritten note on the left says "One function validates all the test cases for the function under test". A handwritten note on the right says "Ensure the test code is separate from production code" and "Use an assert to validate the test cases".

```
C++
#ifndef DEBUG
/*
 * COMPUTE PAY TEST
 * Automation to exercise computePay()
 */
void computePayTest()
{
    // pay          wage   hours
    assert( 0.00 == computePay(0.00, 0.0)); // zeros
    assert( 5.00 == computePay(5.00, 1.0)); // one hour
    assert( 10.00 == computePay(5.00, 2.0)); // two hours
    assert(200.00 == computePay(5.00, 40.0)); // full time
    assert(207.50 == computePay(0.00, 41.0)); // overtime
    assert( 0.00 == computePay(0.00, 30.0)); // no wage
    assert( 0.00 == computePay(5.00, 0.0)); // no hours
}
#endif // DEBUG
```

Figure 15.8:
Automation to test
`computePay()`

A function with no side effects (does not rely on or change global state) should return the same value every time for a given input. This means we should be able to run the test cases in any order and the output of the tests should be the same every time the test function is invoked.

Best Practice 15.4 Create an automation function for every function in the project.

Every function should be tested before it is integrated into the project. The best way to do that is to enumerate all the test cases and encode the test cases in a driver function. This makes it possible to ensure that all the test cases are checked against the production code.

Best Practice 15.5 Run the automation often.

Automation is useful to verify that a newly written function works as expected. It is also useful to verify that a function still works after it has been modified. In other words, production code frequently needs to be augmented and reworked long after it was originally written. Automation can be used to verify that these changes do not break the original functionality. By running the automation frequently, we can be assured that any new change made to the codebase does not adversely impact the original functionality of the program.

Test Runner

A codebase written with a large amount of automation has many benefits. First, it provides assurances to the programmer that all the functions in the project work as expected. Second, it provides assurances than changes made to a given function will not introduce a subtle and difficult to find bug. These assurances can only be made if the automation is run frequently. The best way to encourage developers to run a project's automation is to create a test runner.

A test runner is a function used to invoke all the automation in a project. This function is debug-only; it is not compiled in the release version of the project. Usually, the test runner is executed immediately after the program begins before any production code is executed. This gives the programmer immediate feedback if any recent change has inadvertently broken something.

Test runners are functions used to execute all the automation in a project

The diagram shows a C++ code snippet for a test runner. It starts with a preprocessor directive `#ifdef DEBUG`, followed by a multi-line comment block containing the text `* TEST RUNNER`. Below the comment is the function definition `void test_runner()`. Inside the function, there are two sections of code: one for financial tests and one for account ledger tests. A handwritten note on the left says *Each automation function is run, one at a time*, with an arrow pointing to the first section of code. Another handwritten note on the right says *Ensure the test code is separate from production code*, with an arrow pointing to the function definition. At the bottom, there is a line of code `#endif // DEBUG` and a handwritten note *There could be hundreds of functions called here* with an arrow pointing to it. There is also a line of code `... code removed for brevity ...` with an arrow pointing to it.

```
C++
#ifndef DEBUG
*****
* TEST RUNNER
*****
void test_runner()
{
    // Test the financial code.
    compute_pay_test();
    compute_income_tax_test();
    compute_sales_tax_test();
    compute_compound_interest_test();

    // Test the account ledger.
    account_update_test();
    account_checking_test();
    account_savings_test();
    account_credit_test();

    ... code removed for brevity ...
}
#endif // DEBUG
```

Figure 15.9:
Test runner facilitating the execution of many automated tests

The final step of the test runner process is to call it from `main()`. This code is only executed in the debug build of the project; steps need to be taken to ensure that it is not in the production version.

The diagram shows a C++ code snippet for the `main()` function. It starts with the declaration `int main()`. Inside the function, there is a preprocessor directive `#ifdef DEBUG`, followed by a call to the `test_runner()` function. Below the call, there is another preprocessor directive `#endif // DEBUG`. A handwritten note on the left says *This must be debug only*, with an arrow pointing to the `#ifdef DEBUG` directive. Another handwritten note on the right says *Before any production code is run, the test runner is executed*, with an arrow pointing to the `test_runner()` call. There is also a line of code `... code removed for brevity ...` with an arrow pointing to it.

```
C++
int main()
{
#ifdef DEBUG
    test_runner();
#endif // DEBUG

    ... code removed for brevity ...
}
```

Figure 15.10:
Test runner is called from the beginning of `main()`

Many projects require more sophisticated test runners than this, allowing the user to customize which tests are run and to provide centralized logging of the results. Please see Chapter 25 Quality: Unit Test for more details on this.

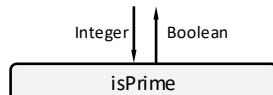
Examples

Example 15.1: Manual Driver Function

This example will demonstrate how to create a manual driver function.

Problem

Create a driver program to facilitate testing the `isPrime()` function:



Solution

Since this is a manual driver, we will be prompting the user for input to the function. Our function takes one input, so a single prompt will be used. Also, we wish to execute more than one test case so we will surround the prompt and the output statement in a `while` loop.

Note that drivers should be debug-only. Since Java does not have pre-compiler directives, we will surround the driver code in a static constant called `DEBUG`. This will inform the interpreter that the code is to be skipped when `DEBUG` is set to false.

```
Java
class Prime {
    private static final boolean DEBUG = false;

    public void isPrimeTest() {
        if (DEBUG) {
            Scanner scanner = new Scanner(System.in);

            while (true) {
                System.out.print("> ");
                int input = scanner.nextInt();
                Boolean output = isPrime(input);
                System.out.println("\t" + output);
            }
        }
    }

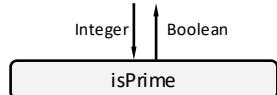
    ... code removed for brevity ...
}
```

Example 15.2: Automation

This example will demonstrate how to create an automation driver function.

Problem

Create a driver program to facilitate testing the `isPrime()` function:



Solution

Our automation will check the special values, the first several prime numbers, the first several non-prime numbers, then some larger prime numbers and larger non-prime numbers.

```
C
#ifndef DEBUG
void isPrimeTest() {
    /* check the special cases */
    assert(0 == isPrime(1)); /* one is not prime */
    assert(0 == isPrime(0)); /* zero is not prime */
    assert(0 == isPrime(-4)); /* negative numbers are not */
    assert(0 == isPrime(-5)); /* negative numbers are not */

    /* check the first several prime numbers */
    assert(1 == isPrime(2));
    assert(1 == isPrime(3));
    assert(1 == isPrime(5));

    /* check the first several composite numbers */
    assert(0 == isPrime(4)); /* 2 x 2 */
    assert(0 == isPrime(6)); /* 2 x 3 */
    assert(0 == isPrime(30)); /* 2 x 3 x 5 */

    /* big prime numbers */
    assert(1 == isPrime(61));
    assert(1 == isPrime(239));
    assert(1 == isPrime(7789));

    /* big composite numbers */
    assert(0 == isPrime(51)); /* 3 x 17 */
    assert(0 == isPrime(759)); /* 3 x 11 x 23 */
    assert(0 == isPrime(3357)); /* 3 x 3 x 373 */
}
#endif /* DEBUG */
```

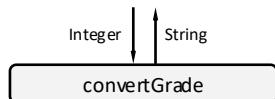
An alternative way to test this is to create a list of prime numbers and compare the list against the rest of `isPrime()`. This would be a comprehensive test but would be difficult to debug if an error was found. We often need to offset test readability against test comprehensiveness.

Example 15.3: Automation

This example will demonstrate how to create an automation driver function.

Problem

Create a driver program to facilitate testing the `convertGrade()` function:



Solution

Run through six sets of test cases: grades without the + or -, grades with the +, grades with the -, the special case that there is no A+, F- or F+, and the error cases.

```
VB
#If DEBUG Then
Sub convertGradeTest()
    ' Test with no + or -
    Debug.Assert("F" = convertGrade(55))
    Debug.Assert("D" = convertGrade(63))
    Debug.Assert("C" = convertGrade(74))
    Debug.Assert("B" = convertGrade(85))
    Debug.Assert("A" = convertGrade(96))

    ' Test +
    Debug.Assert("D+" = convertGrade(67))
    Debug.Assert("C+" = convertGrade(78))
    Debug.Assert("B+" = convertGrade(89))

    ' Test -
    Debug.Assert("D-" = convertGrade(60))
    Debug.Assert("C-" = convertGrade(71))
    Debug.Assert("B-" = convertGrade(82))

    ' F has no + or -
    Debug.Assert("F" = convertGrade(59))
    Debug.Assert("C" = convertGrade(38))
    Debug.Assert("B" = convertGrade(1))

    ' A has no +
    Debug.Assert("A" = convertGrade(99))
    Debug.Assert("A" = convertGrade(100))

    ' Errors
    Debug.Assert("error" = convertGrade(-1))
    Debug.Assert("error" = convertGrade(101))
End Sub
#End If
```

Note that we could have performed an exhaustive test of every possible integer, but it would be more difficult to interpret and take longer to execute.

Exercises

Exercise 15.1: Types of Drivers

In your own words, define the main types of drivers.

Type	Definition
Ad Hoc	
Commandeer	
Driver Function	
Automation	
Test Runner	

Exercise 15.2: Scenarios

In each of the following scenarios, describe the type of driver that would best be utilized.

Scenario	Driver Type
I am writing a new function and, before I get it to work, I would like to write some code that will be used to manually exercise it	
Before beginning work on a given function, I would like to call an existing driver function once to make sure everything works	
I would like to execute all the automation before I release my code to the client	
I would like to add some quick code to validate one small part of a function before I write the rest	
I would like to encode all my test cases in a single function so I can easily verify that this function works as expected	

Exercise 15.3: Type of Driver

What type of driver is utilized in the highlighted code?

Pseudocode

```
absoluteValue(number)
    IF number > 0
        RETURN number
    ELSE
        RETURN -number

main()
    WHILE true
        PROMPT for number
        GET number
        PUT absoluteValue(number)

... rest of the program ...
```

Exercises 15.4: Fact or Fiction

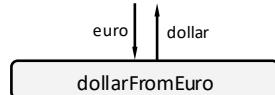
For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Ad hoc drivers are very complicated and expensive to implement	
Driver functions should not exist in shipping code; only debug code!	
Automation is a form of manual driver	
A test runner executes many automation driver functions	
The commandeer technique only works with manual drivers	

Problems

Problem 15.1: Dollar from Euro

Consider a function that converts euros to dollars, from Problem 14.1.



This function honors the following exchange rate:

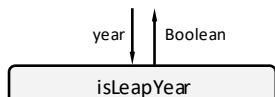
€1.00 = \$1.13

Please do the following:

1. Implement this function in the programming language of your choice.
2. Create a list of test cases for this function. Use what you created for Problem 14.1 if you previously completed that problem.
3. Create a **manual** driver function to facilitate testing this function. Test this function with your test cases.
4. Create an **automated** driver function to run through your test cases.

Problem 15.2: Is Leap Year

Consider a function that determines if a given year is a leap year (from Problem 02.3, Example 05.2, and Problem 07.3).

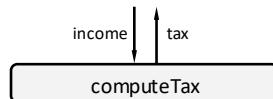


Please do the following:

1. Implement this function in the programming language of your choice.
2. Create a list of test cases for this function. Use what you created for Problem 14.3 if you previously completed that problem.
3. Create a **manual** driver function to facilitate testing this function. Test this function with your test cases.
4. Create an **automated** driver function to run through your test cases.

Problem 15.3: Compute Tax

Consider a function that computes tax (from Problem 07.1 and Problem 14.4).



Taxes are computed the following way:

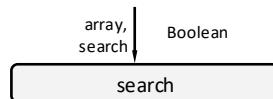
Income range	Tax is
\$0 – \$15,100	10% of amount over \$0
\$15,100 – \$61,300	\$1,510 plus 15% of amount over \$15,100
\$61,300 – \$123,700	\$8,440 plus 25% of amount over \$61,300
\$123,700 – \$188,450	\$24,040 plus 28% of amount over \$123,700
\$188,450 – \$336,550	\$42,170 plus 28% of amount over \$188,450
\$336,550 – no limit	\$91,043 plus 35% of amount over \$336,550

Please do the following:

1. Implement this function in the programming language of your choice. You may have done this already in Problem 07.1.
2. Create a list of test cases for this function. Use what you created for Problem 14.4 if you previously completed that problem.
3. Create a **manual** driver function to facilitate testing this function. Test this function with your test cases.
4. Create an **automated** driver function to run through your test cases.

Problem 15.4: Search

Consider a function takes determines whether a given number is in an array. The first parameter is a sorted array of values. The second parameter is the search value, also an integer. The function will return TRUE if the number is found and FALSE otherwise.



Please do the following:

1. Implement this function in the programming language of your choice. You may have created the pseudocode for this in Challenge 02.1.
2. Create a list of test cases for this function. Use what you created for Problem 14.5 if you previously completed that problem.
3. Create a **manual** driver function to facilitate testing this function. Test this function with your test cases.
4. Create an **automated** driver function to run through your test cases.

Challenges

Challenge 15.1: Insertion Sort

Consider the following function which sorts numbers using the insertion sort algorithm:

Pseudocode

```
sortInsertion(array, n)
    FOR iPivot ← n - 2 ... 0
        valuePivot ← array[iPivot];
        iInsert ← binarySearch(array, valuePivot,
                               iPivot + 1, n - 1)
        iInsert--
        FOR iShift ← iPivot ... iInsert
            array[iShift] ← array[iShift + 1]
        array[iShift] ← valuePivot
```

Please do the following:

1. Implement this function in the programming language of your choice.
2. Create a list of test cases for this function. Use what you created from Challenge 14.1 if you previously completed that problem.
3. Create a **manual** driver function to facilitate testing this function. Test this function with your test cases.
4. Create an **automated** driver function to run through your test cases.

Challenge 15.2: Fibonacci

Consider the following function to compute a Fibonacci number:

Pseudocode

```
fibonacci(n)
    IF n ≤ 1
        RETURN n
    ELSE
        RETURN fibonacci( n - 1 ) + fibonacci( n - 2 )
```

Please do the following:

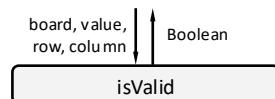
1. Implement this function in the programming language of your choice. If you completed Challenge 08.1: Fibonacci, please use the algorithm you developed there.
2. Create a list of test cases for this function. Use what you created from Challenge 14.2 if you previously completed that problem.
3. Create a **manual** driver function to facilitate testing this function. Test this function with your test cases.
4. Create an **automated** driver function to run through your test cases.

Challenge 15.3: Sudoku

Consider the game of Sudoku:

Sudoku is a numbers game played on a 9x9 grid. The object of the game is to fill in the 9x9 grid while honoring certain constraints: 1) there is no more than one instance of a given number on a given row. 2) there is no more than one instance of a given number on a given column. and 3) there is no more than one instance of a given number on an inside square (the 3x3 squares embedded in the 9x9 grid).

We are particularly interested in the function which determines whether a given value is legal in a given square on the grid:



Please do the following:

1. Implement this function in the programming language of your choice.
2. Create a list of test cases for this function. Use what you created for Challenge 14.3 if you previously completed that problem.
3. Create a **manual** driver function to facilitate testing this function. Test this function with your test cases.
4. Create an **automated** driver function to run through your test cases.

Recursion

Chapter 16

Recursion is a problem-solving technique where a solution is reached by subdividing a large problem into smaller ones. These smaller problems are then fed back through the original function.

Recursion is the process of a function calling itself to compute a result

Recursion is the programming technique characterized by a function calling itself in the process of computing a result. At first, this may seem counterintuitive: if a function cannot compute the answer without help, how can calling itself get it any closer? It seems like the blind leading the blind! Fortunately, recursion does not work like that.

The key to understanding recursion is to recognize that there are two steps. The first step is to define a stopping condition, usually with a trivial input or an input that is of a very small size. The second step is to subdivide the larger problem into smaller problems, each of which is closer to the stop condition. These smaller problems are then fed back to the recursive function. In this way, every call to the function brings the algorithm closer to the stopping condition.

To learn more about recursion, please see Chapter 16: Recursion

Figure 16.1:
A simple example of recursion

Pseudocode

```
add(value1, value2)
  IF value1 = 0
    RETURN value2
  ELSE
    RETURN 1 + add(value1 - 1, value2)
```

Notice how add calls itself to arrive at the solution

Every loop algorithm can be turned into a recursive algorithm and vice versa

Recursion has the same computational power as loops. In fact, every looping algorithm can be turned into a recursive algorithm and vice versa. Some languages, particularly functional programming languages, do not even have looping structures! With these languages, all looping algorithms must be represented recursively.

Designing Recursive Algorithms

Designing recursively begins with understanding the mechanics of recursive algorithms (the end-condition and the progress step). There are also other considerations, such as using head vs. tail recursion and creating wrappers.

Recursion's Two Parts

In most looping problems, there are two fundamental questions: how do you know when to stop and how do you progress towards the solution? These two steps are also present in every recursive algorithm: the end-condition and the progress step.

Part	Description
End-Condition	How the function knows that it has found the solution. Typically, the end-condition is trivial – when the return value for a given input is obvious
Progress Step	The process of taking the larger problem and breaking it into smaller pieces, each of which are closer to the end-condition

The first order of business of any recursive algorithm design is to identify these two parts.

End-Condition

The end-condition is also known as the base case, the trivial case, or the stopping condition. It is usually defined as input of zero size or input with a trivial outcome. It could also be an empty string or a parameter with a NULL value. If we were to write a recursive algorithm to perform addition, the identity property of addition might come in handy:

$$x + 0 = x$$

From here, we can see that if the right parameter is zero, then just return the left parameter. Note that we could do it the other way around just as well.

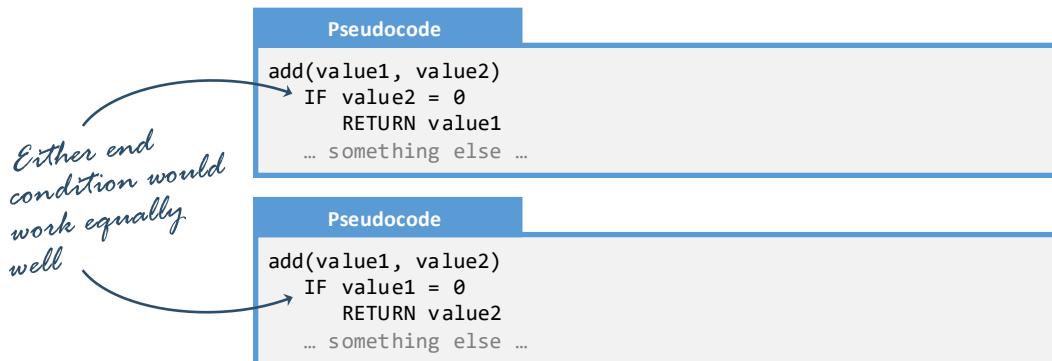
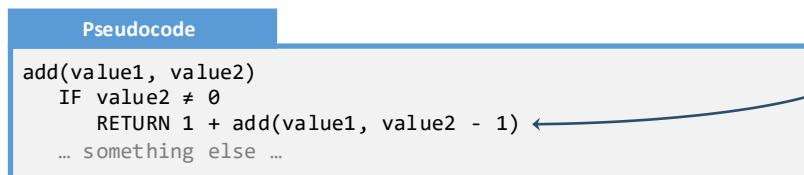


Figure 16.2:
Two different versions of
an end-condition

Progress Step

The progress step is also known as the recursive case. The goal here is to get the input closer to the end-condition. In the above example where the end-condition is the zero value for `value2`, we would want to decrease `value2` with every iteration.

Figure 16.3:
An example of a
progress step



Notice how
value2 gets
smaller with
every recursive
call

Putting it All Together

Let's get back to our addition example. We started with the need to add two numbers together:

```
Pseudocode  
add(value1, value2)  
RETURN value1 + value2
```

Figure 16.4:
A non-recursive version
of an add function

Now let's say, for the sake of argument, that we do not have the plus operator defined for all numbers. In fact, we can only add by one and subtract by one. This forces us to write a loop:

```
Pseudocode  
add(value1, value2)  
sum ← value1  
WHILE value2 ≠ 0  
    sum ← sum + 1  
    value2 ← value2 - 1  
RETURN sum
```

Figure 16.5:
An iterative version
of an add function

This means that $\text{add}(7, 3)$ is turned into a sequence of adding 1s:

$$\text{add}(7, 3) \xrightarrow{\text{yields}} 7 + 1 + 1 + 1$$

We have two steps in our solution, the stop condition and the progress step:

$$\begin{aligned} \text{add}(\text{value1}, 0) &\xrightarrow{\text{yields}} \text{value1} \\ \text{add}(\text{value1}, \text{value2}) &\xrightarrow{\text{yields}} 1 + \text{add}(\text{value1}, \text{value2} - 1) \end{aligned}$$

Our recursive solution is the combination of these two steps:

```
Pseudocode  
add(value1, value2)  
IF value2 = 0  
    RETURN value1  
ELSE  
    RETURN 1 + add(value1, value2 - 1)
```

Progress step

Figure 16.6:
A recursive version
of an add function

To verify our solution, we will perform the derivation:

$$\begin{aligned} \text{add}(7, 3) &\xrightarrow{\text{yields}} 1 + \text{add}(7, 2) \\ &\xrightarrow{\text{yields}} 1 + (1 + \text{add}(7, 1)) \\ &\xrightarrow{\text{yields}} 1 + (1 + (1 + \text{add}(7, 0))) \\ &\xrightarrow{\text{yields}} 1 + (1 + (1 + 7)) \\ &\xrightarrow{\text{yields}} 10 \end{aligned}$$

Head and Tail Recursion

When a recursive function gets called, there are two distinct phases: 1) when all the function calls get added to the call-stack, and 2) when the function returns and the call-stack is emptied. With recursion, we have the choice: do we do most of the work when the call-stack is being emptied, or when it is being built? These are called head recursion and tail recursion respectively.

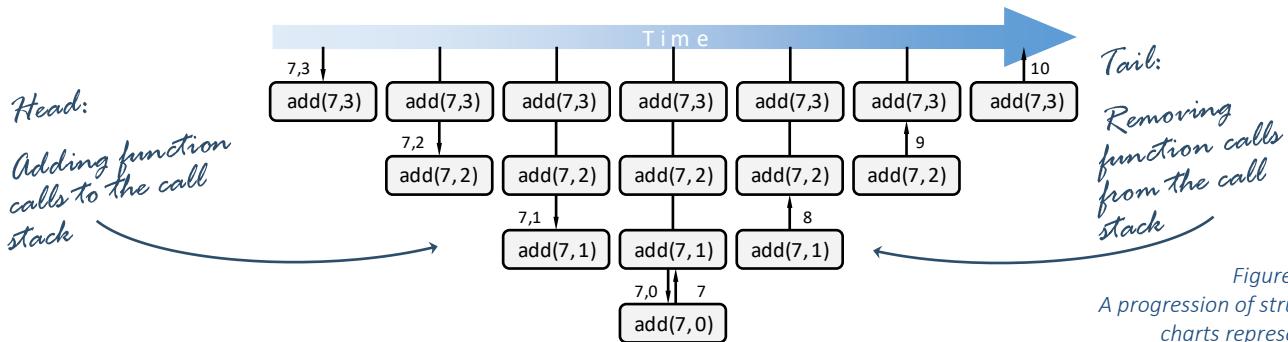


Figure 16.7:
A progression of structure charts representing several recursive calls.

Head Recursion

Head recursion first builds the call-stack and then does the computation as the functions are returned. This is characterized by operations done on the result of the recursive call. By the time the end-condition is reached, no work has yet been done. In other words, all the heavy lifting is accomplished when the call-stack is unwound.

Pseudocode

```
add(value1, value2)
    IF value2 = 0
        RETURN value1
    ELSE
        RETURN 1 + add(value1, value2 - 1)
```

We compute the sum after the recursive call is done

Figure 16.8:
Head recursion

Tail Recursion

Tail recursion performs the most expensive operation in the parameters being passed into the recursive function. By the time the end-condition is reached, the solution is found and the only remaining task is to unwind the call-stack. In other words, the recursive call is the last thing executed by the function.

Pseudocode

```
add(value1, value2)
    IF value2 = 0
        RETURN value1
    ELSE
        RETURN add(value1 + 1, value2 - 1)
```

We compute the sum before the recursive function call is made

Figure 16.9:
Tail Recursion

In most cases, tail recursion is faster. Many compilers can recognize tail recursion and optimize the code to return out of the functions faster. The compiler may recognize that no work is done after the function call is made so it will not bother to save the local variables in the call-stack, thereby cutting most of the function call overhead.

Using a Wrapper

A recursive function needs to pass all the information necessary to not only compute the solution, but know where it is in the recursive process. This sometimes requires us to use a more complex interface than the client expects. We can avoid this problem with a wrapper. A wrapper is a trivial function whose only purpose is to translate one interface to another. To illustrate this point, consider the following function used to sum the values in an array using a traditional loop.

Python

```
def average(numbers) -> float:  
    for number in numbers:  
        sum += number  
    return sum / len(numbers)
```

Figure 16.10:
An iterative version of a
function averaging the
numbers in an array

If we wished to turn this into a recursive function, then we would need to pass a smaller version of the `numbers` array with every function call. This would require copying the entire array! To avoid that, we will pass an `index` parameter.

Python

```
def sum_recursive(numbers, index) -> float:  
    if index >= len(numbers):  
        return 0.0  
    return numbers[index] + sum_recursive(numbers, index + 1)
```

Figure 16.11:
A recursive function
summing the numbers
in an array

Now we have a partial solution. There are two problems, however. The first is that it will only work if we initially call the function with `index = 0`. Anything else will cause the function to skip values or worse! The second problem is that we have not divided the result by the number of elements. We can address both problems with a wrapper:

Python

```
def average(numbers) -> float:  
    return sum_recursive(numbers, 0) / len(numbers)
```

Figure 16.12:
A wrapper function for the
averaging the numbers
in an array

Notice that this has two benefits: it hides the interface with an extra parameter and it also completes the processing that cannot be done recursively.

Wrappers can also be used to minimize the number of recursive iterations. Going back to our add example, swapping the parameters yields very different iteration counts:

$$\begin{array}{ccc} \text{add}(7, 3) & \xrightarrow{\text{yields}} & 7 + 1 + 1 + 1 \\ & \xrightarrow{\text{yields}} & 3 + 1 + 1 + 1 + 1 + 1 + 1 \end{array}$$

Pseudocode

```
add(value1, value2)  
    IF value1 > value2  
        RETURN add_recursive(value1, value2)  
    ELSE  
        RETURN add_recursive(value2, value1)
```

Since we are
recursing on the
second value, it
should be the
smaller of the two

Figure 16.13:
A wrapper function
optimizing the order of
the parameters

Debugging Recursive Algorithms

Debugging recursive algorithms can be more difficult than with iterative algorithms. The first problem is that with each recursive call, we get a different version of the local variables with potentially different values. It is very difficult to tell which copy we are seeing at a given moment in execution. Second, it is difficult to represent the return value in the function which usually plays a critical role in the recursive process. Fortunately, we have two techniques to help us deal with this complexity: functional decomposition and a special form of program trace.

Functional Decomposition

Functional decomposition is the process of substituting a function call for the results of a function. With a traditional function call, this is a one-step process. With recursion, there are many: one step is required for every recursive call. To illustrate this process, consider the following function displaying the contents of a C string:

C++

```
std::ostream & display(const char * text) {
    if (*text)
        return cout << *text
                  << display(text + 1);      // progress step
    else
        return cout << endl;           // end condition
}
```

Figure 16.14:
A recursive function
displaying the contents of
a string

We wish to see what will happen if we call the function with the C string "Hello". We will do this by starting with the original function call `display("Hello")` which is actually `display({'H', 'e', 'l', 'l', 'o', '\0'})`. We will then replace that function with the first recursive call. The next instance of `display()` will be replaced with second recursive call. This process is continued until the end-condition is reached.

Functional Decomposition

```
display({'H', 'e', 'l', 'l', 'o', '\0'});
cout << 'H' << display({'e', 'l', 'l', 'o', '\0'});
cout << 'H' << 'e' << display({'l', 'l', 'o', '\0'});
cout << 'H' << 'e' << 'l' << display({'l', 'o', '\0'});
cout << 'H' << 'e' << 'l' << 'l' << display({'o', '\0'});
cout << 'H' << 'e' << 'l' << 'l' << 'o' << display({'\0'});
cout << 'H' << 'e' << 'l' << 'l' << 'o' << endl;
```

As we progress down the decomposition, the input should decrease in size or progress towards the end-condition. In this case, the end-condition is when the null character is reached at the end of the C string. We can see this is happening with our `display()` function because the input string is smaller with each function call.

Of course, function decomposition is only practical with simple input and a small number of recursive calls. It does, however, illustrate how the recursive process moves through the input until the end-condition is reached.

Trace

Program trace is difficult with recursive functions because it is necessary represent the step in the recursive process. Fortunately, there is a way to accomplish this. Since recursion maintains state exclusively through the input parameters, we can capture the state by adding a column representing the input parameters. Note that it is often helpful to add an additional column to represent the return value of the function since it is often computed in the return statement. Let's go back to our recursive add function:

Figure 16.15:
A recursive function adding
two values

```
Kotlin
fun add(value1: Int, value2: Int): Int {
    if value2 == 0 {
        return value1 // end condition
    } else {
        value1++ // progress step
        value2-- // using tail
        return add(value1, value2) // recursion
    }
}
```

19
20
21
22
23
24
25
26
27

We will trace this with `add(7, 3)`.

The diagram illustrates the execution of a recursive function `add(7, 3)`. On the left, a vertical brace indicates the sequence of calls, labeled "Head, building the call-stack" for the initial call and "Tail, unwinding the call-stack" for the return from the recursive call. On the right, a trace table shows the state of variables over 15 steps. The columns are labeled "params", "value1", "value2", and "return". The "return" column is only filled in the final step (step 27). The trace shows the progression of the function: it starts at (7, 3), moves to (8, 2), (9, 1), and (10, 0), then returns to (9, 1), (8, 2), and finally (7, 3). Handwritten annotations on the right side of the table label the steps: "First progress step" for the first recursive call, "Second progress step" for the second, "Third progress step" for the third, and "End Condition" for the final return.

	params	value1	value2	return
19	(7,3)	7	3	
20		7	3	
24		8	2	
25		8	2	
19	(8,2)	8	2	
20		8	2	
24		9	1	
25		9	1	
19	(9,1)	9	1	
20		9	1	
24		10	0	
25		10	0	
19	(10,0)	10	0	
20		10	0	
21		10	0	
27	(9,1)	10	0	10
27	(8,2)	9	1	10
27	(7,3)	8	2	10

Figure 16.16:
Program trace of a
recursive function

Notice in the above trace table how we can only “see” the version of the local variables that is present in the current scope. Once we make the recursive call, the old version of the local variables is saved on the call stack. Also notice (in line 27) that when we return from a recursive call, our old values are restored to what they were before the function call.

When to Use Recursion

There are many cases where the recursive version of an algorithm is less complex, easier to understand, and even more efficient than the iterative version of the algorithm. When choosing where to use recursion, there are a few things to consider: the computation cost of recursion, the conceptual cost of recursion, and the many recursion pitfalls which should be avoided.

Computational Cost of Recursion

When a function call is made, it is necessary for the compiler to store all the caller's local variables so they can be reinstated when the callee is finished. It is also necessary to store the location in the program where execution will resume when the function call is done. These elements are stored in a structure called a call-stack.

The call-stack is a data structure maintained by the compiler to store program state across function calls

The call-stack is a data structure maintained by the compiler with the assistance of the CPU and the operating system which stores the program state. All local variables are stored in the call-stack. When a function call is made, a special CPU instruction is invoked which stores several critical pieces of information in the stack (the stack frame pointer, the return address, the parameters, and the caller's local variables). The cost of this operation is directly related to the amount of data in the local variables and the parameters that need to be stored in the call-stack. Most CPUs can accomplish this in a few dozen clock cycles, depending on the specific CPU architecture, the available memory, and the needs of the programming languages. From this, notice: 1) it takes time to call a function, and 2) there is a limited amount of space on the call stack.

Generally, recursive solutions are less performant than iterative solutions. Furthermore, because the call-stack is not of infinite size, it is possible to run out of space. Python places a limit on the call stack size. This is determined by the `sys.setrecursionlimit()` function which sets the maximum depth of the stack to prevent infinite recursion.

Generally, recursive solutions are less performant than iterative solutions

Best Practice 16.1 Do not use recursion in performance-critical areas of the application

When pursuing a recursive solution, make sure the number of recursive calls is carefully managed. $O(\log n)$ applications (such as binary search functions or recursive sorting algorithms) are typically acceptable. Algorithms making $O(n)$ function calls should not use recursion unless the size is n is very small.

Conceptual Cost of Recursion

Some problems are easy to conceptualize recursively. This often is related to how the individual was introduced to the concept or how it is commonly communicated. Generally, however, even experienced programmers do not instinctively think

Most find it easier to think iteratively than recursively

recursively. Recursive solutions are typically more difficult to understand than iterative solutions and are therefore not as maintainable. This should be considered when choosing recursion.

Common Pitfalls

There are several pitfalls which can trip up programmers new to recursion.

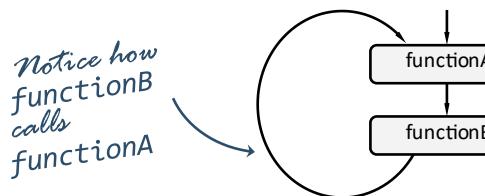
Best Practice 16.2 Avoid multiple recursion

Multiple recursion is when there are more than one recursive variables or there is more than one recursive function call. When not managed carefully, this can lead to $O(2^n)$ algorithmic efficiency which is intractable in all but the simplest cases. Recall from Chapter 03 that the following function is $O(2^n)$ (please see the section titled “ $O(2^n)$ Efficiency” in Chapter 03).

```
Ruby
def fibonacci(n)
    n <= 1 ? n : fibonacci( n - 1 ) + fibonacci( n - 2 )
end
```

Figure 16.17:
Multi-recursion

Best Practice 16.3 Avoid indirect recursion



Direct recursion is when **FunctionA()** directly calls **FunctionA()**. Indirect recursion, also known as mutual recursion, is when **FunctionA()** calls **FunctionB()** which, in turn, calls **FunctionA()**. In these cases, it is not always apparent that recursion is at play. They are difficult to understand and difficult to debug.

Figure 16.17:
Indirect recursion

Best Practice 16.4 Make progress towards the end-condition with every iteration

If your algorithm does not work towards the end-condition with every iteration, you will have an infinite loop. Since there is a finite amount of space on the call-stack, this will quickly result in all the memory being used and the program will crash. If possible, write debug code to ensure that the problem size is smaller with every successive call.

Best Practice 16.5 Be wary of unnecessary or unintentional recursion

Programmers new to using functions often utilize recursion without realizing what they are doing. For example, consider the event loop for a turn-based game. Rather than including a loop in the interaction loop, some programmers are tempted to call it again through recursion. The result is a call-stack which grows the larger the longer the game is played.

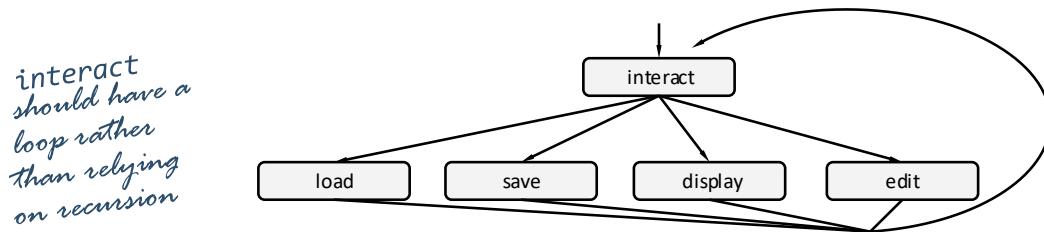


Figure 16.18:
Unintentional recursion

Examples

Example 16.1: List Traversal

This example will demonstrate how to turn a loop algorithm into a recursive algorithm.

Problem

Consider the following code summing all the values in an account register to determine the account balance. Initially, this is performed using an iterative approach.

C++

```
double computeBalance(double transactions[], int num)
{
    double balance = 0.00;                      // start with $0.00

    for (int i = 0; i < num; i++)                // iterate through
        balance += transactions[i];             //    all transactions

    return balance;
}
```

Find a recursive solution to this problem.

Solution

The end-condition is an empty list (when the number of transactions are zero). In this case, the sum is zero dollars, which will be the return value.

The progress step is to add the value at the end of the list to the sum of all the values before it in the list. Notice that we have a function to compute the sum of all the values before it! This is where recursion comes in.

C++

```
double computeBalance(double transactions[], int num)
{
    if (num == 0)                                // stop condition
        return 0.00;
    else
        return transaction[num - 1] +           // progress step
            computeBalance(transactions, num - 1);
}
```

Example 16.2: Factorial

This example will demonstrate how to turn a recursive algorithm into an iterative algorithm.

Problem

Consider the following recursive function computing factorial. The mathematical definition of factorial is the following:

$$n! = \begin{cases} 1, & n = 0 \\ n \times (n - 1)!, & n \geq 1 \end{cases}$$

Our recursive solution is the following:

```
VB
Function factorial(ByVal n As Integer) As Integer
    If n = 0 Then
        Return 1                                ' end condition
    Else
        Return n * factorial(n - 1)            ' progress step
    End If
End Function
```

Solution

Note how the end-condition is well defined: $0! = 1$. We will use this for the end-condition of our loop. The progress step is also well defined: $n!$ is defined in terms of $(n - 1)!$. Our recursive solution uses end-recursion. This means that $5! = 5 \times 4 \times 3 \times 2 \times 1$. What if we took advantage of the commutative property of multiplication and defined $n!$ as $(n - 1)! \times n$. This would give us $5! = 1 \times 2 \times 3 \times 4 \times 5$. From here, our iterative solution should be clear.

```
VB
Function factorial(ByVal n As Integer) As Integer
    Dim value As Integer = 1                      ' end condition

    For number As Integer = 1 To n                ' progress step
        value *= number
    Next
    Return value
End Function
```

Example 16.3: Decimal to Binary

This example will demonstrate how to predict the output of a recursive algorithm using the functional decomposition method.

Problem

Consider the following recursive solution to compute the binary values for a decimal number:

```
Swift
func binaryFromDecimal(_ decimal: Int) -> String {
    if decimal > 1 {
        return binaryFromDecimal(decimal / 2) +
            String(decimal % 2)
    } else {
        return "0"
    }
}
```

Create a functional decomposition for two values: 7 and 42.

Solution

We will use `b()` instead of `binaryFromDecimal()` for brevity's sake. First, we expect 7 decimal to equal 0111 binary.

$$\begin{array}{rcl} b(7) & \xrightarrow{\text{yields}} & b(3) + 1 \\ & \xrightarrow{\text{yields}} & b(1) + 1 + 1 \\ & \xrightarrow{\text{yields}} & b(0) + 1 + 1 + 1 \\ & \xrightarrow{\text{yields}} & 0 + 1 + 1 + 1 \end{array}$$

Next, we expect 42 decimal to yield 0101010 binary.

$$\begin{array}{rcl} b(42) & \xrightarrow{\text{yields}} & b(21) + 0 \\ & \xrightarrow{\text{yields}} & b(10) + 1 + 0 \\ & \xrightarrow{\text{yields}} & b(5) + 0 + 1 + 0 \\ & \xrightarrow{\text{yields}} & b(2) + 1 + 0 + 1 + 0 \\ & \xrightarrow{\text{yields}} & b(1) + 0 + 1 + 0 + 1 + 0 \\ & \xrightarrow{\text{yields}} & b(0) + 1 + 0 + 1 + 0 + 1 + 0 \\ & \xrightarrow{\text{yields}} & 0 + 1 + 0 + 1 + 0 + 1 + 0 \end{array}$$

Example 16.4: Decimal to Binary

This example will demonstrate how to predict the output of a recursive algorithm using the program trace method.

Problem

Consider the following recursive solution to compute the binary values for a decimal number:

Swift

```
func binaryFromDecimal(_ decimal: Int) -> String {
    var binary: String;
    if decimal >= 1 { // A
        binary = binaryFromDecimal(decimal / 2) + // B
            String(decimal % 2)
    } else { // C
        binary = "0"
    }
    return binary; // D
}
```

Create two traces for the values 7 and 42.

Solution

We expect 7 decimal to equal “0111” binary and 42 decimal to equal “0101010” binary.

	param	decimal	binary
A	(7)	7	
B		7	+“1”
A	(3)	3	
B		3	+“1”
A	(1)	1	
B		1	+“1”
A	(0)	0	
C		0	“0”
D	(0)	0	0
D	(1)	1	01
D	(3)	3	011
D	(7)	7	0111

	param	decimal	binary
A	(42)	42	
B		42	+“0”
A	(21)	21	
B		21	+“1”
A	(10)	10	
B		10	+“0”
A	(5)	5	
B		5	+“1”
A	(2)	2	
B		2	+“0”
A	(1)	1	
B		1	+“1”
A	(0)	0	
C		0	“0”
D	(0)	0	0
D	(1)	1	01
D	(2)	2	010
D	(5)	5	0101
D	(10)	10	01010
D	(21)	21	010101
D	(42)	42	0101010

Exercises

Exercise 16.1: Define Recursion

In your own words, define “recursion” in a way that a non-technical person could understand it. Hint: do not use the word “function.”

Exercise 16.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
All recursive solutions need to have an end-condition	
Tail recursion does most of the work as functions are removed from the call-stack	
Multi-recursion is generally a good programming practice	
Any iterative algorithm can be turned into a recursive algorithm	
Recursive solutions are generally as fast as iterative solutions	
Indirect recursion is generally a good programming practice	

Exercise 16.3: Name the Algorithm

What would be the best name for the following recursive algorithm?

Pseudocode

```
function_16_3(x, y)
    IF x = 1
        RETURN y
    ELSE
        RETURN y + function_16_3(x - 1, y)
```

Exercise 16.4: Name the Algorithm

What would be the best name for the following recursive algorithm?

Pseudocode

```
function_16_4(x, y)
    IF x = 0
        RETURN 1
    ELSE
        RETURN y * function_16_4(x - 1, y)
```

Exercise 16.5: Name the Algorithm

What would be the best name for the following recursive algorithm?

Pseudocode

```
function_16_5(x, y)
    IF x = 0
        RETURN y
    ELSE
        RETURN 1 + function_16_5(x - 1, y)
```

Problems

Problem 16.1: Find Highest

Given the following algorithm written as a loop, generate a recursive solution using a programming language of your choice.

Pseudocode

```
findHighest(array, numElements)
    highest ← array[0]
    FOR i ← 1 ... numElements
        IF array[i] > highest
            highest ← array[i]
    RETURN highest
```

Problem 16.2: Sum of Powers

Given the following algorithm written as a loop, generate a recursive solution using a programming language of your choice.

Pseudocode

```
sumPowersOfTwo(numberPowers)
    sum ← 0
    power ← 1
    FOR i ← 1 ... numberPowers
        sum += power
        power *= 2
    return sum
```

Problem 16.3: Display Names

Given the following algorithm written as a loop, generate a recursive solution using a programming language of your choice.

Pseudocode

```
displayNames(names)
    FOREACH name in names
        PUT name
```

Problem 16.4: Sum the Digits in a Number

Given the following recursive algorithm, generate a loop solution using a programming language of your choice.

Pseudocode

```
sumTheDigits(number)
    IF number = 0
        RETURN 0
    ELSE
        RETURN number % 10 +
            sumTheDigits(number / 10)
```

Problem 16.5: Reverse a String

Given the following recursive algorithm, generate a loop solution using a programming language of your choice.

Pseudocode

```
reverseString(text)
    length ← text.length()
    IF length = 0
        RETURN ""
    ELSE
        RETURN text[length - 1] +
            reverseString(substring(text, 0, length - 1))
```

Problem 16.6: Fibonacci

Consider the following recursive algorithm computing the Fibonacci sequence:

Pseudocode

```
fibonacci(number)
    if number = 0
        RETURN 0
    if number = 1
        RETURN 1
    RETURN fibonacci(number - 1) + fibonacci(number - 2)
```

Generate a loop solution using a programming language of your choice.

Hint 1: An array might be a good tool to store values.

Hint 2: Can you find a way to do this with a two-element array?

Problem 16.7: Sum the Digits

Consider the following recursive algorithm summing the digits in a number:

Pseudocode

```
sumTheDigits(number)
  IF number = 0
    RETURN 0
  ELSE
    RETURN number % 10 +
      sumTheDigits(number / 10)
```

01
02
03
04
05
06

Using this algorithm', compute the values returned by `sumTheDigits(13)` and `sumTheDigits(521)` using the program trace method.

Problem 16.8: Reverse a String

Consider the following recursive algorithm that reverses a string:

Pseudocode

```
reverseString(text)
  length ← text.length()
  IF length = 0
    RETURN ""
  ELSE
    RETURN text[length - 1] +
      reverseString(substring(text, 0, length - 1))
```

01
02
03
04
05
06
07

Using this algorithm, compute the values returned by `reverseString("az")` and `reverseString("1234")` using the program trace method.

Problem 16.9: Fibonacci

Consider the following recursive algorithm computing the Fibonacci sequence:

Pseudocode

```
fibonacci(number)
  if number = 0
    RETURN 0
  if number = 1
    RETURN 1
  RETURN fibonacci(number - 1) + fibonacci(number - 2)
```

Using this algorithm, compute the values returned by `fibonacci(5)` and `fibonacci(6)` using the functional decomposition method.

Challenges

Challenge 16.1: Search

Consider the following search algorithm written as a loop:

```
Pseudocode
search(array, search, iBegin, iEnd)
    while iFirst ≤ iLast
        iMiddle ← (iBegin + iEnd)/2

        IF array[iMiddle] = search
            RETURN TRUE

        IF search > array[iMiddle]
            iFirst ← iMiddle + 1
        ELSE
            iLast ← iMiddle - 1
        RETURN FALSE
    END
```

First, generate a recursive solution in the programming language of your choice.

Second, based on the two solutions (the loop solution written in pseudocode and the recursive one you just completed), compare and contrast the solutions. Which is better from an efficiency and maintainability perspective?

Challenge 16.2: Greatest Common Divisor

The greatest common divisor between two numbers is the largest positive integer that divides the two numbers without a remainder. Euclid developed an algorithm to describe this:

$$gcd(a, b) = \begin{cases} a, & b = 0 \\ gcd(b, remainder(a, b)), & b > 0 \end{cases}$$

First, generate a recursive solution in the programming language of your choice.

Second, generate an iterative solution in the programming language of your choice.

Third, compare and contrast the two solutions. Which is better from an efficiency and maintainability perspective?

Chapter 17 Top-Down

Top-down design is the process of describing generally the entire system, then subdividing it into components, then into subcomponents, and then continuing until the smallest component can be described in a programming language.

Designing an algorithm or a single function can be challenging. However, designing an entire program can be much more difficult. How do you even know what functions you will need? Imagine being asked to write a large application like a 3D video game, a personal finance manager, or a mobile application to help someone manage lists. Where do you begin? Without help, this process can be very intimidating. Fortunately, there is a process to help us get started: top-down design.

A top-down design begins by specifying the entire system and then dividing it into successively smaller subcomponents

The top-down design process was commonly used by many programmers since the invention of stored instruction computers but was first formally described by Edsger Dijkstra. Top-down design goes by many names, including step-wise design, divide and conquer, step-wise refinement, decomposition, structured programming, and reductionism. Regardless of the name, the process is the same: one begins by broadly defining the entire system and then dividing the system into successively smaller subcomponents. When finished, the programmer has an outline for the entire program.

This outline or master plan is an essential step when building an application of substantial size. Imagine building a house without knowing the number of bedrooms or the location of the kitchen. Just hammering nails into boards is unlikely to result in a house you want to live in. The same is certainly true with software. A good master plan is a critical first step, even though we all realize that the plan will change as the project is developed.

Top-Down Design Process

There are several terms or concepts which are central to the top-down design process:

Term	Definition
System	The software application we are building.
Component	A part of the larger system.
SubComponent	A component that is part of another component.
Base Element	A component lacking subcomponents.
Relation	The connection between components.

The top-down design process is a design algorithm where the system is described generally, then the main components are identified, and the subcomponents are identified, and so on until base elements are described. This algorithm is inherently recursive. The stop condition is when we reach a base element for all branches in

the tree. The progress step is that each subcomponent is more specific and detailed than the one above it.

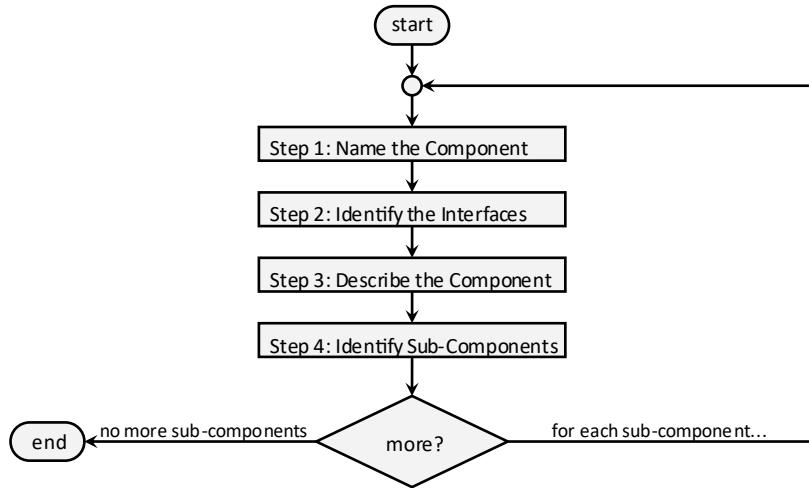


Figure 17.1:
Flowchart of the
top-down design
process

The output of the top-down design process is a structure chart or a data flow diagram (or some combination of both). This is a tree-like structure, though the tree is upside down in typical computer-science fashion. The root of the tree (the topmost node, called “Level 0”) is `main()`, the function that represents the entirety of the program. This is the first component described. From `main()` are several subcomponents, being the functions that `main()` calls directly. These nodes on the tree are at Level 1. This continues until functions are described which have no subcomponents. These form the bottom of the tree and are called base elements.

Note that Level 0 is the most abstract function of the entire program. It contains very few specifics, relying on their higher-numbered levels to provide details. Generally speaking, each successive level provides more details than the level above it.

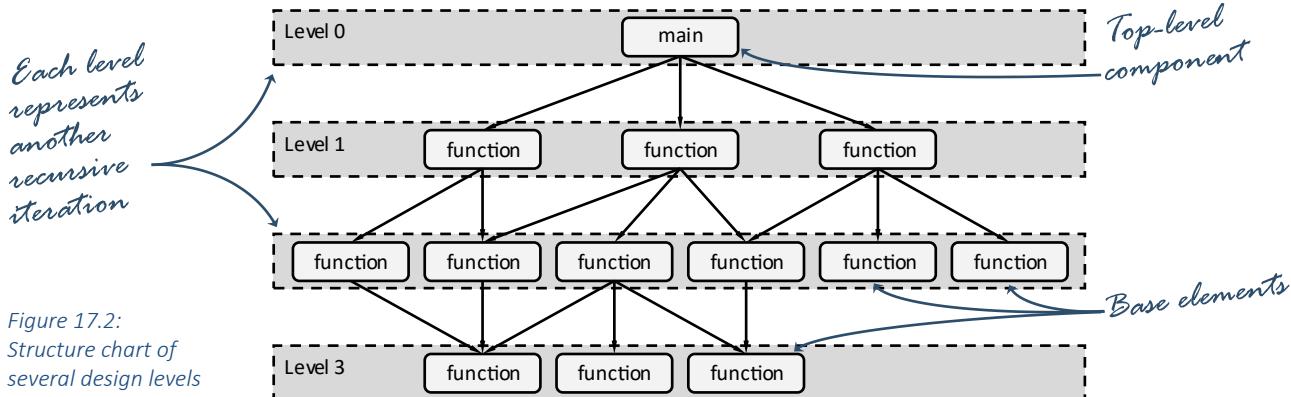


Figure 17.2:
Structure chart of
several design levels

Step 0: The System is a Component

The missing step from the top-down design flowchart is the realization that the system itself is a component. It happens to be the highest-level and most complex component in the design, but it is a component nonetheless. We start the recursive design process with the `main()` function, which represents the entire system.

Step 1: Name the Component

The adage goes, “if you can’t name it, you don’t understand it.” This applies to all levels of software design. In the case of top-down design, naming a component or function is the first step towards understanding its purpose.

Design for cohesion in Step 1

When working on a name, it is important to ask yourself cohesion questions: does this component do one thing and one thing only? In other words, is this component cohesive? We cannot fully identify the cohesion level of the component at this stage of the design process, but we can get early indications of cohesion errors. If we find it difficult to name the component or if we cannot satisfactorily answer cohesion questions, then we have a problem with the design. It is much easier to fix design errors at this stage of the process than later, when actual code has been written.

Figure 17.3:
Structure chart after
Step 1



Figure 17.4:
DFD after Step 1

When finished with Step 1 for a given component, we should have a named node in a structure chart or a named process element in a DFD. This node or process element should, without exception, be strongly cohesive.

Step 2: Identify the Interfaces

Once the name of the component is identified and we have a high-level understanding of what the component is to do, the next step is to identify the interfaces. The inputs into the component need to be listed and characterized. We also need to describe the output. The input and output represent the contract which this new component is meant to fulfill.

Design for coupling in Step 2

With the interfaces identified, we can start to ask coupling questions. As with cohesion, it is best to identify coupling challenges at this stage of the process. They are much easier to fix here than after code is written. Is it possible to simplify the interface? Would it be better to use a different data type or to send less data through the system? Some of these questions might require us to rework substantial parts of the design. This is expected and is not to be considered a setback!

Figure 17.5:
Structure chart after
Step 2



Figure 17.6:
DFD after Step 2

When we are finished with Step 2 for a given component, all the arrows into and out of the named node (in a structure chart) or process element (in the DFD) should be labeled with the interface specifics. All these interfaces should be trivial, encapsulated, or simple. If complex, document, or interactive coupling is required, there needs to be a very good reason to do this. Note that there should never be superfluous coupling at this stage of the design process.

Step 3: Describe the Component

With the component named and the interfaces identified, it is now time to figure out how the component will work. Everything in this step occurs inside a single function, so all that we learned about algorithm design applies in this step of the top-down design process.

This is the right time in the design process to start asking algorithmic efficiency questions and maintainability questions. It is also the right time to think about adding asserts, identifying test cases, and put some thought as to what drivers can be built to simplify the testing process. With the purpose of the function known (Step 1) and the interfaces known (Step 2), we have all the knowledge necessary to fully design a given function.

Figure 17.7:
Flowchart after Step 3

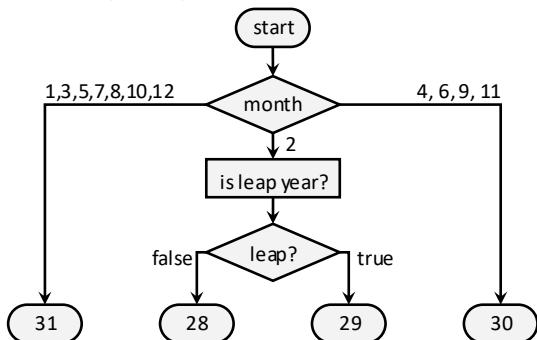


Figure 17.8:
Pseudocode after Step 3

```
Pseudocode  
numDaysInMonth(month, year)  
days = 31, 28, 31,  
30, 31, 30,  
31, 31, 30,  
31, 30, 31  
  
IF month = 2 AND a leap year (year)  
RETURN 29  
ELSE  
RETURN days[month]
```

At very least, Step 3 results in a few sentences describing how the function is to be built. It is more common, however, to have pseudocode or flowcharts describing the algorithm formally. Since there is no room for algorithm design in a DFD or a structure chart, we usually attach the algorithm description, flowchart, or pseudocode in a separate diagram with a link or note to it.

Step 4: Identify Subcomponents

The final step of the design process is to identify the subcomponents that a given component may need to complete its task. While finishing this component, we will only have a vague idea about the subcomponent. In fact, we can consider them to be “black boxes:” things that just work. These things will be worked out when we iterate on it. Even such basic things such as its name (Step 1) and interfaces (Step 2) will be worked out later.

Figure 17.9:
Structure chart after Step 4

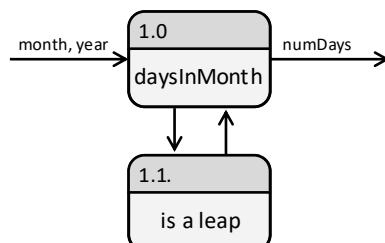


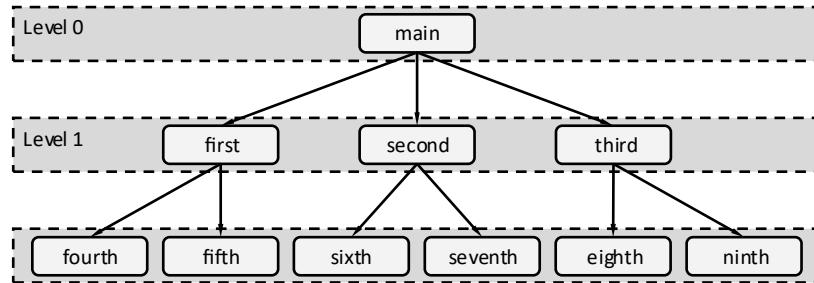
Figure 17.10:
DFD after Step 4

It is essential to be on the lookout for duplicate components. If you have already described a component that is nearly the same, see if you can reuse that aspect of the design to avoid duplication.

Note that most components have many subcomponents. This means that each level in the structure chart is much bigger than the one above it. To put things into perspective, if each component called 4 subcomponents on average, then level 1 would consist of 4 functions, level 2 would consist of about 16, level 3 would consist of 64, and level n would consist of 4^n . Of course, the tree does not increase infinitely. Frequently a component calls a subcomponent that was already described earlier in the tree. Also, the design process stops when all the functionality required of the client has been described. However, a problem remains: how does one keep track of all these yet-to-be-described components when going through the top-down design process?

Breadth-first design works out the details for everything in Level 1 before starting on Level 2

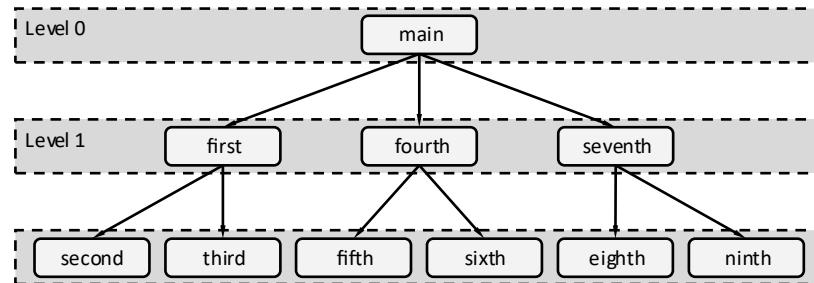
at a traffic light) of subcomponents that are yet to be described. In Step 4 of the design process, add these subcomponents to the end of the queue and, in Step 1 when the process begins anew, take the first one off the head of the queue and start working on it. Following this process will guarantee that all Level 1 functions are described before the Level 2 ones are looked at.



*Figure 17.11:
Breadth-first design,
completing Level 1
before starting Level 2*

Depth-first design works out the details under one branch of Level 1 before working on another branch

Another method is called depth-first design. Here, you fully explore the first subcomponent (with all its children) of Level 1 before looking at the second subcomponent at Level 1. This is done in a similar way to the breadth-first design. At Level 0, you create a list of all the sub elements. When you add a new element to the list, you add it to the head of the list, not the back. This creates a data structure called a stack (first-in, last-out, behaving much like the dishes in your cupboard).



*Figure 17.12:
Depth-first design,
completing everything
under "first" before
starting "fourth"*

It does not matter if you pursue depth-first or breadth-first search when doing a top-down design. It only matters that the component is described before the subcomponent.

Issues and Best Practices

Top down design is a useful technique applicable in a wide variety of circumstances. Certainly, it is a technique that all programmers should have in their tool bag. However, there are some pitfalls that could beset you if you are not careful.

Issue: Impossible problems

When we follow the top-down design process, we often specify that a component needs to do some task before we understand the task itself. By the time we get around to working on that component, we may discover that it is impossible. Perhaps the component requires information which we cannot provide. Perhaps the task itself is difficult or beyond our coding ability. In other words, we have painted ourselves into a corner.

Best Practice 17.1 When you find yourself painted into a corner, back out and start again

We often paint ourselves into a corner because we did not understand the problem well enough to when we started. This is expected and a normal part of the development process. You are never as ignorant about a project as you are right now. When you start a project, you only have a general idea of the client's needs and how the technology will work. When you finish a project, every nuance of the client's needs has been explored and you have worked out every detail of the design. It is helpful to look at the entire software development process as a learning endeavor.

Once we recognize that we are faced with an impossible problem, we should back out one or two levels in the design until the source of the problem is discovered. Hopefully, only a handful of functions will have to be redesigned. Perhaps more. Regardless of the ramifications, no actual code has been written so the cost of reworking parts of the design is not too high. Note that the thrown-away work is not a waste! The second pass through, armed with the hard-earned experience of a deadend design, will probably yield a much better design.

When parts of the design need to be thrown away, it is not wasted work. You learned something in the process which will make the next version better

Issue: Top-down Design Helps with Verbs, Not Nouns

Up to this point, we have been working with algorithms and functions. These, fundamentally, deal with verbs. They do not describe how data are organized. It turns out that many of the most interesting and difficult design problems center around nouns.

Object-oriented design is a development methodology focusing on nouns

Designing for the nouns of a system requires a completely different set of tools and a completely different mindset. We call this "object-oriented design." We will learn about the tools of object-oriented design (called class diagrams), the metrics of object-oriented design (fidelity, robustness, convenience, abstraction, adaptability, alignment, and redundancy), and quality techniques (unit tests, test-driven development, and debuggers) in the next two units.

Issue: Testing is Difficult

When we follow the top-down design process with rigor, not a single line of code is written until the entire project is designed. This puts all testing activities at the tailend of the development process. Note that if testing is put off too long in the development process, then it becomes difficult or impossible to make real quality assurances to the client.

Best Practice 17.2 Top-down is for design activities, bottom-up is for development activities

The design process should be top-down, working from the general to the specific. The process of writing the code, however, should be bottom-up. We will discuss bottom-up development next chapter.

Issue: Top-down Works Best for Smaller Projects

Many large and complex projects do not lend themselves to top-down designs. The reason for this is the ignorance issue discussed earlier (you are never as ignorant about the project as you are right now). With large or gargantuan problems, the amount of unknown vastly exceeds the amount that is known. This means it is naive and unhelpful to completely design a project before the code is written: there is a high

Top-down design assumes a complete and correct set of requirements

probability that everything will be thrown away and you will need to start over. In other words, top-down design assumes that a complete and correct set of requirements are available at the beginning of the project. This assumption is false in all but the simplest projects.

In the early days of software engineering (up until the mid-1990s), it was common practice for developers to expect complete and correct requirements at the onset of any coding project. They would then build software to meet these needs and deliver it to the client (those who sponsored the project they were working on). Often, the resulting software was not fit for use due to some unrealized client need. “I built what you told me to build,” the programmer would complain. “But it is not what I need,” would be the reply of the client.

Best Practice 17.3 Large projects should be built iteratively with heavy involvement from the client

Modern software is built-in an iterative fashion with close involvement of the client. The client should be readily available to answer questions as they come up. Periodically, parts of the system should be shown to the client to make sure things are going in the right direction. This helps maximize the chance that what is built is what the client needs. If we go too far down the design process without client involvement, then we are likely relying on a false assumption about what we think the client needs.

Modern software is built iteratively, with close involvement of the client

Best Practice 17.4 Always have a master plan, and be prepared to adjust it

The project should always have a master plan as to what the final code will look like. It is understood that this master plan is a draft, subject to change. The more code is shown to the client, the more the client is able to make an informed decision as to whether the software will meet the needs. When the inevitable design change is required, the master plan needs to be updated so it always reflects the latest thinking on the part of the design team. In other words, it is a living document.

The design master plan should be a living document, constantly being updated as more is learned about the project

Stubs

A stub is a placeholder for a forthcoming replacement promising to be more complete. In the context of designing and building software, a stub is a tool enabling us to put placeholders in the codebase for all the functions in our structure chart. We can do this even when we do not yet know how the functions will be implemented.

How to Use Stubs

Perhaps the best way to explain how to use stubs is demonstrate it. Consider a small part of a structure chart with the function `numDaysInMonth()`.

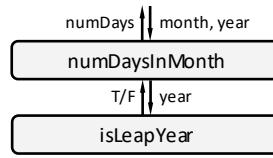


Figure 17.13:
Structure chart for
`numDaysInMonth()`

Based on this structure chart fragment, we only know the parameters into the function (month and year) as well as the return value (number of days). A stub for this function would be the following:

```
Ruby
def numDaysInMonth(month, year)
    # Just a stub for now
    return 31
end
```

Figure 17.14:
Stub for
`numDaysInMonth()`

Notice that this is little more than a placeholder for now. It compiles and runs, but will not return correct output for all input. Creating stubs can be accomplished in several steps:

1. Create a trivial function definition using the name from the structure chart or DFD.
2. If there is a return type, return something trivial or typical. It doesn't matter what is returned, as long as the function compiles.
3. If the function calls another function, then make sure the stub calls it in a trivial way.

The third step is the one that often trips up new programmers. A structure chart fully implemented with stubs will call all the functions in the program so the functions can be implemented in any order. This means our above stub is not quite complete.

```
Ruby
def isLeapYear(year)
    return true
end

def numDaysInMonth(month, year)
    isLeapYear(1)
    return 31
end
```

Figure 17.15:
Stub for
`isLeapYear()`

Benefits of Stubs

Stubs are useful tools because they allow us to convert a structure chart into working code, allow us to figure out data flow, always have the project in a state that compiles and executes, and allows us to implement the functions of the program in any order we choose.

Structure Chart into Working Code

You can look at a structure chart as analogous to an outline for an essay. In this analogy, the stubs would be the process of putting the outline's headings in an actual document under which the content of the essay will flow.

Stubs are the bridge between an abstract design and a functioning program

Stubs are the bridge between an abstract design and a functioning program. They allow the programmer to codify the design sketched on a sheet of paper or on a white board. When a structure chart has been converted into stubs, a working program exists that compiles and executes. The programmer next needs to fill in the functions to finish the application.

Figure Out Data Flow

One of the challenges of writing a large application is to figure out how data flows through the program. Of course, a DFD is designed to help the programmer visualize this process. When a program design has been converted into stubs (complete with parameter lists and return values), defects in the design become more evident. If, for example, two functions expect a parameter of the same name but of different data types, this defect will not be evident in a DFD or a structure chart. However, when you attempt to compile the stub, such issues are obvious.

The Project Always Compiles

One of the great benefits of using stubs is that the program is nearly always in a state where it compiles and runs. Gone are the days when hundreds of lines of code are

Gone are the days of multiple pages of compile errors

written and then, just when the programmer thinks the project is complete, the first compilation is attempted. No one likes to see page after page of compile errors!

When a structure chart or a DFD is converted to stubs, it nearly always compiles on the first try because there is so little code written. Then, as only one function at a time is implemented, the project remains at a state where it compiles or has just a few errors.

Implement Functions in Any Order

With a program completely stubbed out, the programmer is free to implement the functions in any order he or she chooses. Imagine a programmer with a dozen functions still stubbed out. If only 10 minutes are available to work on the project, then a small and easy function can be implemented in that time. If a larger block of

With a program fully stubbed out, even small blocks of time can be filled with useful work

time becomes available, then a more ambitious function can be chosen. The programmer does not need to find a large block of time to work on the project because everything is nicely subdivided into small bite-size chunks.

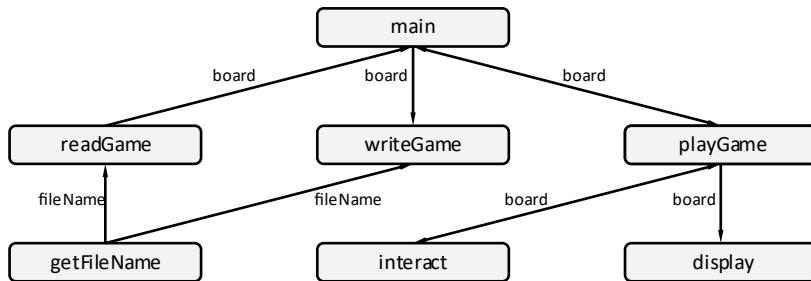
Examples

Example 17.1: Stubs from Structure Chart

This example will demonstrate how to turn a structure chart into stubs.

Problem

Create stub functions for the following structure chart for a Sudoku program.



Solution

There are seven functions, and each function should be represented in the stubs below. Note that if one function calls another, that function call should be present.

```
C
int main() {
    int board[9][9];
    readGame(board);
    playGame(board);
    writeGame(board);
}

void readGame(int board[9][9]) {
    char fileName[256];
    getFileNames(fileName);
}

void playGame(int board[9][9]) {
    interact(board);
    display(board);
}

void interact(int board[9][9]) {}

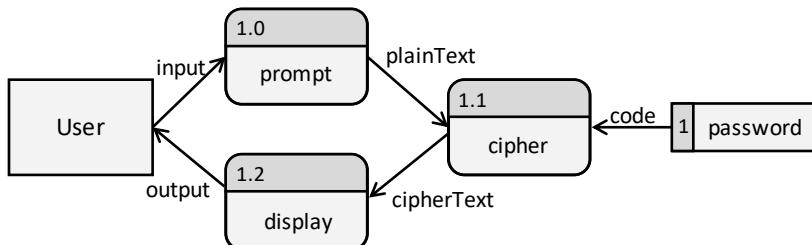
void display(int board[9][9]) {}
```

Example 17.2: Stubs from DFD

This example will demonstrate how to turn a DFD into stubs.

Problem

Create stub functions for the following DFD for an encrypting program:



Solution

Every element in a structure chart is a function. The same is not true with DFDs. Not only are some things variables (such as the password item), but even process elements might not be functions. For example, the prompt and the display process elements are actually Python statements. In fact, aside from the `main()` function, the only other function in this DFD is the `cipher()` processing element.

```
Python
def main():
    plainText = prompt("")
    cipherText = cipher(plainText)
    print(cipherText)

def cipher(plainText):
    return plainText
```

For now, `cipher()` is stubbed out to return the plaintext message back to the caller. This is not a very secure cipher; do not use it to store your confidential information.

Example 17.3: Top-Down Design

This example will demonstrate how to perform the first iteration of design for a mildly complex application.

Problem

Perform the first iteration of top-down design for a program to play chess:

Chess is a two-person game played on an 8x8 grid where the players arrange their pieces on opposite sides of the board. There are six types of pieces: pawn, knight, bishop, rook, queen, and king. Each type is governed by a different set of rules. The players alternate turns until one player checkmates another, until one player gives up, or the same position on the board is repeated three times.

Solution

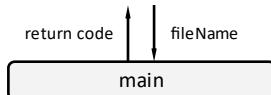
We will begin by calling the entire game a component. We will start with that component.

Step 1: Name the component.

In this case the component is simply called `main()`.

Step 2: Identify the Interfaces.

After a brief discussion with our client, it becomes apparent that we will need to pass a filename as a parameter so the user can double-click on a saved game to load it. Also, most languages allow the program to return a code indicating whether it was successful.



Step 3: Describe the Component.

Pseudocode

```
main(filename)
  IF filename
    readFile(filename)

    play game (game)

    IF !filename
      get file name (filename)
      writeFile(filename)
```

Step 4: Identify the Subcomponents

From our pseudocode, it is apparent that there are subcomponents. These are the following:

- Read the file into a board
- Write game from a board into a file
- Play game
- Get the file name from the user

Example 17.4: Top-Down Design

This example will demonstrate how to perform the second iteration of design for a mildly complex application.

Problem

Perform the second iteration of top-down design for a program to play chess.

Solution

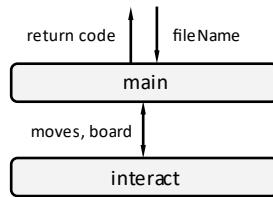
The function we will be working on is “play the game.”

Step 1: Name the component.

This function is the main interactive element, allowing users to play the game and is a central feature of the turn-based game. Candidate names include `play()`, `interact()`, and `mainLoop()`. Of these, `interact()` seems to be the less ambiguous.

Step 2: Identify the Interfaces.

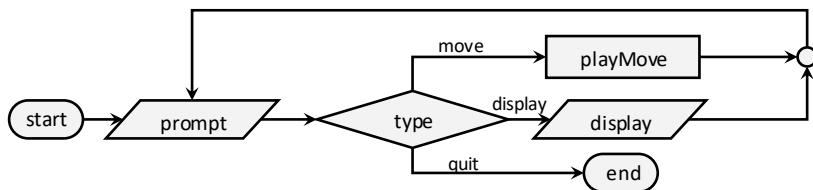
`Interact` will accept the board from `main` as well as the list of moves. It will also return an updated board and an updated list of moves.



The coupling for `interact()` is complex because each move in the `moves` array has nontrivial syntax, and the same can be said for `board`. In Unit 2, we will learn how to encapsulate these to improve the coupling.

Step 3: Describe the Component.

This main loop will have to allow for multiple interactions as well as carry out the user’s intention.



The cohesion of `interact()` is strong because the function does one thing and one thing only; all the complex tasks are delegated to subcomponents.

Step 4: Identify the Subcomponents

From our pseudocode, it is apparent that there are the following subcomponents:

- Prompt for a move
- Play the move which updates the board and records the move in the history
- Display the game

Exercises

Exercise 17.1: Top-Down Steps

For each of the following steps in the top-down design process, name it and give a brief description of what happens during that step.

Step Number	Name and Description
Step 0	
Step 1	
Step 2	
Step 3	
Step 4	

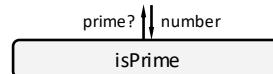
Exercise 17.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
The top-down process is recursive	
A stub is a placeholder for a function to be written later	
Bottom-up is for design (not development) activities	
Design for cohesion in Step 2 of the process	
Larger project should be built iteratively involving the client	
Top-down is for development (not design) activities	

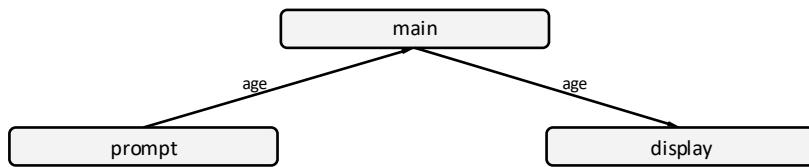
Exercise 17.3: Stub from Structure Chart

In the programming language of your choice, create a stub for `isPrime()` in the following DFD:



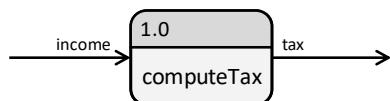
Exercise 17.4 Stub from Structure Chart

In the programming language of your choice, create a stub for `main()` in the following structure chart:



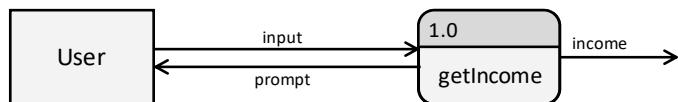
Exercise 17.5: Stub from DFD

In the programming language of your choice, create a stub for `computeTax()` in the following DFD:



Exercise 17.6 Stub from DFD

In the programming language of your choice, create a stub for `getIncome()` in the following DFD:



Problems

Problem 17.1: Tic-Tac-Toe

Consider the game Tic-Tac-Toe.

The game of Tic-Tac-Toe is played in a 3x3 grid where players alternate putting Xs and Os in the squares. The game is won when there are three Xs or three Os in a horizontal, vertical, or diagonal row. The game allows the players to store and retrieve their game from a file. When a game is won or when all the squares are filled with an X or O, then the game is over and the program begins anew with an empty board.

For the game Tic-Tac-Toe, please do the following:

1. Create a structure chart and pseudocode using the top-down design process
2. For each function, identify the level of cohesion
3. For each function, identify the level of coupling
4. Create stubs for your design in the programming language of your choice

Problem 17.2: Calendar

Consider the following problem definition.

The calendar application places a calendar on the screen corresponding to a given month and year. The program will first prompt the user for the month and year, making sure the user input is within the accepted range (year greater than 1753 and month between 1 and 12). The program will then compute how many days there are in the month and the day of the week that the month begins on. The latter is computed by counting the number of days since 1/1/1753, which is a Monday. Therefore, if the number of days is evenly divisible by 7, then it is a Monday.

For the calendar application, please do the following:

1. Create a structure chart and pseudocode using the top-down design process
2. For each function, identify the level of cohesion
3. For each function, identify the level of coupling
4. Create stubs for your design in the programming language of your choice

Problem 17.3: MadLib

Consider the game MadLib.

Mad Libs is a word game where one player prompts another for a list of words to substitute for blanks in a story; these word substitutions have a humorous effect when the resulting story is then read aloud. The program will prompt the user for the file that describes the Mad Lib, and then prompt him for all the substitute words. When all the prompts are complete, the program will display the completed story on the screen.

For the MadLib application, please do the following:

1. Create a structure chart and pseudocode using the top-down design process
2. For each function, identify the level of cohesion
3. For each function, identify the level of coupling
4. Create stubs for your design in the programming language of your choice

Problem 17.4: Sudoku

Consider the following problem definition.

Sudoku is a numbers game played on a 9x9 grid, subdivided into nine 3x3 inside squares. The game begins when a board is loaded from a file. The user can specify the number in a given location on the board by specifying the coordinates (such as "D3") and the value. If a number does not fit, then an error is presented to the user and the board remains unchanged. This board consists of several numbers between 1 and 9 on the 9x9 grid with many spaces left blank. The objective of the game is to fill in the blanks so the Sudoku constraint is met. The Sudoku constraint is that there is no more than one instance of every number on each row, column, or inside square.

For the Sudoku application, please do the following:

1. Create a structure chart and pseudocode using the top-down design process
2. For each function, identify the level of cohesion
3. For each function, identify the level of coupling
4. Create stubs for your design in the programming language of your choice

Challenges

Challenge 17.1: Shopping List

A shopping list application is described in the following way:

The shopping list mobile application allows the user to create a shopping list and check off items when at the grocery store. The application can accommodate any number of shopping lists, each of which has a name, a short description, and the list itself. The user can add, duplicate, edit, checkoff, and delete shopping lists. When the program begins, the user is presented with the current collection of shopping lists. Each list has a duplicate, edit, use, and delete icon. At the bottom of the list is an “add list” icon.

This shopping list application should consist of a couple dozen functions. For this application, please do the following:

1. Create a structure chart and pseudocode using the top-down design process
2. For each function, identify the level of cohesion
3. For each function, identify the level of coupling
4. Create stubs for your design in the programming language of your choice

Challenge 17.2: Credit Card Transactions

A personal finance application is described in the following way:

The personal finance application allows the user to store credit card transactions. When the application begins, the current account information is loaded and the user is presented with an option of entering a transaction, reconciling an account, or displaying a report. The “enter a transaction” option prompts the user for a single transaction (date, name, category, amount) and adds it to the account. The “reconcile” option allows the user to update the status of a transaction from “Open” to “Reconciled”. The “report” option will display a summary of all the transactions within a given time period, collected by category.

This credit card transaction application should consist of a couple dozen functions. For this application, please do the following:

1. Create a structure chart and pseudocode using the top-down design process
2. For each function, identify the level of cohesion
3. For each function, identify the level of coupling
4. Create stubs for your design in the programming language of your choice

Challenge 17.3: Maze

A maze game is described in the following way:

The maze game is a 2D game where the object is for a ship to navigate a maze without hitting a wall or running out of time/fuel. The game begins when a maze is loaded from a file and the ship is placed at the beginning. The user can turn the ship and apply thrusters. This game will obey the laws of motion, so in order to stop, the ship must turn around and apply counterthrust. The game is over when the user hits a wall, runs out of fuel, or reaches the end of the maze.

This maze game should consist of a couple dozen functions. For this application, please:

1. Create a structure chart and pseudocode using the top-down design process
2. For each function, identify the level of cohesion
3. For each function, identify the level of coupling
4. Create stubs for your design in the programming language of your choice

Bottom-Up

Bottom-up development is a code-writing process beginning with the smallest and simplest components, gradually working to the most complex and interconnected ones. Using this methodology, the final version of `main()` is often the last function written.

A common interaction between two Computer Science students goes something like this:

Sue: Are you making any progress on that large project due tomorrow?

Sam: Yes, it is almost done! I just need to compile it.

We all know what happens next. The first attempt at compiling the project yields several screens of errors. After an hour or two, it finally compiles. Unfortunately, it crashes immediately! A few more hours pass before the basic functionality works. It is somewhat resembles what is expected. Has this ever happened to you?

A bottom-up development minimizes the number of errors present in the project at a given moment in time

The bottom-up development methodology is a code-writing process designed to avoid the difficulty of shifting through large amounts of compile errors, finding bugs in large codebases, and working with unstable code. It involves implementing functions one at a time, starting with the simplest and gradually working to the most complex. At every stage of the process, functions are implemented, compiled, and tested before moving on to the next. The promises are that the project is always in a state where it compiles, errors can be easily localized to just a few dozen lines of code, and work can be separated into bite-size chunks.

Top-Down and Bottom-Up

Top-down is the process of working from the general and large down to the detailed and specific. Bottom-up starts with the most detailed and specific and gradually works towards the large and general. Which is the better practice? Well, it depends on your development activity. There are four activities that a software engineer does while working on a project of significant size:

Activity	Description
Requirements Elicitation	Learn what the client needs from the software
Design	Figure out how the software is to be built
Development	Actually writing the code
Testing	Finding defects and fixing bugs

Generally, bottom-up is preferred for development and testing, but not recommended as a design strategy.

Requirements elicitation is learning what the client needs from the system

The requirements elicitation activity involves learning what the client needs from the software. Seldom can the software engineer expect to get a complete and correct requirements document (called a Software Requirements Specification or SRS). Usually it involves many interactions, observations, and interviews with the various stakeholders. This is a very important and complex process. There are many books and research papers written on this process, far more material than can be discussed here. Though there are specialists who fulfill this role on many teams (called Requirements Engineers), most developers need to do this from time to time. Is it better to do requirements elicitation top-down or bottom-up?

Top-Down

Top-down requirements elicitation is the process of starting very general (“what is this system supposed to do?”) and gradually drilling down to the details. Only after understanding the system generally does one move down to the next level of questions. As with top-down design, this can be done breadth-first (answering all the Level 0 questions before moving on to Level 1 questions) or depth-first (understanding one subsystem before moving on to another).

A common technique employed in top-down requirements elicitation is “precision questioning.” This is a question-answer technique where the interviewer starts with a somewhat general topic and drills down on more specifics by asking “Why” and “How” questions. This process is continued until the interviewee either cannot answer the question or enough detail is gathered to create a requirement. When asking these “Why” or “How” questions, it is important to make them as specific as possible, usually a more detailed version of the previously provided answer.

Precision questioning is the process of asking questions that are increasingly specific to get to the heart of an issue

Top-down requirements elicitation can be very effective when the client has a holistic understanding of the system. This is especially true when he or she can see the major components and how they interact with each other.

Bottom-Up

Bottom-up requirements elicitation is the process of starting with very specific features or system components and gradually building them up to the overall system. Begin with a well-understood piece and ask the client how it is used. When that piece is fully described, ask questions about the subsystem in which the piece fits. It is important at this point to try to identify other pieces which may be part of this subsystem. Again, start small and work up.

One of the biggest challenges of bottom-up requirements elicitation is to help the interviewee visualize what part of the system you are talking about. It is often helpful to draw a rough diagram of the system as pieces are discovered and described. This technique is useful for clients who do not have the big picture or whose understanding of the system is somehow flawed or limited.

Best Practice 18.1 Use top-down or bottom-up elicitation depending on the needs of the client.

A good requirements engineer is skilled at both top-down and bottom-up processes. Choose whichever is best for the client.

Design

Software design is the process of gaining an understanding of the organization and functionality of the system before it is built. It is the place where all the high-level decisions are made, which is why many call this activity the most important of the four (requirements → design → implementation → testing).

Top-Down

The top-down methodology tends to produce the most elegant and cohesive designs

The top-down methodology tends to produce the most elegant and cohesive design. It allows for a deeper understanding of how components interoperate, and what they need to accomplish before specifying details.

The biggest drawback of top-down design is the tendency to get painted into a corner. If you overcommit to a high-level design decision before the details are understood, it is not uncommon to encounter a subcomponent that is very awkward or impossible to implement. When this happens, it is usually necessary to backtrack—revisit previously completed designs and refactor them to consider newly acquired knowledge. Even under the best circumstances, this can be a painful process.

Be willing to backtrack when you reach a design deadend

Bottom-Up

Bottom-up design is the process of designing small features first, then working up to larger components. Following the bottom-up design process, the last piece to be understood is the highest level: the system.

Bottom-up tends to result in “ball-of-mud” designs. Imagine a group of features, each individually well-designed but completely unrelated. When they are brought together, they tend to not fit well. The glue that must be created to make everything work well together gives the resulting system the impression of a ball of mud.

Overengineered solutions tend to be inefficient, expensive to build, and overly complex. Avoid when possible!

Another challenge of bottom-up design is not knowing the role a feature will play in the final system. Because the final system is largely unknown with bottom-up design, there is a tendency to overengineer or underengineer each component.

Overengineering is the process of building too much functionality, flexibility, or robustness. Unfortunately, this extra functionality, flexibility, and robustness come at price. Overengineered systems tend to be inefficient, expensive to build, and overly complex. Underengineering, by contrast, under-delivers on functionality, flexibility, or robustness. Here, the component does not meet the needs of system, often causing the system to fail. Ferdinand Porsche from whom the sports cars get their name once said, “the perfect race car crosses the finish line in first place and immediately falls into pieces.” Overengineering and underengineering are both detrimental to the overall system design quality and are difficult to avoid with bottom-up design.

Underengineered solutions are seldom fit for use

Best Practice 18.2 Top-down design is better than bottom-up in most scenarios

Though exceptions inevitably exist, it is almost always best to design top-down rather than bottom-up or any other strategy.

Development

Development is the process of converting a high-level design into working code. Though many see this as the core activity of the process, it would be impossible without requirements elicitation, design, and testing.

Top-Down

Top-down development is the process of starting at the highest-level function (usually `main()`) and gradually working down to leaf-level functions.

A problem with top-down development is that it is that high-level functions tend to change as their subcomponents are defined. In other words, it is often the case that interfaces need to change as a function is written.

Top-down development often results in expensive rewrites

This interface change requires the caller to adjust as well (perhaps also changing the caller's caller and so on), resulting in many expensive rewrites of high-level functions.

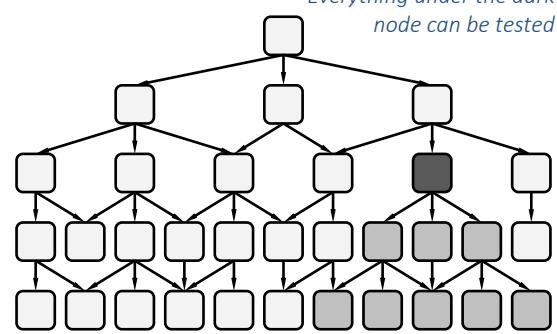
Another problem is that testing is postponed. Since `main()` and other high-level functions have all their subfunctions stubbed, they can't be tested until their dependencies are implemented. This does not happen until the end of the project. As a result, it is difficult to conduct meaningful testing until late in the development process.

Bottom-Up

Bottom-up development begins with the simplest and most independent function. This function is written and tested before working on the next simplest and most independent.

Bottom-up development avoids unnecessary rewrites because no function is written until the subfunctions have been completed and verified. In other words, everything needed to complete a given function (the system design related to the function as well as the functions it depends on) are available when it comes time to write the function.

Another advantage is that at every step in the development process, the project is in a semi complete state and should be testable. To see what is meant by that, visualize a large structure chart and a single function being represented as a node in that tree. Bottom-up development ensures that every node below the function we are currently working on is already finished and tested. If we think of this function and all its children as a component of the larger system, then that component is essentially complete once that function is implemented and tested.



Best Practice 18.3 Bottom-up development is easier, has less wasted effort, and takes less time

As a rule, bottom-up is the recommended path for the development activity.

Testing

Testing is the process of finding and fixing defects in the codebase. Many testing activities (such as systems testing) are traditionally completed by specialists (quality assurance engineers, testers, software development engineers in test, etc.). Software developers carry a large burden in making code testable, conducting unit and integration tests, and fixing defects.

Top-Down

Top-down testing involves starting with system testing and working down to unit testing. System testing looks at the system as a whole, often from the user's perspective. When this is finished and everything works as expected, then the interplay between the various subsystems are verified. This is called integration testing. When this is finished, then each individual component or function is validated. This is called unit testing. If we were to liken this to testing a new airplane, then top-down testing would involve taking the plane for a test-flight before first making sure the engine works or even that the bolts are properly fastened in the landing gear – not a very good practice!

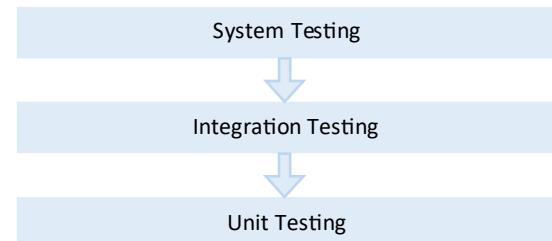


Figure 18.2:

Unit testing should be the first thing done, not the last

Top-down testing works well from the tester's perspective because it is just as easy to start at the top as it is at the bottom. However, it is very difficult from the developer's perspective. When a defect is found, where did it come from? Often a considerable amount of investigation is required before the origin of the bug can be found. This investigation often takes much longer than the bug fix itself.

Another problem with top-down testing is that it can only happen after most of the system is built. This delays testing until the end of the development process, resulting in a large lag between when bugs are introduced and when they are fixed. Generally, the larger this lag, the more difficult it is to fix the bug.

Bottom-Up

Bottom-up testing involves starting with the smallest unit of software (typically a function) and generally working up to larger components. Thus, the last thing tested is the holistic system, done only after each individual component has been verified. If we were to liken this to testing a new airplane, then the oil filter would be thoroughly verified before being fastened to the oil system. This system, in turn, would be carefully tested before connected to the engine. The engine would spend countless hours on a test stand, while the engineers run it through a myriad of scenarios. Only after the engineers have a high degree of confidence in the engine would they mount it on the airframe. This also would undergo extensive integration testing before a pilot would be allowed in the aircraft.

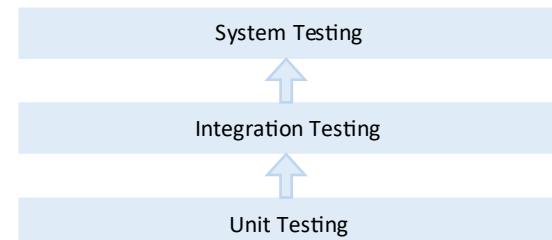


Figure 18.3:

A better way to test a system

Bottom-up testing can happen early in the development process, giving feedback to the programmer minutes after the development work on a function is finished. It helps localize bugs and ensures higher overall product quality.

Best Practice 18.4 Implement bottom-up testing in most software development processes

All indicators suggest that bottom-up testing is superior to top-down testing.

Bottom-Up Development Process

The bottom-up design process begins with a completed design and all the functions stubbed. From there, developer proceeds one function at a time until the program is completed.

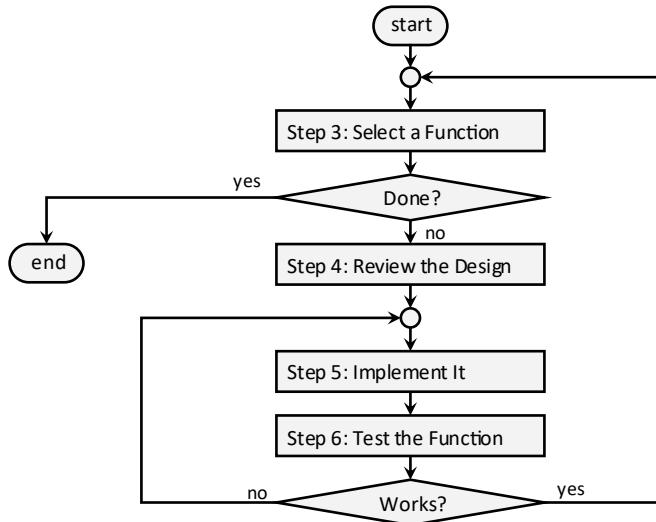


Figure 18.4:
The bottom-up
development process

The outer loop (containing Steps 3–6 inclusive) represents the bottom-up development process and the inner loop (containing Steps 5–6) is the iterative process of making sure each individual function works before moving on to the next. The overall development process depends on a system design. Note that this is often done in stages: a draft of the system is designed and stubbed. When that draft is implemented, then more of the design is completed. This continues until the entire project is complete. These constitute Step 1 and 2, the top-down component.

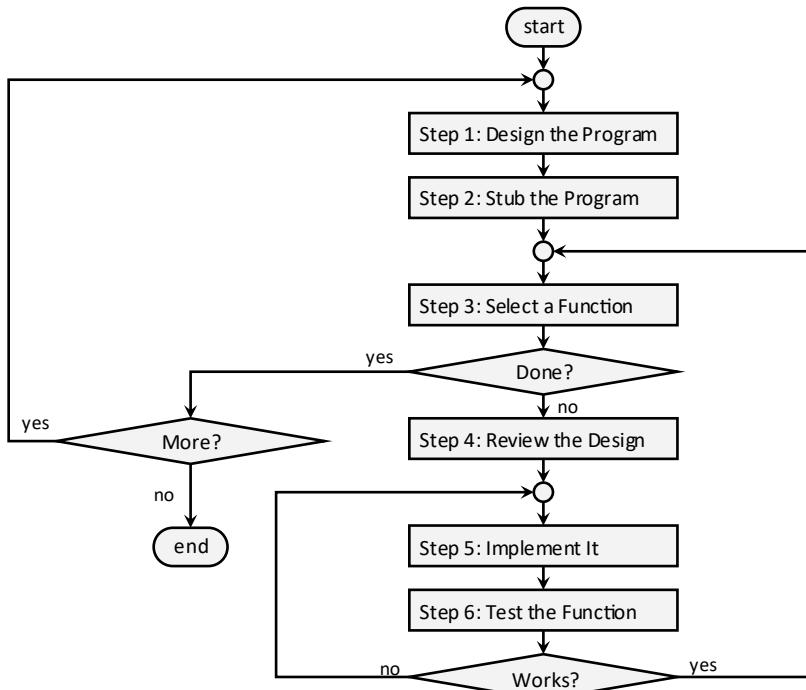
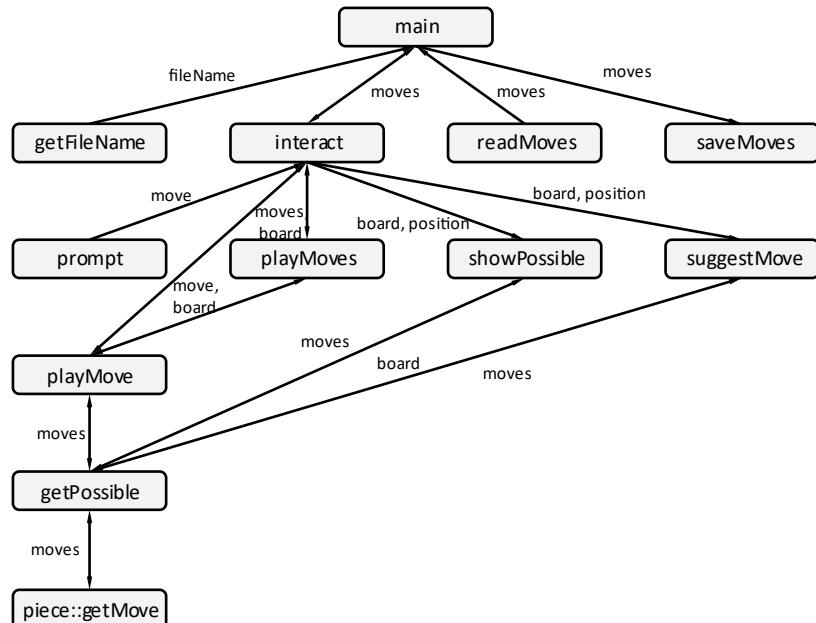


Figure 18.5:
The bottom-up process
with top-down design
coming first

Step 1: Design the Program

You cannot start coding without a design. Note that the design does not need to be complete. In fact, it often cannot be complete due to uncertainty in the requirements and the technology. It does, however, need to be complete enough to move forward. At a minimum, the design should precede development by two weeks. Ideally, the design should be roughed out in its entirety, even though it is understood that things will change.

For example, if we were to build a program to play the game of chess, we would need an initial design in the form of a structure chart. This would be completed using the top-down design process, discussed in more detail in Chapter 17:



*Figure 18.6:
A structure chart
resulting from
top-down design*

Along with this structure chart is the pseudocode to several key functions. Some functions (such as `getFileName()`) probably do not need pseudocode. The pseudocode of `saveMoves()` is the following:

```
Pseudocode

saveMoves(moves, filename)
    OPEN file from filename
    IF file ERROR
        PUT error message
        RETURN with error code

    FOREACH move IN moves
        WRITE move TO file

    CLOSE infile
```

*Figure 18.7:
Pseudocode resulting
from top-down design*

Even though this design is complete to the best of our ability, it is understood that more functions will be needed, parameters will need to be changed, and algorithms will need to be adjusted. In other words, this is a work in progress. We will have one more chance to refine this design (in Step 4) before we write any code (in Step 5).

Step 2: Stub out the Program

Before the bottom-up development process can begin (Steps 3–6), it is necessary to have each function stubbed out. This means that every function in the structure chart should have a stub. Again, Chapter 17 details how this is accomplished.

C++

```
struct Position {};
struct Move { Position from; };
class Piece {
public :
    list<Move> getMoves() { return list<Move>(); }
};

list<Move> getPossible(Piece board[8][8], Position p) {
    return board[0][0].getMoves();
}
void playMove(const Move & m, Piece board[8][8]) {
    list<Move> moves = getPossible(board, m.from);
}
Move prompt() { return Move(); }
void playMoves(list<Move> & moves, Piece board[8][8]) {
    playMove(moves.front(), board);
}
void showPossible(Piece board[8][8], Position p) {
    getPossible(board, p);
}
void suggestMove(Piece board[8][8], Position p) {
    getPossible(board, p);
}
string getFileName() { return "game.txt"; }
void interact(list<Move> & moves) {
    Piece board[8][8];
    playMoves(moves, board);
    Position p;
    → showPossible(board, p);
    suggestMove(board, p);
    Move m = prompt();
    playMove(m, board);
}
list <Move> readMoves(const string & fileName) {
    return list<Move>();
}
void saveMoves(const list<Move> & moves,
               const string & fileName) {}
int main() {
    list<Move> moves = readMoves(getFileName());
    interact(moves);
    saveMoves(moves, getFileName());
    return 0;
}
```

The class and structure definitions are just placeholders

If one function calls another, make sure to include those calls in the stubs in the simplest way possible

The parameters from the structure chart needs to be represented

main() should indirectly call every function

Figure 18.8:
The stub functions
for a chess game

Every detail in the structure chart should be represented in the stubs and vice versa. It should be possible to create a structure chart from this stubs that is exactly equivalent, right down to the parameter and function names.

Best Practice 18.5 Keep the structure chart up to date so it faithfully represents the project

Whenever the structure chart is updated, make sure to add the corresponding stubs to the project. Also, when a function is changed, update the structure chart so it is a faithful representation of the project.

Step 3: Select a Function

Whereas Steps 1 and 2 were part of the top-down design process, the bottom-up development process starts in Step 3. The purpose here is to find a suitable function from the structure chart that is to be implemented next.

As a rule, a function should be selected with the smallest number of dependencies. This could include a leaf-level function that calls no other, or it could include a function that calls others which are already implemented. The most important thing is to avoid functions that call stubs.

At this stage in the bottom-up development process, the value of keeping the structure chart up to date becomes apparent. If it is not, it is difficult to tell which functions are complete and which are not. The most convenient way to do this is to mark off functions in the structure chart once they are finished.

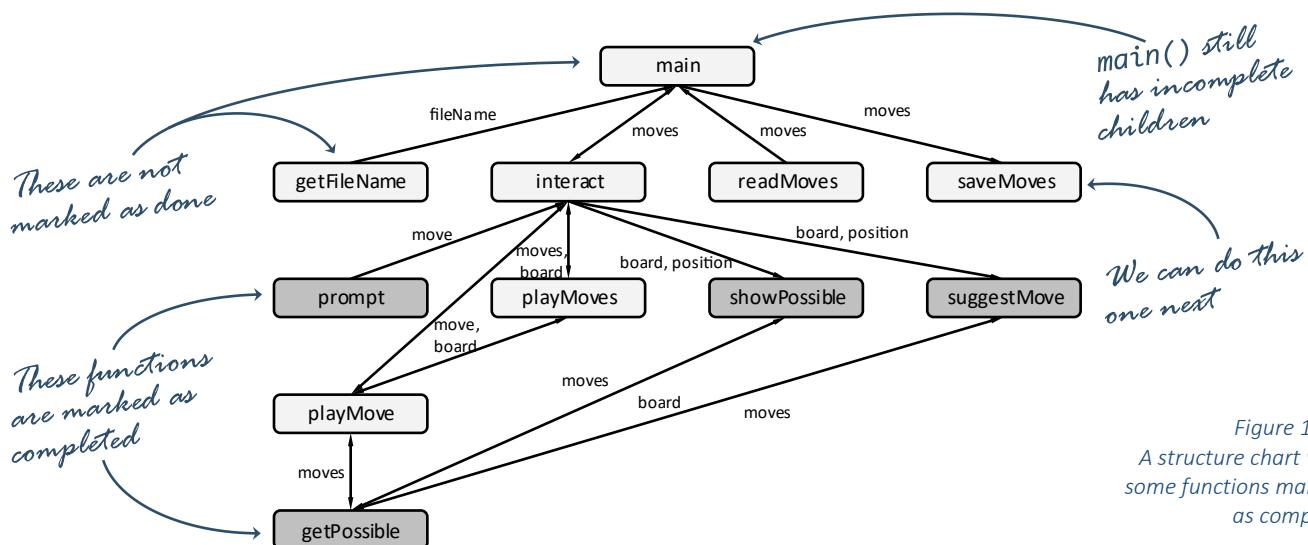


Figure 18.9:
A structure chart with
some functions marked
as complete

From this structure chart, we can see that the following functions are complete:

- `getPossible()`
- `prompt()`
- `showPossible()`
- `suggestMove()`

We can see that the following are ready for development:

- `playMove()`
- `getFileName()`
- `readMoves()`
- `saveMoves()`

There are several which are not ready because they have incomplete children:

- `playMoves()` → requires `playMove()` to be finished
- `interact()` → requires `playMove()` and `playMoves()` to be finished
- `main()` → requires `getFileName()`, `interact()`, `readMoves()`, and `saveMoves()`. This will have to be finished last.

Based on this, we will select `saveMoves()` as the next function.

Step 4: Review the Design

There is often a considerable delay between the design of a function and the function's implementation. The design could have been done by another engineer, it could make assumptions about the customer's requirements that are incorrect, or it could depend on technology that is not implemented as we expect it to be. Even if the design is flawless, it is still necessary to get the design "in your head" before you write the code. All these things mean that we need to carefully review the design before we begin development.

Recall from Step 3 that our function is `saveMoves()`. The pseudocode from Step 1 is the following:

```
Pseudocode
saveMoves(moves, filename)
OPEN file from filename
IF file ERROR
    PUT error message
    RETURN with error code

FOREACH move IN moves
    WRITE move TO file

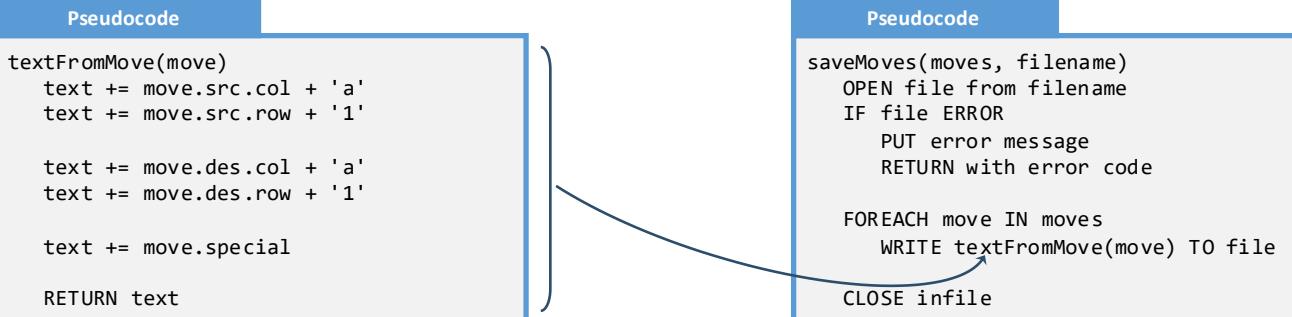
CLOSE infile
```

Figure 18.10:
Incomplete pseudocode

Most of the code follows the standard pattern for writing data to a file. This looks easy to implement in just about any programming language. There is one problem, however. How do we write a **Move** to a file? There is not currently any code to do this. Clearly another function is needed.

To make things more complicated, we don't even know what the file format is supposed to look like. With an external interface such as this, we need to perform some more requirements elicitation and talk with our stakeholder. From this, we learn that we are to use the Smith Notation of chess algebra.

Figure 18.11:
Improved pseudocode



We now have a decision to make: should we do these in two steps or one? Since this new function is distinct from the others, we decide to work on it and leave `saveMoves()` for another day. We thus add `textFromMove()` to our structure chart and continue the process.

Note that some development methodologies (such as test-driven development, discussed in Chapter 26) have us develop test cases and even implement the driver program at this step, even though the function under test is yet to be implemented.

Step 5: Implement the Function

With the design fully understood, it is time to write the code.

```
C++  
*****  
* TEXT FROM MOVE  
* Generate a textual version of a move  
* in Smith Notation  
*****  
string textFromMove(const Move & move)  
{  
    string text;  
  
    // source location  
    text += move.src.col + 'a';  
    text += move.src.row + '1';  
  
    // destination location  
    text += move.des.col + 'a';  
    text += move.des.row + '1';  
  
    // handle all the special stuff  
    if (move.enpassant)  
        text += 'E';  
    if (move.castleK)  
        text += 'c';  
    if (move.castleQ)  
        text += 'C';  
    if (move.piece != '\0')  
        text += move.piece;  
    if (move.capture != '\0')  
        text += (char)tolower(move.capture);  
  
    return text;  
}
```

Figure 18.12:
An implementation of
textFromMove()

This is a leaf-level function, depending on no other functions in the project. In fact, the only external dependencies are the `+=` and `tolower()` functions built into the string library, and the `Move` structure. This `Move` structure also needs to be defined.

```
C++  
struct Move {  
    Position src;          // where the move originated from  
    Position des;          // where the move finished  
    char piece;            // piece to be promoted to  
    char capture;          // did a capture happen this move?  
    bool enpassant;        // Enpassant pawn capture  
    bool castleK;          // kingside castle  
    bool castleQ;          // queenside castle  
};
```

Figure 18.13:
A structure needed for
textFromMove()

Best Practice 18.6 Never leave the code in a state when it does not compile

Do not let yourself be interrupted from your work when the code is in a state where it is not compiling. Mostly this means that you should make sure you have time to finish a function before starting to write code. If this absolutely cannot be avoided, comment out the incomplete code and begin the process anew the next day.

Step 6: Test the Function

The final step in the bottom-up development process is to test functions. This includes generating test cases, creating a driver function, running through all the test cases, and fixing all the bugs that are found.

Enumerate Test Cases

Since the source, destination, and special move designations are independent of one another, we can get a large amount of coverage with a small number of cases:

Name	Inputs	Outputs
Middle	src: (3,3) des:(4,4) special(None)	d4e5
Extremes1	src: (0,0) des:(7,7) special(pawn capture)	a1h8p
Extremes2	src: (7,0) des:(0,7) special(rook promote)	a8h1R
Error out of bounds	src: (-1,-1) des:(8,8) special(None)	error
Queen side castle	src: (0,4) des:(0,1) special(queen-side castle)	e1b1C
King side castle	src: (7,4) des:(7,6) special(king-side castle)	e8g8c
Enpassant	src: (5,3) des:(6,2) special(enpassant)	d6c7E

Create a Driver Function

Next, the test cases are encoded in a driver function.

```
C++  
void textFromMoveTest()  
{  
    Move moves[ ] =  
    { // src      des      pro      cap      en      king      queen  
        { {3,3},     {4,4},   '\0',   '\0',   false,  false,  false },  
        { {0,0},     {7,7},   'p',    '\0',   false,  false,  false },  
        { {7,0},     {0,7},   '\0',   'R',    false,  false,  false },  
        { {-1,-1},   {8,8},   '\0',   '\0',   false,  false,  false },  
        { {0,4},     {0,1},   '\0',   '\0',   false,  false,  true  },  
        { {7,4},     {7,6},   '\0',   '\0',   false,  true,  false },  
        { {5,3},     {6,2},   '\0',   '\0',   true,  false,  false }  
    };  
  
    for (int i = 0; i < 7; i++)  
        cout << i + 1 << '\t' << textFromMove(moves[i]) << endl;  
}
```

Figure 18.14:
A driver function

Run Through the Test Cases

From this test case, a bug is found: the error condition is not handled properly.

Output
1 d4e5
2 a1h8p
3 a8h1r
4 `0i9 ←

An error was expected here.

Figure 18.15:
Part of the output from
the driver function

When this bug is fixed and all the test cases pass, then we can return to Step 3 and work on another function. Make sure you always leave the project in a good state with all the non-stubbed functions completed and passing their tests.

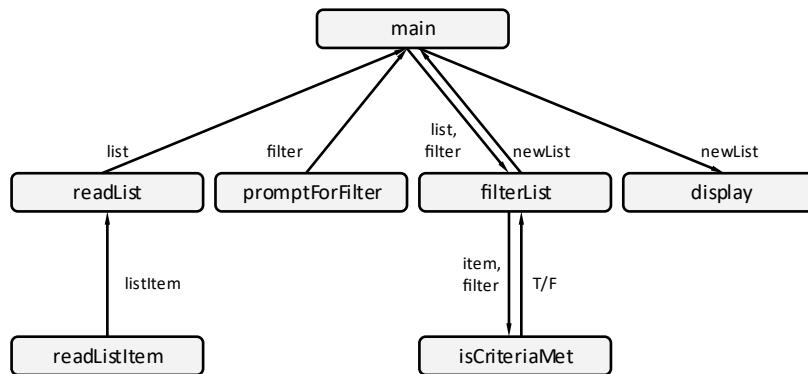
Examples

Example 18.1: Step 3 of a List Application

This example will demonstrate how to complete Step 3 of the bottom-up development process.

Problem

Consider an application that represents a shopping list. Every item in the list has a name, a location in the grocery store, and a price. The structure chart of this application is the following:



Complete Step 3 of the bottom-up development process

Solution

There are four candidates to begin the process:

Function	Reason for selection
readListItem()	When <code>readListItem()</code> is completed, then we can do <code>readList()</code> which will make testing other functions much easier.
promptForFilter()	This can certainly be done at any time.
isCriteriaMet()	Can certainly be done, but completing it does not enable <code>filterList</code> to be developed next due to its dependencies.
display()	Hard to test without a mechanism to easily create a variety of different lists.

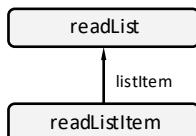
It appears that `readListItem()` will be the best place to start. We can then do `readList()` which will then enable us to do `display()`. With `readList()` and `display()`, then the filtering mechanism can be more easily tested.

Example 18.2: Step 4 of a List Application

This example will demonstrate how to complete Step 4 of the bottom-up development process.

Problem

Consider the following structure chart fragment for `readListItem()`:



We also have the following pseudocode:

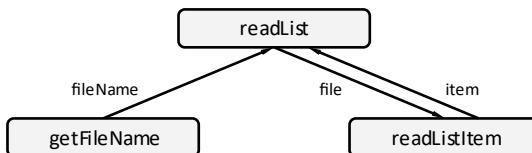
Pseudocode

```
readListItem()
    READ nameItem from file
    READ nameCategory from file
    indexCategory ← categoryDictionary[nameCategory]
    READ price
    item ← (nameItem, indexCategory, price)
    RETURN item
```

Based on this preliminary design, complete Step 4 of the bottom-up development process.

Solution

The first thing to notice is that we need to pass a file object to the function. Further investigation reveals that there is no `fileName` passed to `readList()`. Based on these observations, the following alterations are made:



Pseudocode

```
readListItem(file)
    READ nameItem from file
    READ nameCategory from file
    indexCategory ← categoryDictionary[nameCategory]
    READ price
    item ← (nameItem, indexCategory, price)
    RETURN item
```

We are now ready to begin development on `readListItem()`.

Example 18.3: Step 5 of a List Application

This example will demonstrate how to complete Step 5 of the bottom-up development process.

Problem

Implement the `readListItem()` function from Example 18.2.

Solution

We will start with the stub function (already created in Step 2):

Python

```
readListItem(file):
    return {'name': "Test",
            'category': 6,
            'price': 0.00}
```

Now the production code will be created from the pseudocode and placed in the stub function:

Python

```
readListItem(file):
    """
    This function reads one item from the file
    :param file: The file object we are reading from
    """

    # Read the raw data from the file
    name_item      = file.readline()
    name_category = file.readline()
    price          = float(file.readline())

    # Convert the category name into a code
    category_dictionary = {'produce':0,
                           'meats':1,
                           'cans':2,
                           'cereals':3,
                           'dairy':4,
                           'frozen':5,
                           'misc.':6}
    item = {'name': name_item,
            'category':category_dictionary[name_category],
            'price':price)
    return item
```

A simple run-through with the data from a trivial file reveals that it works, but more thorough testing will be needed in Step 6.

Example 18.4: Step 6 of a List Application

This example will demonstrate how to complete Step 6 of the bottom-up development process.

Problem

Test the `readListItem()` function from Example 18.3.

Solution

We will need to start with some test cases:

Name	Inputs	Outputs
Produce	Apples, produce, 0.99	Apples, 0, \$0.99
Meats	Hot dogs, meats, 0.00	Hot dogs, 1, \$0.00
Cans	Spicy salsa, cans, 100000	Spicy salsa, 2, \$100,000.00
Cereals	Life, cereals, -1.00	Life, 3, -\$1.00
Error	Banana, nothing!, 0	Error

Now we will develop a simple manual driver program.

Python

```
readListItemTest():
    with open("test.txt", "r") as file:
        item = readListItem(file)
        print("\tName:", item["name"],
              "\tCategory:", item["category"],
              "\tPrice: $", item["price"])
```

The final step is to put the test cases in a file:

test.txt

```
Apples
produce
0.99
Hot dogs
meats
0.00
...
```

When we run through these test cases, we discover than the error case is not properly handled. After fixing that case, then all the tests pass and we can return to Step 3 of the process.

Exercises

Exercise 18.1: Type of Activity

For each of the following software engineering activities, classify it as requirements elicitation, design, development, or testing.

Activity	Name
Generate test cases	
Create a structure chart representing part of the program	
Create automation representing test cases	
Ask the client what is needed from the software	
Write code	
Create a DFD describing how data move through the system	
Observe the client at work to understand his or her needs	
Create a flowchart representing a complex decision-making process in the program	
Create a manual driver program	
Create pseudocode for an algorithm	
Execute test cases	

Exercise 18.2: Steps in the Process

For each step in the bottom-up development process, give its name and briefly describe it.

Step Number	Name and Description
Step 1	
Step 2	
Step 3	
Step 4	
Step 5	
Step 6	

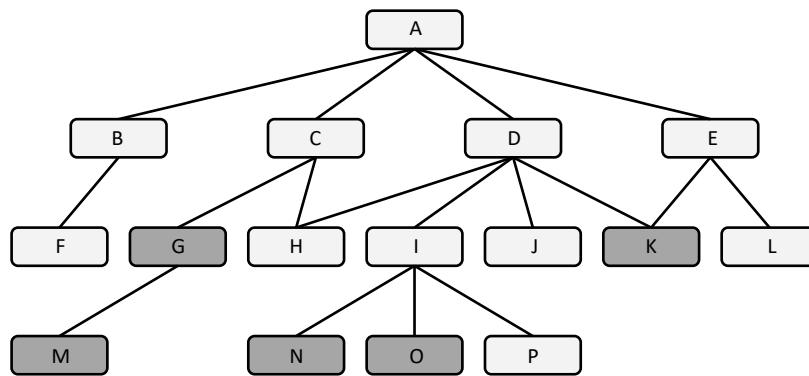
Exercise 18.3: Top-Down or Bottom-Up

For each of the following, identify whether it is better to pursue this software engineering activity as top-down or bottom-up.

Activity	Justification
Requirements Elicitation	
Design	
Development	
Testing	

Exercise 18.4: Eligible Functions

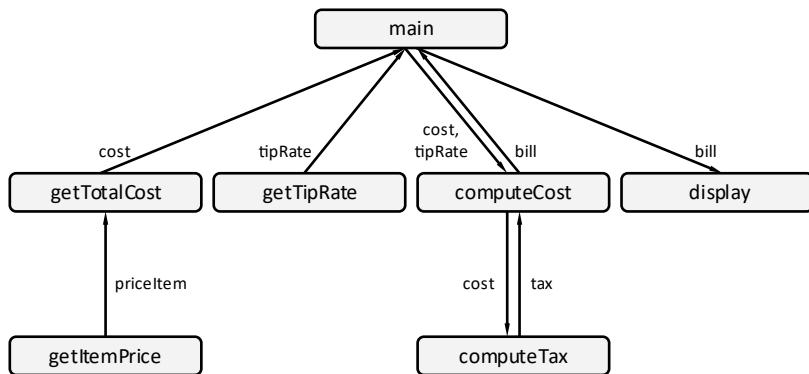
In the following structure chart, which functions are completed, which are eligible for development, and which are not ready?



Problems

Problem 18.1: Tip Calculator

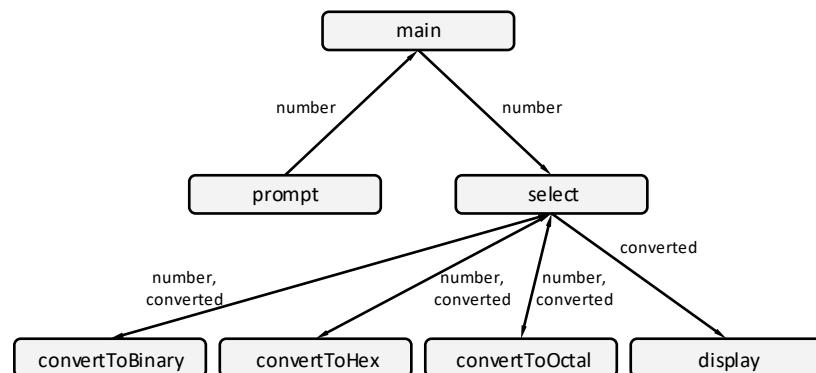
Consider a simple application to determine the prices of a meal. It will prompt the user for the individual item prices and tabulate them for the total cost. It will then prompt the user for the quality of the service: excellent = 20%, average = 15%, and poor = 10%. It will compute the tax (assume 7.8%) and then display the predicted cost of the meal.



In the programming language of your choice, implement this application using the bottom-up development process.

Problem 18.2: Convert Number

Consider a simple application to convert a decimal number (such as 42) into a binary number (b101010), hexadecimal (0x2a), or octal (o52). The structure chart is the following:



In the programming language of your choice, implement this application using the bottom-up development process.

Challenges

Challenge 18.1: Shopping List

Consider the shopping list application from Challenge 17.1:

The shopping list mobile application allows the user to create a shopping list and check off items when at the grocery store. The application can accommodate any number of shopping lists, each of which has a name, a short description, and the list itself. The user can add, duplicate, edit, checkoff, and delete shopping lists. When the program begins, the user is presented with the current collection of shopping lists. Each list has duplicate, edit, use, and delete icons. At the bottom of the list is an “add list” icon.

Implement this application in the programming language of your choice using the bottom-up development methodology.

Challenge 18.2: Credit Card Transactions

Consider the personal finance application from Challenge 17.2:

The personal finance application allows the user to store credit card transactions. When the application begins, the current account information is loaded and the user is presented with an option of entering a transaction, reconciling an account, or displaying a report. The “enter a transaction” option prompts the user for a single transaction (date, name, category, amount) and adds it to the account. The “reconcile” option allows the user to update the status of a transaction from “Open” to “Reconciled”. The “report” option will display a summary of all the transactions within a given time period, collected by category.

Implement this application in the programming language of your choice using the bottom-up development methodology.

Challenge 18.3: Maze

Consider the maze game from Challenge 17.3:

The maze game is a 2D game where the object is for a ship to navigate a maze without hitting a wall or running out of time/fuel. The game begins when a maze is loaded from a file and the ship is placed at the beginning. The user can turn the ship and apply thrusters. This game will obey the laws of motion, so in order to stop, the ship must turn around and apply counter thrusters. The game is over when the user hits a wall, runs out of fuel, or reaches the end of the maze.

Implement this application in the programming language of your choice using the bottom-up development methodology.

Functional

Functional programming is a style of programming centered on one concept: referential transparency. This means that every function's behavior is solely dependent on its input, completely independent of system state.

Complexity is the thorn in every programmer's side, the cause of the downfall of many a project. While a certain degree of complexity is intrinsic to the problem domain an application is meant to represent, much of it is self-imposed. We can reduce system complexity by avoiding programming constructs known to increase it. There is a programming paradigm that helps us reduce complexity and makes a whole class of problems easier to solve. This paradigm is functional programming.

Functional programming is a style of programming where every function's behavior is solely dependent on its input, completely independent of system state. Imagine a single function called twice with identical inputs resulting in different outputs. The outputs are different because they depend on input into the system as well as on a global variable, data from a file, or some other system state. When a function changes system state or relies upon volatile system state, it is said to be sensitive to side effects. Functional programming is a programming style where side effects are impossible because there is no "system state"; functions only have access to their input.

A side effect is an alteration of system state that may cause a change in the behavior of one or more components

Functional programming has several benefits, including facilitation of mathematical modeling, simplification of concurrency challenges in multi-process scenarios, and, of course, reduction of overall system complexity. Functional programming should be in every developer's tool kit.

Three Programming Paradigms

A programming paradigm is a way of looking at or approaching a problem. Oftentimes, paradigms are accompanied by a philosophy or a set of values. There are several programming paradigms utilized by developers the world over, but the most common three are imperative, object-oriented, and functional.

Paradigm	Description
Imperative	Programs are created by a collection of ordered statements that control flow and compute values
Object-Oriented	Programs are organized into classes (each of which maintains state) and the methods that work on them
Functional	Programs are organized into functions, each of which produces output from input and cannot change system state

Imperative

Imperative programming, also known as structured programming, or procedural programming, is a style of programming where applications are assembled from variables, IF-statements, mathematical equations, loops, and functions. Each of these operations map directly to machine language instructions found on virtually every computer. Most programmers learn to write code using the imperative paradigm. Imperative programming is the subject of Unit 0 and Unit 1 of this book. Imperative code also represents the vast majority of the code that executes on the world's computers.

In the imperative world, nouns and verbs stand on equal footing. Nouns are represented by variables, and their values can change according to the needs of the program.

Best Practice 19.1 Variable names should be nouns because they describe things

Verbs are represented as functions; they perform tasks. Imperative programs are a combination of nouns and verbs.

Best Practice 19.2 Function names should be verbs because they describe actions

The first programming languages (aside from Lisp) were imperative. Early languages were developed without an understanding of what makes large software systems difficult to produce and maintain. As a result, they allow for many constructs which have been found to be problematic: global variables, goto statements, etc. Purely imperative programs often do not scale well. This is because the largest unit of programming is a function. A typical desktop application can have tens of thousands of functions!

Object-Oriented

Object-oriented programming is a style of programming where applications are assembled from classes. A class is a programming construct consisting of attributes and operations. Attributes are variables associated with the class (also known as member variables) and operations are functions associated with the class (also known as member functions and methods). Object-oriented programming is an extension of imperative programming: the code within an operation is imperative by nature.

In the object-oriented world, nouns reign supreme. Programs are developed by first identifying the nouns and then determining the verbs that act on the nouns. Pure object-oriented languages such as Java do not allow for standalone functions! A function can only exist if it is escorted by a class. Note that object-oriented functions (called methods) contain all the code one would expect to find in an imperative algorithm: IF statements, FOR loops, variables, and data structures such as arrays.

Best Practice 19.3 Classes should be nouns because they describe things

Object-oriented programming became popular in the 1980s. It presented programming constructs larger than functions, reducing the complexity of large applications. Units 2 and 3 of this text will discuss the two aspects of object-oriented programming (encapsulation and inheritance) in more detail.

Functional

Functional programming is a style of programming where applications are assembled from functions. Verbs are at the center and nouns are relegated to being silent observers. The functional programming philosophy was inspired by mathematical functions, specifically by a branch of mathematics called lambda calculus. Here, each function is independent from the rest of the program. This is called referential transparency: where the output of a function is completely determined by the input with no room for side effects. Referential transparency is achieved in function programming with two constraints: pure functions and immutability.

Referential transparency is achieved through pure functions and immutable data

Pure Functions

A function is pure if its operation is completely independent of any program state:

Not sensitive to system state: The only thing influencing the output of a function is the input parameters. If the function is called twice with identical parameters, it will produce exactly the same output.

Does not change system state: The only thing that results from the function is the output it produces. It does not alter system state in any capacity. Aside from the return value, a function call has no residue.

The adjective “pure” relates to the degree of independence the function has from the rest of the program – in other words, the coupling. A pure function cannot have interactive coupling because the interaction would require it to maintain and alter some sort of state. It often cannot have superfluous coupling because the extra scope often touches upon system state.

Best Practice 19:4 Make a function pure whenever possible; it simplifies coupling

As a general rule, a programmer should strive to make functions pure. This reduces complexity and simplifies integration with the rest of the program. What, then, disqualifies a program from being pure?

- **Global Data:** Any reference to global data, whether to read the data or to write to the data, disqualifies a function from being pure. Some will argue that this also applies to global constants. For example, if a program defines π as 3.14159, then any function referencing π would not be pure.
- **Files:** Any function reading from or writing to a file cannot be pure. Clearly the file will be sensitive to system state (where the “system” is larger than the program boundary). However, most useful applications need to perform file I/O, so the loss of the “pure” designation does not indicate a reduction in quality.
- **Random Number Generators:** Randomness is fundamentally contrary to predictability, a core component of the functional programming philosophy. Functions requiring randomness are difficult to make pure.
- **User Interactions:** For much the same reasons that files and random number generators preclude a function being called “pure,” user input and output disqualify a function from being pure.
- **By-Reference:** Any function accepting by-reference parameters (excluding constant by-reference parameters) are not pure because no guarantee can be made that the parameter will not be altered by the function.

Immutability

Immutability is the variable property where, once a value is assigned, it cannot be changed. In other words, constant variables are immutable. In the functional programming world, all variables must be immutable.

Immutability eliminates side effects because all variables are read-only

it becomes possible to mathematically prove a whole host of desirable properties of code, such as being thread safe (works well in a multi-threaded programming environment).

The immutability constraint, however, is the Achilles heel of functional programming. It prohibits such common tasks as counting (how do you count when your counter cannot change?), updating values (what if I want to update my account balance?), and transforming data (what if I need to filter a list?). Functional programming has answers to all of these scenarios, but they tend to be less straightforward than their imperative counterparts.

Immutability prohibits counting, updating values, and transforming data

Counting Through Recursion

While counting is impossible due to the immutability constraint, recursion is not. This means that if you want to count from 1 to 100, you need to write a recursive function to perform the 100 iterations for you. Not only is the recursion less intuitive than the imperative **FOR** loop, it is less efficient. A hundred instances of the function will find themselves in the call stack, providing vastly more overhead and much less performance.

Most functional programming languages minimize the cost of recursion by performing a host of optimizations (see the section about Tail Recursion in Chapter 16). However, even the most efficient solutions are not as fast as a simple **FOR** loop.

Updating Through Copying

While it is impossible to update a value due the immutability constraint, copying values are not. The default way to handle update-like functionality is to simply create a new copy.

Most functional programming implementations are able to detect when an old copy is no longer used, and they will aggressively destroy it. This certainly reduces the cost of having many abandoned and copies consuming memory. However, it does not address the cost of making the copy in the first place.

Many functional languages perform a significant amount of work to reduce or eliminate redundant copies of values. Python, for example, will only copy a variable if it is different than the original. Otherwise two variables will refer to the same physical location in memory.

Transformations through Special Tools

The first imperative languages did not have functions as we know them today. Instead, it was necessary to jump to a different block of code then jump back. Eventually this common operation was codified as procedures and then as functions. Functional programming has similarly codified common transformations with the filter, map, and reduce functions. Skilled use of these tools greatly reduce the need for mutable variables.

Functional Programming Tools

Functional programming is built from several building blocks: high-order functions, lambda, recursion, filter, map, and reduce. Each of these can be used in non-functional programming environments and thus have far-reaching applicability.

High-Order Functions

A first-order function is a function that only takes data as a parameter and returns only data. Probably every function you have written up to this point was a first-order function. High-order functions, also known as composing functions, are functions that operate on other functions. This is usually done by taking another function as a parameter or returning a function as a result.

Traditionally, we think of the terms “function” and “algorithm” as interchangeable. Both describe a set of actions designed to perform an action. However, a function is something more. A function, when compiled, resides in memory. This means we can create a variable which refers to that location in memory. Thus, variables can hold data and variables can also refer to functions. The big advantage to this is that we can pass a function as a parameter to another function. Why would we want to do such a thing?

Imagine a function which sorts an array of strings. Sometimes we wish to sort the array alphabetically (from A through Z), sometimes reverse alphabetically (from Z to A), sometimes by the length of the string, and sometimes in a case-insensitive way. Rather than write four separate sort functions, we can write a single sort function and pass a comparison function as a parameter (called `compare()`). This function would be used by the sorter to tell which of two strings goes in the front.

```
Pseudocode
isGreater(text1, text2)
    RETURN lowercase(text1) > lowercase(text2)

sort(array, compare)
    FOR iPivot ← array.size - 1 ... 1
        FOR iCompare ← 1 ... iPivot
            IF compare(array[iCompare - 1], array[iCompare])
                swap(array, iCompare - 1, iCompare)
```

Figure 19.1:
Sort is a high-order
function because it takes
compare as a parameter

Here, `sort()` is a high-order function because it takes `compare` as a parameter. Many programming languages support storing function references as variables. The following are examples:

```
C
typedef bool (*Compare)(const char *, const char *);
void sort(char * array[], int size, Compare compare);

Python
def sort(array, compare): ...

C#
public delegate bool Compare(string t1, string t2);
void sort(string array[], Compare compare);
```

Figure 19.2:
Different ways to pass
functions as parameters

Lambda

A variable is a named location that stores data. Note that we can have anonymous variables which are just places that stores data but have no name. An example would be a mathematical expression:

Figure 19.3:
An anonymous variable

Ruby

```
puts "A circle has an area of #{3.14159 * radius * 2.0}"
```

This value was never given a name

Figure 19.4:
An anonymous function

A lambda is an anonymous function: that is, a collection of statements designed to perform a task. The only difference between a lambda and a traditional function is that lambdas are not given a name. Why would we do such a thing? As with our anonymous expression example above, we can pass a lambda as parameter without needing a formal function declaration.

Swift

```
array.sorted(by: { t1, t2 in return t1 > t2 })
```

This function was never given a name

Many programming languages support lambda functions. The following are examples of greater-than between integers in a variety of languages. In these cases, we will assign the lambda to the variable `l` for convenience.

C++

```
auto l = [](int p1, int p2) { return param1 < param2; };
```

Java

```
Func l = [](int p1, int p2) -> p1 < p2;
```

PHP

```
$l = function($p1, $p2) { return $p1 < $p2; };
```

C#

```
Func<int, int> l = (int p1, int p2) => p1 < p2;
```

Python

```
l = lambda p1, p2 : p1 < p2
```

JavaScript

```
var l = (p1, p2) => p1 < p2;
```

VB

```
Dim l = Function(p1, p2) p1 < p2
```

Figure 19.5:
Lambda functions in a variety of languages

Filter

Filter is a function that produces a new collection from an old one based on a passed criteria. Filter expects two parameters: the collection to be operated on and the function which is to be applied. The function must take a single instance of the collection as a parameter and return a Boolean value. Functions returning Boolean values are called predicates. In the context of filters, a predicate can be thought of as a gatekeeper; it determines which elements from the collection are copied into the output collection and which are excluded.

Filter produces a new collection from an old one by selecting only those elements that satisfy a given condition

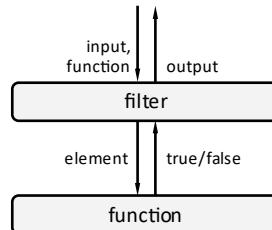


Figure 19.6:
filter() takes a function as
a parameter and calls it

The purpose of filter is to reduce the size of a collection according to some criterion. For example, if we had a collection of numbers and wished to extract only the odd ones, then filter would be a good choice for this task.

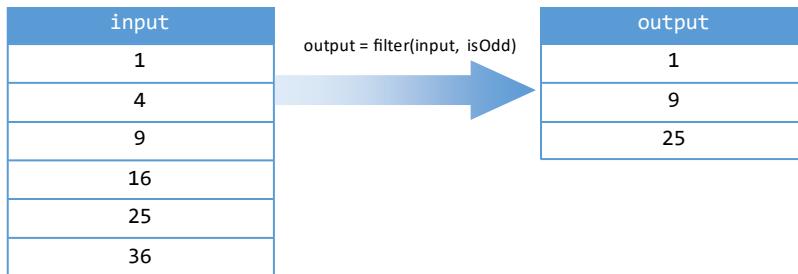


Figure 19.7:
filter() returns a subset of
the input array

The easiest way to call the `filter()` function which such a simple predicate is to use a lambda:

PHP

```
$input = array(1, 4, 6, 16, 25, 36);
$output = array_filter($input,
    function ($element) { return ($element & 1); });
```

Figure 19.8:
An implementation of
Figure 19.7

The `filter()` function is implemented by iterating through all the elements in the input array and, if the predicate function returns `true`, then adding that element to the output array.

Pseudocode

```
filter(input, function)
  FOREACH element IN input
    IF function(element)
      output += element
  RETURN output
```

Figure 19.9:
The code behind the filter
function

Map

Map is a function that produces a new collection from an old one based on a passed criterion. The result is the same size as the original. Map expects two parameters:

the collection to be operated on and the function which is to be applied. The function must take a single instance of the collection as a parameter and return another element, potentially of a different data type.

Map applies a function to all the members of a collection to produce a new collection

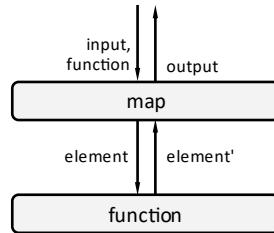


Figure 19.10:
map() takes a function as a parameter and calls it

The purpose of map is to transform data. For example, if we had a collection of numbers and wished know which were odd and which were even, then a map would be a good choice for this task.

input	output
1	true
4	false
9	true
16	false
25	true
36	false

Figure 19.11:
map() transforms an input array into an output array

Notice that `isOdd()` returns a Boolean, so the output array is an array of Booleans whereas the input array was one of integers. The easiest way to call the `map()` function which such a simple predicate is to use a lambda:

```
Python
input  = [1, 4, 6, 16, 25, 36]
output = list(map(lambda x: x % 2 == 0, input))
```

Figure 19.12:
The code producing the output from Figure 19.11

The `map()` function is implemented by iterating through all the elements in the input array and applying the passed predicate function to each. The result is then added to the output array.

```
Pseudocode
map(input, function)
    FOREACH element IN input
        output += function(element)
    RETURN output
```

Figure 19.13:
The code behind map()

Reduce

Reduce is a function that produces a singular value that is computed by combining all the values in the input collection. Reduce expects two parameters: the collection to be operated on and the function which is to be applied. This function must take two parameters. The first is of the same data type as each element of `reduce()`'s input array. The second is of the same data type as `reduce()`'s output.

Reduce computes a singular value representing the combination of all the elements in a collection

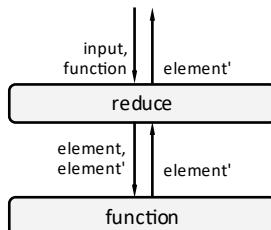


Figure 19.14:
Reduce takes a function as
a parameter and calls it

The purpose of reduce is to combine data. For example, say we wished to know the sum of all the letters in an array of names. The input array would be an array of strings. The output value would be an integer. The input function would need to take a string as the first parameter (corresponding to the data type of one element of `reduce()`'s input array) and an integer as the second parameter (corresponding to the output of `reduce()`).

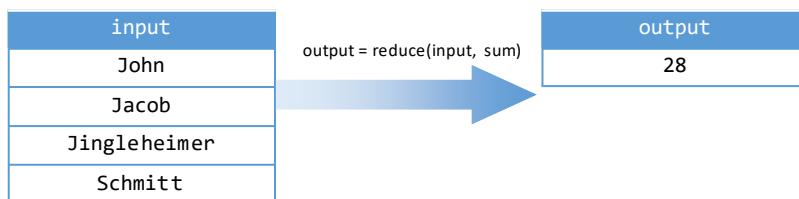


Figure 19.15:
Reduce produces a
singular output from an
input array

Notice that `sum()` returns an integer, so the output value is a single integer. This takes a string (one element of the input array) and an integer (the sum up to this point in the iteration) as parameters and returns the new sum.

Swift

```
var input = ["John", "Jacob", "Jingleheimer", "Schmitt"]
var output = input.reduce(0, {$0 + $1.characters.count})
```

Figure 19.16:
The code producing the
output from Figure 19.15

The `reduce()` function is implemented by iterating through all the elements in the input array and applying the passed predicate function to each. The result is then returned.

Pseudocode

```
reduce(input, function)
  FOREACH element IN input
    output ← function(element, output)
  RETURN output
```

Figure 19.17:
The code behind `reduce()`

Practical Applications

There are several programming languages that are purely functional. Few practical applications have been built from such languages due to the limited number of available libraries and their performance challenges. However, most modern languages have functional capabilities. In many scenarios, we can reduce complexity by applying some of these principles.

Minimize Side Effects

As a rule, minimizing side effects is a good thing in algorithm or system design. The more functions in your system that are “pure,” the less impact side effects will have on your overall design.

Best Practice 19.5 Make functions pure whenever possible

All functions in your design should be pure with few exceptions. Each exception should have a very good reason for breaking such a rule. Common examples include file I/O, user I/O, and random number generators. If a function is not pure and does not fall into one of these categories, then careful consideration should be taken as to why that is the case.

Apply Recursion Appropriately

Recursion can reduce conceptual complexity. It can also have minimal performance impact when used appropriately.

Best Practice 19.6 Favor iteration over recursion

As a rule, iterative solutions outperform recursive solutions. There are some exceptions to this rule. One example is iterating over a binary tree or similarly complex data structures. There, the recursive depth is small and the iterative equivalent is complex. For most other solutions, iteration is superior.

Best Practice 19.7 Master recursion

Functional programming languages are convenient and powerful tools to help you practice recursion. Many a programmer avoids an elegant recursive solution for a cumbersome iterative one due to their lack of familiarity with recursion. Let this not be you! Recursion is a potent tool that should be in every programmer’s tool kit.

Pass Functions as Parameters

Before the invention of templates and object-oriented programming, passing functions as parameters was a common programming technique. Though the need for it has decreased in subsequent years, it remains a powerful technique that should be in every programmer’s tool kit. As is the case with recursion, functional languages are a great place to practice this technique.

Use Filter, Map, and Reduce

Filter, map, and reduce exist in almost all modern languages and are applicable to a surprising number of scenarios. Every programmer should have them in their tool kit.

Examples

Example 19.1: Filter

This example will demonstrate how to use filter to retrieve a subset of an array.

Problem

Consider the following collection of words and their associated part of speech:

Word	Part of Speech
chicken	Noun
cattywampus	Adjective
platypus	Noun
lollygag	Verb
wabbit	Noun
behoove	Verb
plethora	Noun
grumpily	Adverb
clownish	Adjective
grunt	Verb
ruthlessly	Adverb
sticky	Adjective

Write a function to extract all the nouns from this list.

Solution

Because we are taking a large list and turning it into a small one, this looks like a job for reduce. Notice, however, that this is not just a list: it is a list of tuples, the first item being the word and the second being the part of speech.

```
JavaScript
const words = [{ word:"chicken", type:"noun"}, { word:"cattywampus", type:"adjective"}, { word:"platypus", type:"noun"}, { word:"lollygag", type:"verb"}, { word:"wabbit", type:"noun"}, { word:"behoove", type:"verb"}, { word:"plethora", type:"noun"}, { word:"grumpily", type:"adverb"}, { word:"clownish", type:"adjective"}, { word:"grunt", type:"verb"}, { word:"ruthlessly", type:"adverb"}, { word:"sticky", type:"adjective"}];
```

From here, we can write a simple lambda function which looks at the `type` field of each tuple:

```
JavaScript
const nouns = words.filter(record => record.type == "noun");
```

Example 19.2: Map

This example will demonstrate how to use map to convert the elements in an array from one format to another.

Problem

Write a program to convert an array of floating-point numbers into strings in a currency format. The imperative algorithm is the following:

Pseudocode

```
convertAll(prices)
    FORALL price IN prices
        result ← $ + price (To two decimal places)
        results += result
    RETURN results
```

Solution

This is a great application for map: we need to apply the same function to every element in the collection. Note that C++ uses `vector` for arrays and the `transform()` function for map.

C++

```
// application function
string convertItem(float value)
{
    // format for currency
    stringstream ss;
    ss.setf(ios::fixed | ios::showpoint);
    ss.precision(2);

    // create a string and return it
    ss << "$" << value;
    return ss.str();
}

// take a vector of floats and return a vector of strings
vector<string> convertAll(const vector<float> & input)
{
    vector<string> output(input.size());
    transform(input.begin(), input.end(), // input buffer
              output.begin(),           // output buffer
              convertItem);            // conversion func
    return output;
}
```

Example 19.3: Reduce

This example will demonstrate how to use reduce to turn an imperative algorithm into a functional one.

Problem

Write a program to find the largest item in a list. The imperative algorithm is the following:

Pseudocode

```
findLargest(numbers)
    largest ← numbers[0]
    FOR i ← 1 ... numbers.size
        IF numbers[i] > largest
            largest ← numbers[i]
    RETURN largest
```

Solution

Our first attempt at a solution is to handle the **FOR** loop with recursion. We will identify the stop condition (when the size of the array is one) and the progress step (shrinking the size of the array by one using the `[1:]` notation in Python).

Python

```
def find_largest(numbers):
    if len(numbers) == 1:           # End condition
        return numbers[0]
    else:                          # Progress step
        rest = find_largest(numbers[1:])
        return rest if rest > numbers[0] else numbers[0]
```

We may notice that this is just a reduce application. In this case, we are “passing down” the largest item in the array.

Python

```
def largest(x, y) -> int:
    return x if x > y else y

def find_largest(numbers) -> int:
    return reduce(largest, numbers)
```

This can be further reduced (pardon the pun) by using a lambda rather than a traditional named function:

Python

```
def find_largest(numbers) -> int:
    return reduce(lambda x, y : x if x > y else y, numbers)
```

Exercises

Exercise 19.1: Terms and Definitions

From memory, recite the definition corresponding to each term.

Term	Definition
Map	
Pure Function	
Reduce	
Immutability	
Filter	
Lambda	
High-Order Function	

Exercise 19.2: Programming Paradigms

Classify each of the following as being part of the imperative, object-oriented, or functional programming paradigm. Note that some answers may be a combination.

Programming Tool	Paradigm
A repetitive control structure such as a FOR loop	
A constant variable	
A class	
An updatable variable	
A standalone function, not part of a class	
Recursion	
Array	

Exercise 19.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Lambda functions may be without a name	
You need a dedicated function language to use functional programming techniques	
A filter always reduces an array to a single value	
Functional languages allow for counting using FOR loops	
A map can return an array of a different size than the input array	
A pure function can alter system state	
A filter is a high-order function	

Exercise 19.4: Functional Programming Tools

For each of the following scenarios, identify a functional programming tool which would be useful. These tools include lambda, recursion, filter, map, and reduce.

Scenario	Tool
I would like to count from 1 to 10	
I would like to transform the data in an array from one format to another	
I would like to pass a predicate to a function, and the predicate is trivial	
I would like to reduce a large array to a small array based on a given condition	
I would like to continue iterating until the end of the file is reached	
I would like to compute the average value in an array of grades	
I would like to convert an array of number grades to an array of letter grades	

Problems

Problem 19.1: Filter

I would like to take up cycling and was given a shopping list of “essential items,” each of which has a name and a category:

Name	Category	Price
Bib Shorts	Clothing	\$92.50
Roubaix	Bicycle	\$3,599.99
Cycling Computer	Accessories	\$394.99
Helmet	Accessories	\$299.99
Road Shoes	Shoes	\$144.99
700c Presta Tube	Accessories	\$5.25
Jersey	Clothing	\$25.99
Multi-function Tool	Accessories	\$22.99
Gloves	Accessories	\$8.99
Cleats	Shoes	\$15.99
Power Pedals	Accessories	\$999.99
Socks	Clothing	\$8.50

I would like to retrieve a list of elements that are Accessories. In the programming language of your choice, write the code to produce this list using filter.

Problem 19.2: Map

Consider the list of items in the shopping list from Problem 19.1. On Saturday, the store will be having a sale where everything is 20% off. In the programming language of your choice, write the code to alter these prices using map.

Problem 19.3: Reduce

Consider the list of items in the shopping list from Problem 19.1. I would like to know how much this new hobby will cost me. In the programming language of your choice, write the code to compute the cost using reduce.

Problem 19.4: Remove Spaces

The following code is written imperatively:

Pseudocode

```
removeSpaces(input)
    FOREACH letter IN input
        IF not isSpace(letter)
            output += letter
    RETURN text
```

Rewrite this code using the functional programming paradigm in the language of your choice.

Problem 19.5: Fahrenheit to Celsius

The following code is written declaratively:

Pseudocode

```
convert_celsius(fahrenheit)
    FOR i ← 0 ... fahrenheit.length - 1
        celsius[i] ← (fahrenheit[i] - 32) × 5/9
    RETURN celsius
```

Rewrite this code using the functional programming paradigm in the language of your choice.

Problem 19.6: Is Prime

The following code is written declaratively:

Pseudocode

```
isPrime(number)
    prime ← true
    FOR i ← 2 ... √ number
        prime ← prime AND (number % i ≠ 0)
    RETURN prime
```

Rewrite this code using the functional programming paradigm in the language of your choice.

Challenges

Challenge 19.1: Filter

Consider the pseudocode for the `filter()` function defined imperatively.

Pseudocode

```
filter(input, function)
  FOREACH element IN input
    IF function(element)
      output += element
  RETURN output
```

In the programming language of your choice, create a functional version of `filter()` without using the built-in filter, map, or reduce functions. Hint: you will need to define this recursively.

Challenge 19.2: Map

Consider the pseudocode for the `map()` function defined imperatively.

Pseudocode

```
map(input, function)
  FOREACH element IN input
    output += function(element)
  RETURN output
```

In the programming language of your choice, create a functional version of `map()` without using the built-in filter, map, or reduce functions.

Challenge 19.3: Reduce

Consider the pseudocode for the `reduce()` function defined imperatively.

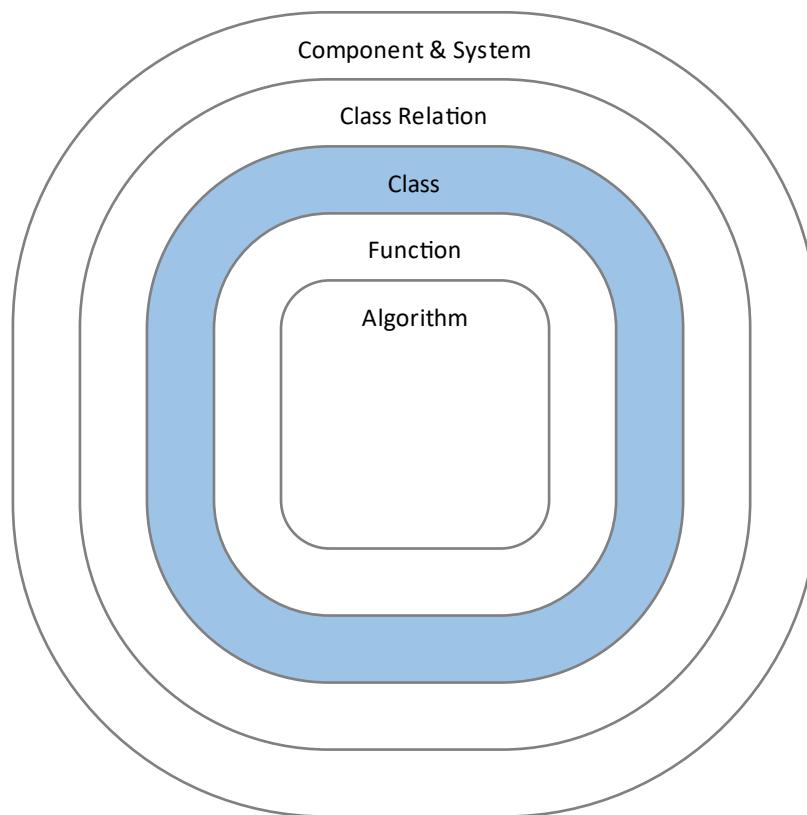
Pseudocode

```
reduce(input, function)
  FOREACH element IN input
    output ← function(element, output)
  RETURN output
```

In the programming language of your choice, create a functional version of `reduce()` without using the built-in filter, map, or reduce functions.

Encapsulation Design

Encapsulation is the process of defining a class to represent a concept within a program. Since classes are comprised of data (called attributes) and functions (called operators), classes are built from functions. Thus, encapsulation builds from modularization which builds off algorithms.



Class Diagram I

Chapter 20

UML Class Diagrams are a design and analysis tool enabling programmers to visualize object-oriented (OO) design features in a program.

Virtually every new programming language developed in the past thirty years includes classes in some capacity. While it is easy to define a class in most programming languages, designing classes effectively can be very difficult. Fortunately, we have a design language for classes enabling us to design and evaluate classes without concerning ourselves with implementation details or the syntax of the target programming language. Of all the design tools available to programmers today, UML Class Diagrams are the most commonly used. They are the cornerstone of object-oriented analysis and design.

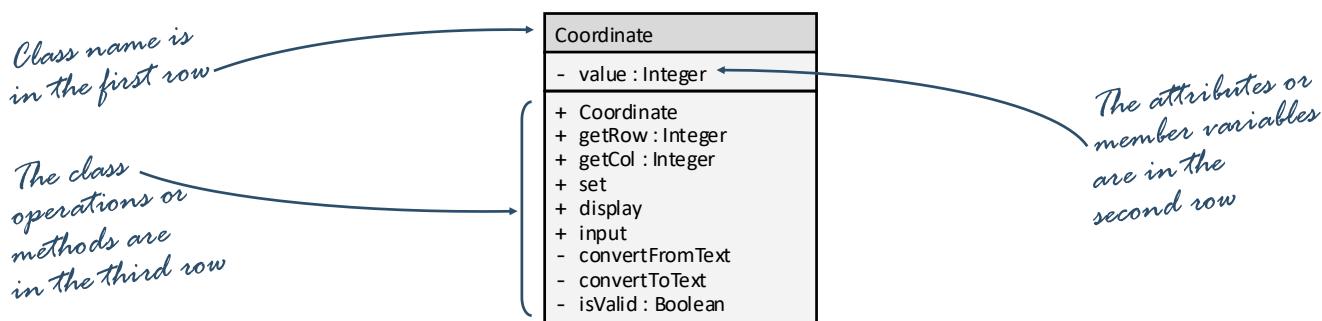


Figure 20.1:
Class diagram

A class diagram is a one-column, three-row table where the first row represents the name of the class, the second contains a list of all the attributes, and the third contains a list of all the operations in the class. Note that it is not uncommon to omit the second or third row in some situations. The class diagram above illustrates the structure of the **Coordinate** class. There is one attribute or member variable (**value**), six public operations or methods (**Coordinate()**, **getRow()**, **getCol()**, **set()**, **display()**, and **input()**), and three private operations (**convertFromText()**, **convertToText()**, and **isValid()**). Class diagrams have the following properties:

Property	Description
Use	Design and illustrate class definitions
Viewpoint	Logical: how nouns are represented in the program
Strength	The standard tool for OO design
Weakness	Additional tools are usually needed to complete a design

Viewpoints

Recall from Chapter 10 the four different design viewpoints:

Viewpoint	Description
Process	The verbs of the system. How algorithms work.
Development	How a program is partitioned and organized.
Logical	The nouns of the system. How data are stored.
Physical	The hardware and physical components.

We have seen two different Process tools (flowcharts and pseudocode) enabling us to describe the verbs of the system. We have seen two different Development tools (structure charts and DFDs) helping us visualize the structure of the system. The class diagram is our first Logical tool. It describes the nouns of the system. Not only are class diagrams the most commonly used Logical tools, but they are arguably the most widespread view used by software engineers around the world. There is one additional Logical tool which you will almost certainly encounter in your career: the entity relationship diagram. This powerful tool is used to describe databases and is thus beyond the scope of this text.

With the inclusion of class diagrams, our viewpoint decision tree is more complex.

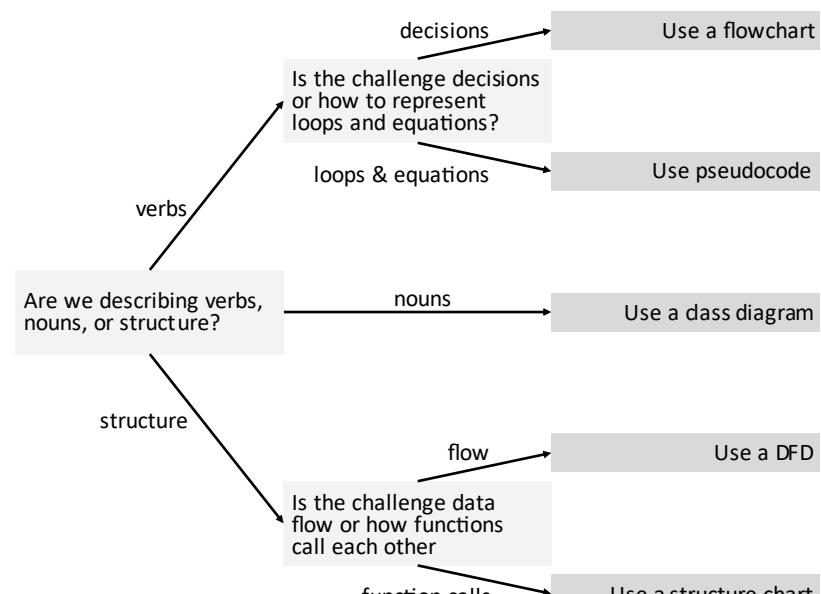


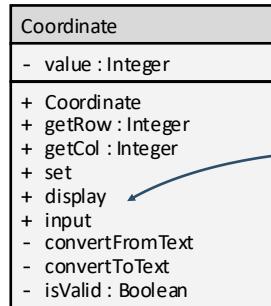
Figure 20.2:
Decision tree representing
the selection of a design
tool

Note that the choice between flowcharts and pseudocode can be subtle. In many situations, it may boil down to personal preference. The choice between structure charts and DFDs may also be difficult to discern. However, selecting a class diagram is straightforward. If you are designing a class (not the functions within the class), then it is the clear choice.

Multiple Tools to Describe One Program

For most nontrivial design problems, one tool is insufficient to describe a given program. For example, consider a designer building a class to represent a position on a chessboard. Since we are building a class, then a Logical viewpoint such as a class diagram is an obvious choice.

Figure 20.3:
Class Diagram for
Coordinate



Class diagrams list the methods in the class, but do not describe how they work or how they call each other.

Note that the class diagram describes the member variables and methods of the class but does not describe how any of the methods work. For that, pseudocode is needed.

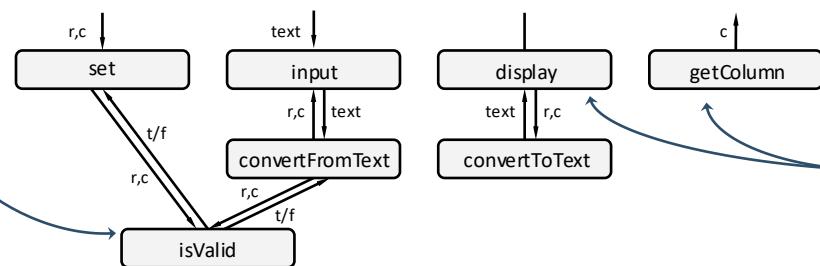
Pseudocode does not provide any clues about the structure of the class

Pseudocode
<pre> Coordinate ::= convertFromText(text) IF a ≤ text[0] ≤ h row ← text[0] - a ELSE ERROR IF 1 ≤ text[1] ≤ 8 row ← text[1] - 1 ELSE ERROR END </pre>

Figure 20.4:
Pseudocode for
convertFromText()

Finally, neither the pseudocode nor the class diagram describes how the different methods call each other. For this, we will need a structure chart.

The structure chart does not describe the layout of the classes or what happens in the methods



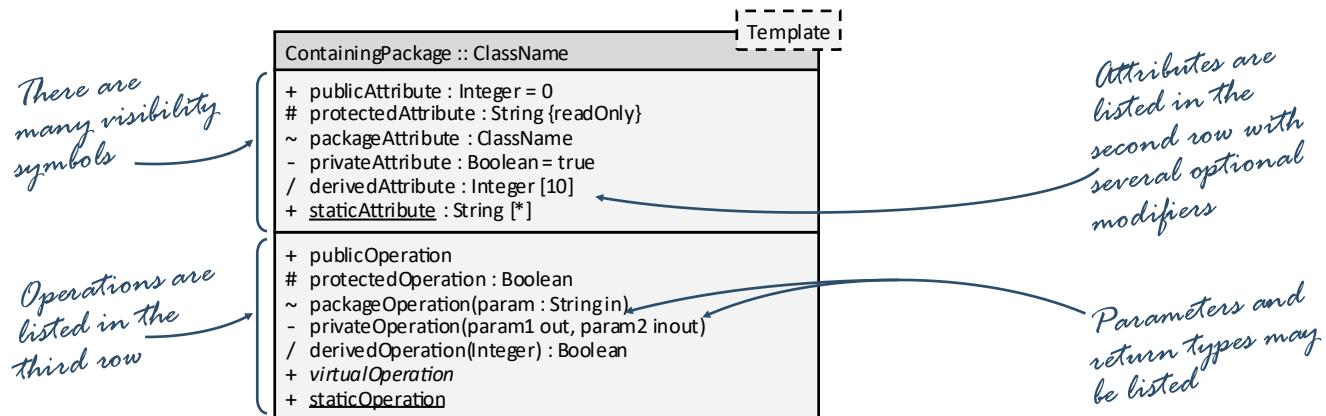
Structure charts for classes often have more than one starting point

Notice that each tools shines light on a different aspect of the problem. If we use only one tool, then large parts of the design remain in shadows.

Figure 20.5:
Structure chart for the
methods in the
Coordinate class

Class Diagram Elements

Most class diagrams are relatively simple three-row tables. For more complex challenges, we need a richer vocabulary in which to express design ideas. The following figure describes all the different pieces of a class diagram.



The following table gives a brief overview of these pieces.

Symbol	Meaning
ClassName	The name of the class goes at the top of the diagram
Template	Whether the class is a stereotype or template
+ public	The public modifier applies to attributes and operations
# protected	The protected modifier is a hash sign
- private	The private modifier is a minus sign
<u>static</u>	Underline if the attribute or operation is static
<i>virtual</i>	Italics are used to signify a method is virtual
: DataType	Both attributes and parameters can specify data types
[10]	If an attribute or parameter has more than one element
(parameters)	Operations may specify parameters
= defaultValue	Parameters and attributes can specify default values
{readOnly}	The read-only modifier can be applied to any variable

Figure 20.6:
Structure chart for the
methods in the
Coordinate class

Class Name

Rule 20.1 Class names go in the first row of a class diagram

A class name occupies the top row of every class diagram. There are some circumstances where the class name is the only data in a class diagram. For example, the class name might be the only thing known about a class early in the design process. Also, if one class refers to another class defined elsewhere, it is often appropriate to only include the class name of the referenced class.

Rule 20.2 Class names should be nouns because classes represent things

As a rule, functions and methods are verbs whereas classes are nouns. If you find yourself giving a class a verb name, then there might be a problem with your class.

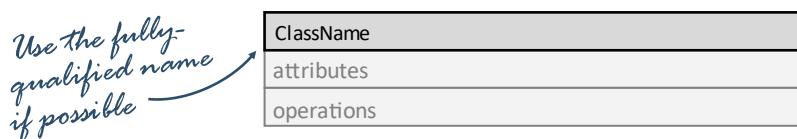


Figure 20.7:
The class name
component of a
class diagram

In the case where a class is part of a package, is nested in another class, or is in a specific namespace, then that information should be present in the class name. All this information together is called the “fully qualified name.”

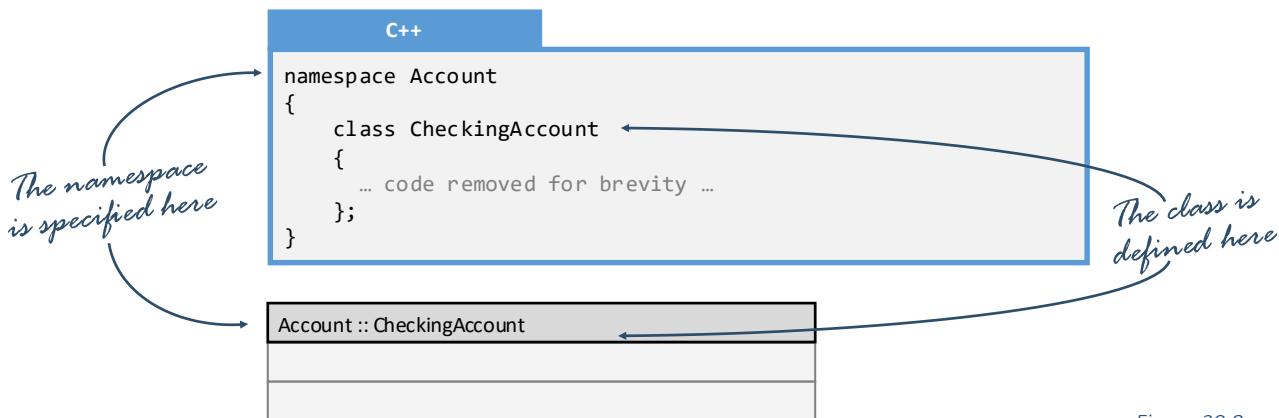


Figure 20.8:
Fully qualified name

For those languages allowing for stereotypes, generics, and/or templates, a rectangle with a dashed outline is used to provide this information. For example, consider the following C++ class definition and the associated class diagram.

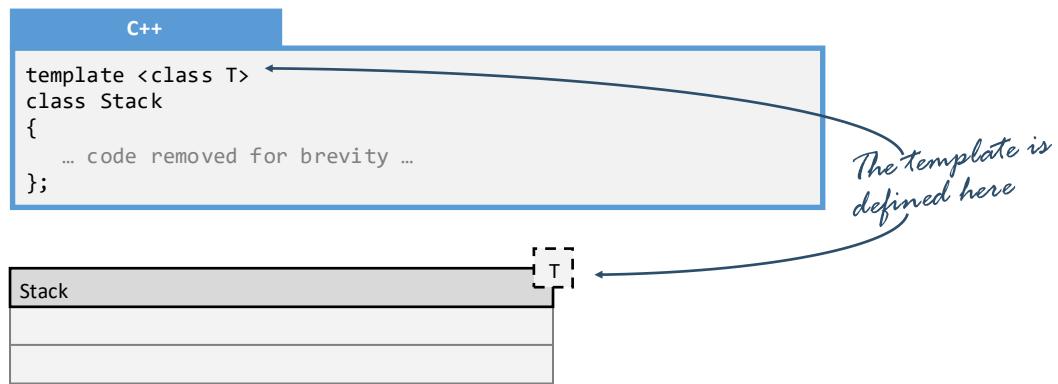


Figure 20.9:
Class name with a
template

Access Modifiers

A key component of OO design is the concept of data hiding. The basic idea is that the client or user of a class should not have to be aware of any more implementation details than is necessary to effectively use the class. Ideally, those details should be invisible or inaccessible to the client. Most OO languages provide a mechanism to specify visibility.

Rule 20.3 If an access modifier is provided, it must be one of the five: + - # ~ /

There are five access modifiers in class diagrams, each of which can be applied to both attributes and operations. These are public, private, protected, package, and derived.

Property	Description
+ Public	The attribute or operation is available to both the methods within the class as well as the users of the class. There are no access restrictions.
- Private	The attribute or operation is only visible to the methods within the class. The client and derived classes have no access.
# Protected	The attribute or operation is available to both the methods within the class and to the derived classes.
~ Package	The attribute or operation is available to all classes, functions, and variables defined in the current namespace.
/ Derived	The attribute or operation is not defined in this class but is defined in the parent class from which this is derived. It is possible to treat this attribute or operation as if it is defined in this class.

The four most common access modifiers are demonstrated in the following Java code and associated class diagram.

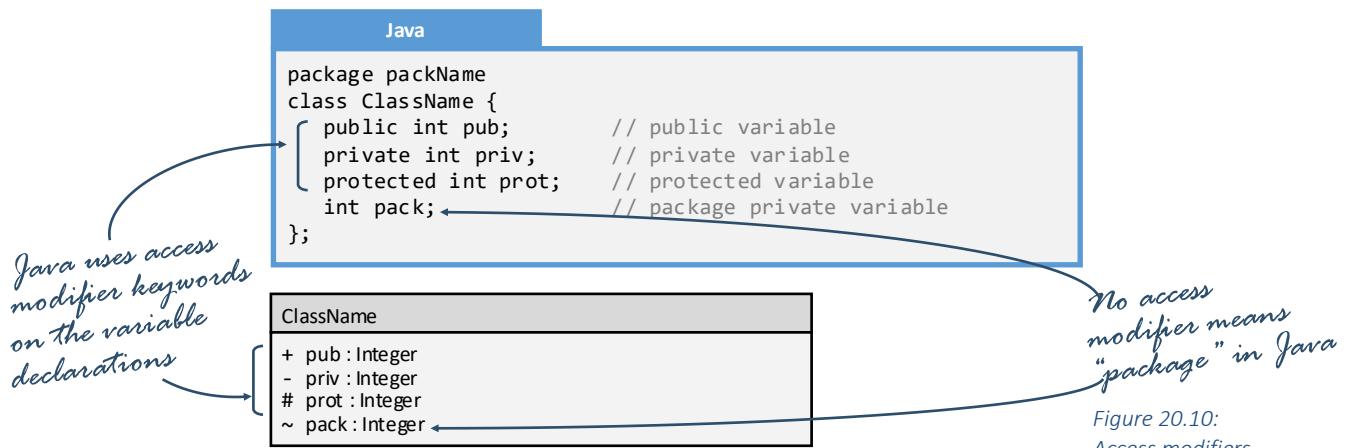


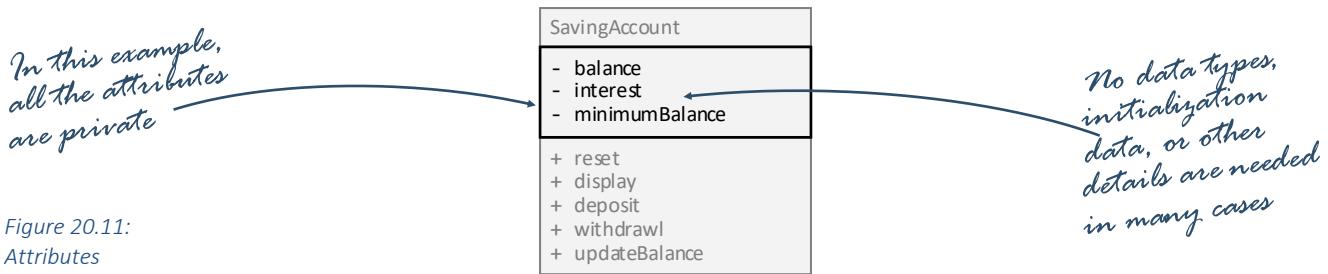
Figure 20.10:
Access modifiers

A derived attribute is not defined in the derived class, it is inherited. We include it in the class diagram only when we want to emphasize that the variable is accessible from a given class.

Attributes

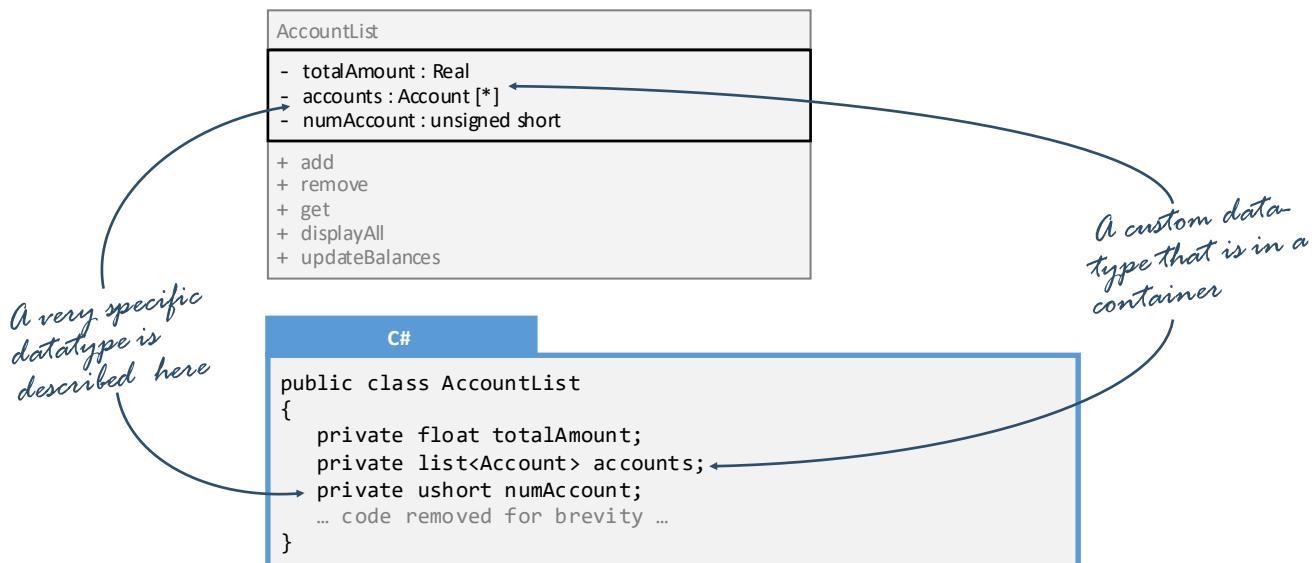
Rule 20.4 Attributes are normally on the second row, but must go above the operations

Attributes, also known as class states and member variables, are depicted in the second row of a class diagram table. In most cases, the access modifier and the variable name are enough to describe a given design.



Rule 20.5 Data types may be specified after the attribute with a colon preceding it

It is common to need to specify the data type of the attributes. The most commonly used types are `Boolean`, `Integer`, `String`, `UnlimitedRational`, and `Real`. If a member variable is an object, then the class name can be specified here as well. You can also indicate than a given attribute represents an aggregate data type such as an array. If that is the case, then you can specify the number of elements in square brackets. It is useful to reiterate at this point that class diagrams are design tools. Therefore, if you feel the need to modify the syntax to be able to express your design intent, then you may do so. For example, if you know that a given member variable will be an array of unsigned shorts, then specify that in the class diagram.



It is not uncommon to specify a default value. This is done with an equal sign and the default value at the end of the attribute line: `= 0`.

You can indicate that an attribute is constant or read-only by appending a `{readOnly}` label on the line. Finally, making an attribute underline specifies that it is static.

Figure 20.12:
Data types added to attributes

Operations

Rule 20.6 Operations are normally on the third row, and cannot go above attributes

Operations, otherwise known as methods or member functions, reside in the third and final row of a class diagram. They can have all the same access modifiers and decorators as attributes along with a few new ones specific to functions.

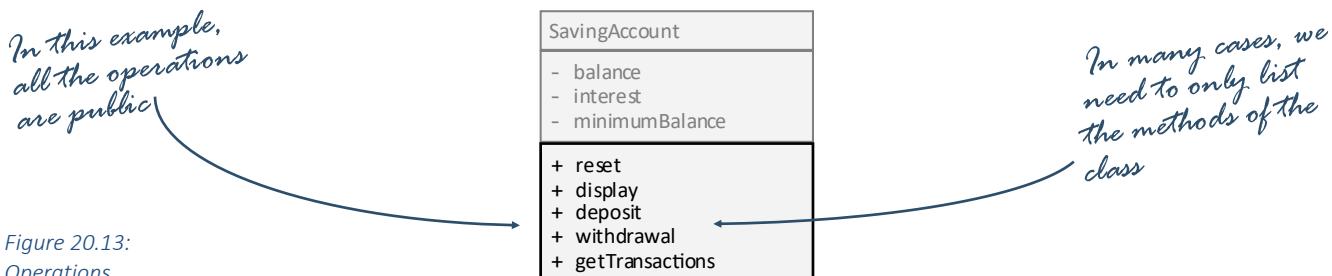


Figure 20.13:
Operations

Rule 20.7 Operation parameters may be specified in parentheses after the method name

It is often useful to specify the parameters to a method. These may or may not show the data type. Note that the data type is specified with a colon in the same way that data types are specified with attributes.

Rule 20.8 Return types may be specified for operations after parameters.

Though most programming languages specify the return type of a function or method before function/method name, class diagrams place that information at the end of the declaration. This, as with parameters, is optional.

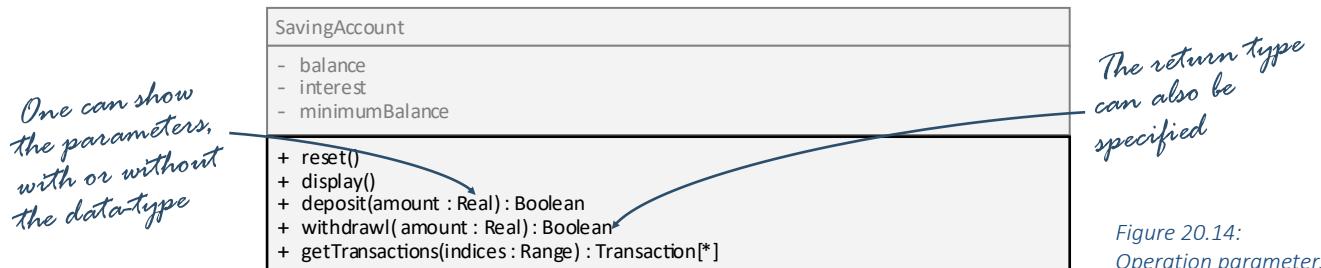


Figure 20.14:
Operation parameters

Rule 20.9 Operations can be embellished with three types: virtual, pure, and static.

There are several embellishments that are often helpful to include in class diagrams. The most common are virtual functions (in italic), pure virtual functions (italic with the `= 0` modifier), and static (underlined just like attributes).



Figure 20.15:
Embellishments

Designing with Class Diagrams

Design tools such as class diagrams are essential in the design process because they enable the programmer to paint with broad brush strokes without getting bogged down in the minutiae of detail that programming languages demand. This is particularly true with class diagrams; there is no place to specify the method implementation details!

Class design usually happens in four distinct stages. New designers usually get themselves in trouble to performing these stages out of order. These stages are:

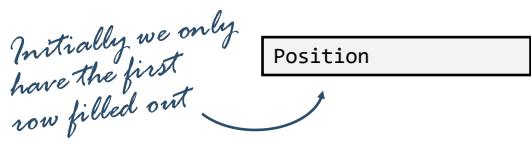
1. Specify the class name. Fill out the first row of the class diagram.
2. Identify the public operations. Fill out the third row of the class diagram.
3. Specify the attributes. Fill out the second row of the class diagram.
4. Fill in any remaining details. This includes private operations, parameters, data types, and anything else required to make your design intent clear.

Stage 1: Specify the Class Name

Many underestimate the importance of a well-chosen name. Some put off the variable, function, or class naming process until late in the process. At this point, they are frustrated because the program entity has so many purposes that it cannot be easily described in a single name. One program I worked on had a function named `formatLineCoreProc2()`. I remember the meeting where a developer defended the name choice with a straight face!

The name of a class should completely, concisely, and unambiguously describe what the class does. Because classes represent things, the class name must be a noun. If a class name has a conjunction such as “and” or “or” in the name, then this is a good indication that there is a problem! If you cannot come up with a good name for a class you are thinking of building, then there is no need to proceed to the other stages of design; you are already lost!

The name of a class should completely, concisely, and unambiguously describe what the class does.



If we were to build a class to represent the position of an entity in a 3D flying game, then we would start by just filling in the first row of the UML class diagram. In this case, since our class is representing a position, **Position** seems to be an appropriate name for the class.

Figure 20.16:
Position class: just the
class name

Stage 2: Identify Public Operations

After the purpose of the class has been clearly identified with a well-chosen name, the next step is to identify the public operations. This includes everything that the client or user of the class will see.

At this stage of the design process, it is important to get a deep understanding of how the class will be used. The designer of the class needs to carefully consider the system for which the class is meant to serve. Specifically, how will the system use this new class? If we don't get to the bottom of this question, then it is likely that our class will be difficult to use or not used to its full potential.

Identify the three main operations

Position
+ computeDistance
+ move
+ isInBounds

Figure 20.17:
The public operators of the Position class

How will the system use this class?

All the public operations to a class should be expressed in terms of the needs of the system. Going back to our **Position** class from Stage 1, how will our game use this class? After looking into our system design in more detail, it becomes apparent that it will move according to some velocity, it will be compared against other positions to find the distance between,

and it will determine whether it is out of bounds. Based on this, we can identify the public methods of our class.

Stage 3: Determine Attributes

Once we know what the class is designed to do (Stage 1, the class name) and have figured out how the client will interact with the class (Stage 2, the public operations), the next step is to determine what kind of state the class needs to maintain so the operations can be supported. This state is represented with the class attributes.

Classes fundamentally represent nouns, and those nouns are the attributes (or member variables if you prefer). The attributes of the class should capture all the states of the concept it is meant to represent, no more and no less. For example, if a class was to represent biological sex, then two or three states would be all that is needed. A text file representing millions of states would be less than ideal. The class should focus on efficiency. How much storage space is required to represent the concept? How much processing does it take to convert the attributes into the information needed by the operations? All of these things are important considerations.

The attributes of the class should capture all the states of the concept it is meant to represent, no more and no less

Three member variables, plus six variables shared with all other Position objects

Position
- x, y, z
- xMin, xMax
- yMin, yMax
- zMin, zMax
+ computeDistance
+ move
+ isInBounds

Figure 20.18:
The attributes of the Position class

Back to our **Position** example, there are three obvious attributes: the **x**, **y**, and **z** coordinates. However, after looking into the operations more carefully, it appears that some positions are not valid. We could add an **isValid** Boolean value to represent when the game entity is out of the arena or not initialized, or we could store that somehow in the **x**, **y**, and **z** variables. Here, we will choose the latter approach. Finally, the method **isInBounds()**

needs to know the bounds of the arena. We could add **xMin**, **xMax** variables, but that seems like overkill. We don't need one per **Position**, we need one per game. Thus, the Min/Max attributes should be static.

Stage 4: Fill in the Details

The final stage of the class design process is to fill in all the details necessary to define the class in a programming language. This includes identifying the parameters and return values of all the public operations. Though this is often done in Stage 2, there are frequently a few details that got skipped in the design process.

Stage 4 is also a good time to identify the data types of the attributes and to double-check that all the attributes are found. Again, this can be done in Stage 3, but it is a good idea to give it another consideration before finishing the process.

Finally, this last stage is the time to look for missing items. Is there a missing attribute, a missing public method, or a missing private helper function that we should include? Though almost certainly things will come up when we implement this class, problems are much easier to spot and fix in this early design phase.

At the end of Stage 4, our **Position** class looks finished.

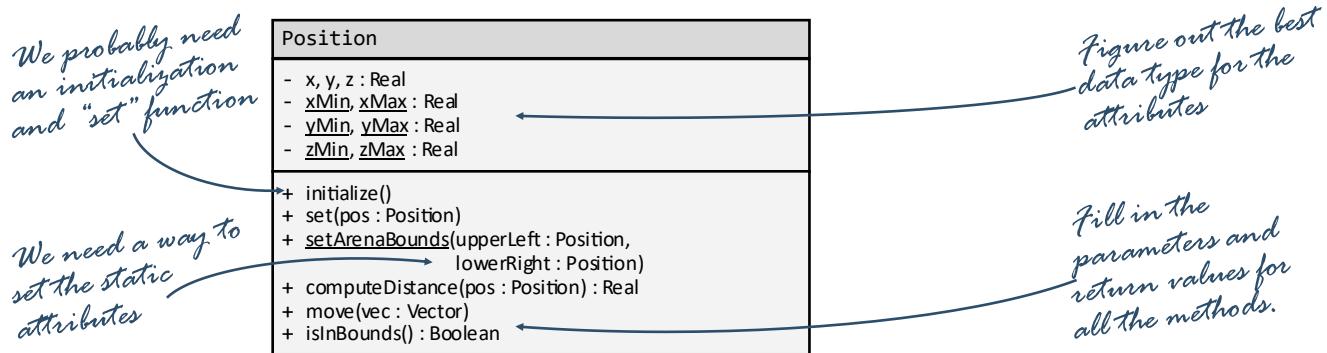


Figure 20.19:
The complete **Position** class

Examples

Examples 20.1: Class Diagram from Code

This example will demonstrate how to create a class diagram to represent an existing class definition.

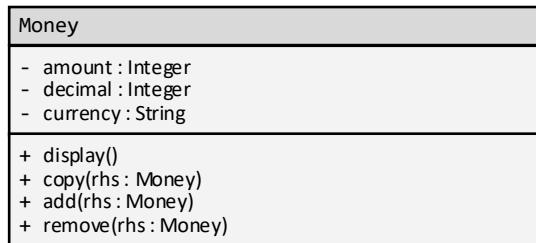
Problem

From the following Swift code, create a class diagram:

```
Swift
class Money {
    private var amount: Int
    private var decimal: Int
    private var currency: String?

    func display() {
        print("\(currency) \(amount).\(decimal)")
    }
    func copy(rhs: Money) {
        self.amount = rhs.amount
        self.decimal = rhs.decimal
        self.currency = rhs.currency
    }
    func add(rhs: Money) {
        self.amount += rhs.amount
        self.decimal += rhs.decimal
    }
    func remove(rhs: Money) {
        self.amount -= rhs.amount
        self.decimal -= rhs.decimal
    }
}
```

Solution



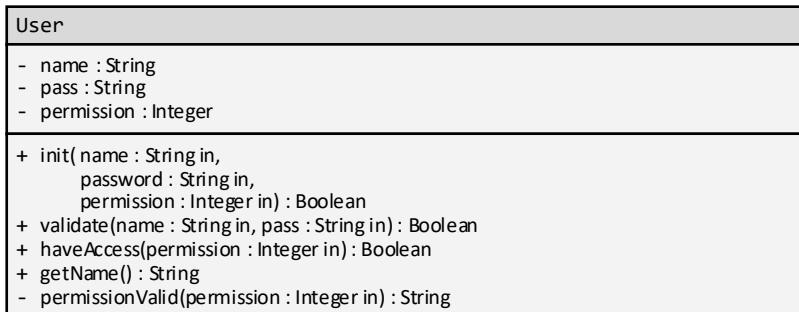
Notice that the syntax of the Swift programming language closely resembles the format of the class diagram. This makes it very easy to translate the design from one formation into another.

Examples 20.2: Code from a Class Diagram

This example will demonstrate how to convert a class diagram into a class definition in C++.

Problem

Create a class definition from the following class diagram:



Solution

```
C++

class User
{
    private:
        string name;
        string password;
        int permission;
        bool permissionValid(int permission) const;

    public:
        User(const string & name,
              const string & password,
              int permission);

        bool validate(const string & name,
                     const string & password) const;

        bool haveAccess(int permission) const;

        string getName() const;
};
```

Notice that the class diagram provides enough detail to complete the class definition. Of course, we do not have enough information from the class diagram alone to complete the methods of the class. This will require pseudocode or flowcharts.

Exercises

Exercises 20.1: Scenarios and Tools

For each of the following scenarios, select the best tool for the job.

Scenario	Tool
There are many private methods in a class that interact with the public methods in nontrivial ways.	
You would like to identify the public interfaces of a class.	
You are working on a single method in a class. This method requires complex decision making.	
You would like to analyze how data enter a class and how data leave a class.	
You are working on a complex algorithm involving many loops in a method of a class.	
You would like to specify the member variables of a class.	

Exercise 20.2: Identify Parts of a Class Diagram

Draw the symbols corresponding to the parts of a class diagram.

Description	Symbol
A public member variable of type <code>string</code> with the name <code>fileName</code> .	
A protected method named <code>isValid()</code> returns a Boolean and takes an integer parameter: <code>age</code> .	
An integer variable is inherited from the base class. The name is <code>accountNumber</code> , which is an integer.	
A Boolean variable named <code>isMale</code> is visible to all members of the namespace.	
The class name is <code>Checking</code> and it is a type of <code>Account</code> .	
There is a private static floating-point number named <code>exchangeRate</code> .	
The string variable <code>clientName</code> is visible to all members of the class and the derived classes is well.	
A class is a container and has two templates: <code>Key</code> and <code>Value</code> .	

Exercises 20.3: Class Diagram Errors

Identify the errors in the following class diagram:

```
GetDate
+ void setDate(int d, int m, int y)
# isValid() const
/ virtual display()

~ day : Integer
- month : Integer
$ year : Integer
```

Exercises 20.4: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Class diagrams describe the operations of a class.	
Class diagrams are a Logical view.	
Class diagrams describe nouns.	
Class diagrams can represent algorithms.	
Class diagrams are three-column tables.	
Class diagrams only work with a couple of programming languages.	
Class diagrams describe how data flow through the system.	
Class diagrams are three-row tables.	
Class diagrams describe the attributes of a class.	

Problems

Problem 20.1: Institution Class

Create a class diagram matching the following code:

```
Kotlin
class Institution {
    private val name: String
    private val url: String

    fun sendMessage()

    fun getName(): String

    fun setName(newName: String)

    fun compare(rhs: Institution): Boolean

    private fun validateURL(): Boolean
}
```

Problem 20.2: Time Class

Translate the following class diagram into a programming language of your choice.

```
Time
# minutesSinceMidnight : Integer
- timeFormat : FormatEnumeration

+ initialize()
+ display()
+ set(hour : Integer, minute : Integer, second : Integer)
+ difference(rhs : Time) : Integer
+ getHour() : Integer
+ getMinute() : Integer
+ getSecond() : Integer
- convertToMinuteSinceMidnight(hour : Integer,
                                minute : Integer, second : Integer)
```

Problem 20.3: Credit Card Transaction

Create a class diagram matching the following scenario:

A credit card transaction is a single instance of a credit card purchase. This transaction class will be used in a credit card class where an array of transactions will be stored. The credit card class will display the complete collection of transactions in a register, will sum up the various transactions to compute a balance, and will filter the transactions matching a certain criterion.

Each transaction will contain the attributes you may find on your credit card statement.

Hint: Look at your statement this month to see what those attributes are and how they are depicted.

Problem 20.4: Score Class

Create a class diagram matching the following scenario:

A video game contains a score class. This class will represent the score the current player has earned at a given moment in the game. The score will need to draw itself in the upper-left-hand corner of the screen. The leaderboard class will need to be able to request the score at the end of the game so it can see if the current game ranks among the best played.

The score will constantly be updated as the game progresses. Every second of gameplay, 1 point is added to the score. If the player makes it through a checkpoint, then 20 points are added. If the player hits a wall, then 5 points are deducted. If the player hits an obstacle other than a wall, then 50 points are deducted.

Problem 20.5: Recipe Item

Create a class diagram matching the following scenario:

A recipe program contains a collection of recipes. Each recipe consists of a collection of recipe items. Your class will represent a single recipe item.

Each recipe item will contain the attributes you may find on your favorite recipe.

Hint: Look at your favorite recipe or look at a recipe on the internet to see what those attributes are and how they are depicted.

Challenges

Challenge 20.1: Credit Card

Create a class diagram matching the following scenario:

A credit card class represents all the information necessary to represent a credit card on a personal finance software package. What would you, the user, wish to do with your credit card report on your personal financial software?

This credit card class contains a collection of individual credit card transactions. Please see Problem 20.3 for an idea of what that looks like. Additionally, a host of other attributes and operations are needed.

Hint: Look at your credit card statement or your credit card website as a starting point.

Challenge 20.2: Recipe

Create a class diagram matching the following scenario:

Create a recipe class for a list-management mobile application. This class will consist of collection of recipe items (please see Problem 20.5) and a collection of other attributes.

Hint: Please look at your favorite recipe software, a paper recipe, or a recipe you have downloaded from the internet to get an idea of what kind of attributes and operations are needed here.

Challenge 20.3: Ship

Create a class diagram matching the following scenario:

Imagine a video game where a ship navigates a 3D world filled with walls and moving obstacles. You are to create a class representing this moving ship.

Hint: What attributes and operations would be needed in this ship class?

Fidelity is the suitability of a class in representing a design concern, analogous in many ways to the modularization metric of cohesion

Fidelity is an encapsulation metric very similar to the cohesion metric of modularization. Both are designed to measure whether the programming construct stands for that which it is designed to represent. This, however, is where the similarities end. Cohesion measures functions which are verbs whereas fidelity measures classes which are nouns. Cohesion is about whether a function does what it says it does whereas fidelity is about the quality of the virtual representation of a physical thing.

Fidelity is the suitability of a class in representing a design concern

One of the first challenges early computer engineers faced was how to represent numbers in an electronic medium. From this early work, we get the base-2 representation of integers and the floating-

point representation of real numbers. Both systems represent numbers to a reasonable degree of fidelity; they work well enough for most applications. The next challenge was how to represent text. This was accomplished by assigning numbers to each letter and symbol in a system we call American Symbolic Code for Information Interchange (ASCII) today. There is a common thread in both stories: the quest to identify an appropriate virtual representation for a concept which is not natively binary. We call this concept fidelity. Fidelity is the suitability of a class in representing a design concern.

Levels of Fidelity

Classes are blueprints for variables (called objects), describing how the variable organizes data so they can represent something the programmer cares about. Each of these variables is nothing but a collection of bits in memory (or 1s and 0s or, if you want to go deeper, +0.5 volts to -0.5 volts on a computer chip). It is the goal of class designers to organize these bits so they can stand in for a design concern. This is a remarkable task! How can a collection of bits represent the note D#? How can a collection of bits represent the color indigo? How can a collection of bits represent a location, a person, or an emotion? Perhaps it may be a bit of hubris that programmers will even try such a thing, but alas it is our charge to do so. The metric of fidelity helps programmers understand the suitability of this representation in four levels of quality: complete, extraneous, partial, and poor.

Level	Description
Complete	States of the class completely match the design concern.
Extraneous	There are extra states but none missing.
Partial	There are missing states but nothing extra.
Poor	The class poorly represents the design concern.

Complete Fidelity

Complete fidelity is when there is a direct mapping between states in the class and states in the design concern

there are two states (open or closed) and one bit to represent it (0 and 1), we have complete fidelity. The bank now wishes to add several other states to the status class: whether the account is locked, if the account is suspended, and whether the account is in privileged standing. To keep this class complete, it is necessary to add states or combination of states to represent all possible account status levels.

It is possible to mathematically prove a class has complete fidelity when there are a finite number of design concern states. Since data are stored in a binary computer using bits, this mapping is easiest when there are 2^n states. When the number of states in the design concern does not fit neatly into a power of 2, it is necessary to formally specify what the extra states mean. Going back to our account example, there are five states. We will represent this with a single byte containing 8 bits or 256 unique states. To achieve complete fidelity, all 256 states need to be mapped to a design concern.

When working with unbounded design concerns, work with the stakeholders to determine the practical limits. These should be represented with complete fidelity

It is not possible to mathematically prove a class has complete fidelity when there are an unbounded or infinite number of design concern states. The best one can hope for is to approximate the design concerns. This does not represent as big of a hurdle as one might think. While there are an infinite number of GPAs between 0.0 and 4.0, few users need more than 2 digits of accuracy beyond the decimal point. If the stakeholder demands 3 digits to be safe, then there are 4,000 possible combinations. This can easily be represented with an integer where the extra states are assigned to the "0.000" GPA state. In summary, when working with potentially unbounded design concerns, turn to the stakeholders to determine the practical limits and represent those limits with complete fidelity.

The gold standard for class fidelity is completeness, where there is a direct mapping between the states in the class and the states of the design concern. Perhaps this is best explained by example. Consider a class representing the status of an account. Since

Mapping of account status to class state

0	Open and normal
1	Open and locked
2	Open and suspended
3	Open and privileged
4-255	Closed

Extraneous Fidelity

A class can be classified as having extraneous fidelity when two conditions are met: when every state of the design concern is represented in the class, and when some states in the class do not correspond to a design concern. In other words, some combination of bits maps to a state that does not align with a design concern.

*Negative rates
should not
be possible*

Interest Rate
- rate : Float
+ get(): Float + set(rate : Float)

Figure 21.1:
Extraneous fidelity

Extraneous is when there are extra class states but none missing

Consider a class designed to represent the interest rate on a credit card. Since an interest rate is a decimal number, it makes sense to represent this as a floating-point number. The problem with this approach is that there are many possible floating-point numbers which do not make sense for an interest rate. Even the worst of loan sharks would not charge an interest rate of over 100%. Similarly, credit cards never have a negative interest rate. This implementation therefore has extraneous fidelity.

A better design would be to incorporate the fact that the valid range for credit cards is $50\% \leq \text{rate} \leq 0\%$. Furthermore, interest rates are never represented with more than three digits of accuracy. From this, we can see that there are 50,000 possible values. If we extend the range slightly to 65.535% as the maximum interest rate for a credit card, we can see that it will fit nicely into a short. This implementation will have higher fidelity.

The most common cause of extraneous fidelity is when a primitive data type is used to represent a concept that represents more states than the design concern calls for. Why was the primitive data type used? Perhaps the programmer thought it was the “obvious” choice. Perhaps it conveniently mapped to how the property would be used in the application. More than likely it occurred because the programmer did not realize or consider the dangers that can result from extraneous classes.

Extraneous fidelity is a potential quality problem

When a class exhibits extraneous fidelity, there are values or combinations of values that the member variables can have which do not correspond to a design concern. In other words, when these values are realized in the code, we have a bug. While it may be possible to minimize the impact of the bug in the software through a variety of means, the bug cannot be completely eliminated unless we address the extraneous fidelity issue.

*It's should not
be possible to have
-300 children*

NumberChildren
- num : Integer
+ get(): Integer + get (num : Integer)

Figure 21.2:
Extraneous fidelity

Partial Fidelity

Partial fidelity is when there are missing design concerns but no extraneous states

A class can be classified as having partial fidelity when some states of the design concern are missing but there are no extraneous states. In other words, a class does not have the capacity to represent all that it is meant to represent.

This might be best explained by example. In the 1970s and 1980s, it was popular to represent dates as a 2-digit number. For example, 1982 would be represented as just 82. This 2-digit representation was considered good enough because no one would confuse 1882 with 1982. As the 1990s approached, computers became more powerful and memory was more abundant. At this time, best practices dictated that dates should be represented as 4-digit numbers. While little new code written in the late 1990s employed 2-digit years, plenty of legacy systems still did. In 1998, many people started asking what would happen on the 31st of December, 1999. It quickly became apparent that many systems would treat the next day as the 1st of January 1900. This became known as the year-2000 bug, also known as Y2K. Globally, between \$200 and \$600 billion was spent fixing these bugs, mostly consisting of moving from a 2-digit to a 4-digit year. What was the core cause of this problem? Partial fidelity, of course!

In a more recent example, the popular video service YouTube used a signed integer to keep track of the number of times a video was played. This is not a problem, the maximum value for an integer is a little over 2 billion. What video would be watched 2 billion times? On the 4th of December, 2014, the artist Park Jae-sang (Psy) blew through that limit with his music video "Oppan Gangnam Style." On that day, his video went from just over 2 billion views to -2,147,483,647. As with the Y2K bug, the core problem was partial fidelity.

Partial fidelity is a suitability problem, potentially rendering the software as not fit for use

Any instance of partial fidelity is dangerous because valid and potentially important functionality required by the customer may not be present in the software. There are times when this is unavoidable: computers have a difficult time representing infinite sequences after all. In cases like these, it is important to document these limitations, create software checks to gracefully recover from them when they happen, and have a plan in place to fix the problem if the need arises.

```
C++  
ViewCounter & ViewCounter :: operator++ ()  
{  
    // if this assert fires, we need to move  
    // our counter type to a long integer  
    assert(counter != INT_MAX);  
  
    ++counter;  
    return *this;  
}
```

Figure 21.4:
Partial fidelity with checks
to fix if need

While partial fidelity is most often worse than extraneous fidelity, it is not always the case. It can be possible that a missing unlikely or inconsequential state could be better than a mainstream instability caused by extraneous fidelity. In other words, the relative severity of these levels of fidelity is application domain specific.

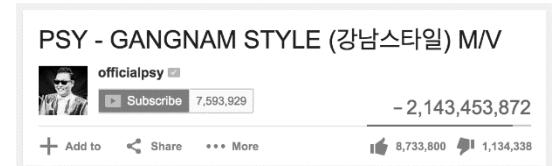


Figure 21.3:
Integer overflow from
partial fidelity

Poor Fidelity

A class can be classified as having poor fidelity when it poorly represents the design concerns. It should be a goal of all designers to never encounter a class with poor fidelity, let alone create one ourselves!

Poor fidelity is when the class poorly represents the design concerns

Poor fidelity is easy to identify: it is the combination of extraneous and partial. If a class fails to cover all the cases of a design concern and contains extra cases, then it is an example of poor fidelity.

There are a surprising number of poor classes in common use today. Several are subtle, with the extraneous cases being improbable and the partial cases being nonmainstream. Can you spot the poor fidelity in the following class diagram?

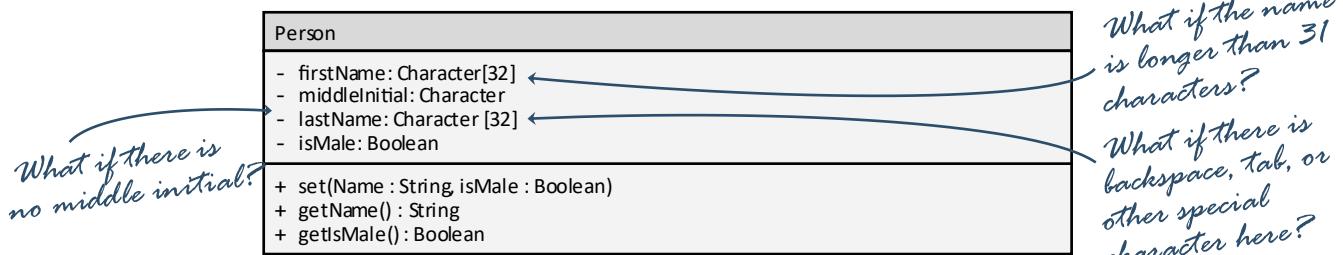


Figure 21.5:
Poor fidelity

There is a serious extraneous fidelity problem with this seemingly simple class. What if a space, tab, newline, or backspace character is put in the character stream? If you look in the ASCII table, there are many glyphs that are not permissible in a person's name. This code as it currently stands does not reject such values, making it possible to put a **Person** object in an invalid state. This is a bug waiting to happen!

There is also a host of possible partial fidelity problems. The class does not handle the case when the user has no middle initial. The class cannot handle first names or last names that are longer than 31 characters in length (note that a 32-character buffer has space for a 31-character string leaving one space for the null character). It cannot handle the case when there are more than four parts to a name, or those cases when an individual only has a single name. While these cases may seem unlikely (everyone I know has a first name under 31 characters), they will certainly be encountered by real users if this code becomes widely adopted. Furthermore, we are likely to discover that a huge segment of the population will be affected by perhaps the most serious problem of all: Asian names with Chinese, Japanese, Korean, Hindi, Cyrillic, and Arabic characters cannot be represented with the **Character** data type.

To avoid poor fidelity, it is necessary to have an in-depth understanding of how the class is going to be used in the application. It is worthwhile to spend a few minutes brainstorming about cases that can lead to extraneous or partial fidelity. It is much easier to fix these defects while drawing a class diagram than months after the code was written when a user has encountered a bug!

Variations

There are two encapsulation metrics related to fidelity: class cohesion and the single responsibility principle.

Class Cohesion

Recall the modularization metric of cohesion:

Cohesion is a measurement of how well a unit of software represents one concept or performs one task

Clearly a class is a “unit of software” just as much as a function. There have been several attempts to apply the concept of cohesion directly to classes. Most are centered on how tightly the methods of a class are bound to each other.

One application of cohesion to classes is called “lack of cohesion of methods (LCOM).” If two methods share a disjoint set of member variables, then they are probably loosely connected. This means the overall cohesion level of the class is low. Note that this is an indirect measure of whether a single class represents more than one separate design concern. In other words, if one class is used to represent two independent concepts, then it will probably have many methods which are unrelated and therefore have a low LCOM score.

Classes are more cohesive if the methods operate on the same data

Another application of cohesion relates to how many of the member variables are utilized by each member variable. When every method utilizes every attribute, then the cohesion score is 0 referring to “perfect cohesion.”

Single Responsibility Principle

Defined by Robert Martin in 2003, the “single responsibility principle (SRP)” is the ‘S’ in the SOLID design guidelines. SRP states that a class should do one thing and one thing only. If it stores information, it should not process it. If it processes information, it should not handle I/O. Each of these different larger tasks are a “responsibility” and a class should focus on doing just one of them.

An indicator that a class has violated the SRP is when a single class or even a single method within a class has more than one reason to change. It also means that a single bug fix or feature update should not impact more than one method or class. In fact, Martin defines the single responsibility principle in terms of motivation for change:

Gather together the things that change for the same reason. Separate those things that change for different reasons.

Examples

Example 21.1: Extraneous Fidelity

This example will demonstrate how to identify extraneous fidelity from a class diagram, and then how to improve the fidelity.

Problem

Classify the level of fidelity of the following class meant to represent the position on a chessboard.

Coordinate
- row : Integer
- column : Integer

+ getRow() : Integer
+ getColumn() : Integer
+ set(row : Integer, column : Integer)

Solution

Since rows and columns are two integer values, it is not surprising that the **Coordinate** class has two integer member variables. However, there are many states represented in the class which do not represent valid coordinates. In fact, there are 18 quintillion possible states represented by two 32-bit integers when only 64 states are needed. This means that the class is extraneous.

A better solution would reduce the number of possible states. Since byte is the smallest unit of data represented in most programming languages and it has 256 possible, it is more than enough for our needs. We will therefore call 0...63 as valid coordinates and 64...255 as the zero coordinate (0, 0). Finally, we need a couple of private methods to translate between our internal representation and the format the client needs.

Coordinate
- value : Unsigned Char
+ getRow() : Integer
+ getColumn() : Integer
+ set(row : Integer, column : Integer)
- valueFromRC(row : Integer, column : Integer) : Unsigned Char
- rowFromValue(value : Unsigned Char) : Integer
- columnFromValue(value : Unsigned Char) : Integer

Example 21.2: Partial Fidelity

This example will demonstrate how to identify partial fidelity from a class diagram, and then how to improve the fidelity.

Problem

Classify the level of fidelity of the following class meant to represent an ingredient on a recipe.

Ingredient
- quantity : Positive Float
- unit : Units
- item : ItemType
...

Solution

The first step is to determine if we have extraneous fidelity. Since the **quantity** attribute is positive and presumably any positive number is valid, then we have no extra states with that member variable. The **unit** and **item** attributes are more difficult to pin down. The fidelity of these items depends on the fidelity of the **Units** and **ItemType** class. Assuming that these have complete fidelity, we can see that **Ingredient** does not exhibit extraneous fidelity.

The second step is to determine if we have partial fidelity. In other words, are there ingredients in an ingredient list which cannot be represented with these three member variables? A bit of research reveals the following ingredients:

Ingredient item
1 cup boiling water
½ cup oil (vegetable or canola oil)
1½ cups warmed milk (110–115 degrees Fahrenheit)
1 quart salsa (not too spicy!)
1¼ tablespoon packed brown sugar

From these ingredients, we can see that we are missing an adjective associated with the item such as “boiling,” “warmed,” or “packed”. We are also missing a field to put generic notes such as “vegetable or canola oil,” “110–115 degrees Fahrenheit,” or “not too spicy!” The absence of these fields yields partial fidelity. We can fix this defect by adding appropriate fields (being careful not to introduce extraneous problems in the process).

Ingredient
- quantity : Positive Float
- unit : Units
- item : ItemType
- adjective : String
- notes : String
...

Example 21.3: Poor Fidelity

This example will demonstrate how to identify poor fidelity from a class diagram, and then how to improve the fidelity.

Problem

Classify the level of fidelity of the following class meant to represent an authenticated user on a secure system:

User
- name : String
...

Solution

Since any string can fit into the `name` variable, and since presumably many names are not legal identifiers for the system (such as a name consisting of only spaces or tabs), then there are several states possible that are not legal. This makes for extraneous fidelity.

Notice that there is no obvious way to represent the invalid or unauthenticated user. If we choose a keyword such as “invalid,” then the possibility exists that a user could choose that label for his/her username. This means that there are required states in the system that are not represented with our class. The result is partial fidelity.

Since this class exhibits both extraneous and partial fidelity, then poor fidelity is the most accurate classification.

A better solution would be to store not a username but rather an ID. This makes it possible to identify the unauthenticated user as `id = 0`. It also makes it easy to ensure that every `id` value maps to a valid state. This can be done by treating as invalid all values for `id` that are greater than the number of usernames. This username table is a necessary addition, enabling us to translate between `String` usernames and integer `ids`.

User
- id : Unsigned Integer
- usernames : String [*]

- getNameFromID(id : Unsigned Integer) : String
- getIdFromName(name : String) : Unsigned Integer

Exercises

Exercise 21.1: Define the Level of Fidelity

From memory, recite the definition of each level of fidelity:

Level	Definition
Complete	
Extraneous	
Partial	
Poor	

Exercise 21.2: Identify the Level of Fidelity

Identify the level of fidelity of the following class designed to represent an individual's marriage status: married or not married.

Marriage Status
- isMarried : Boolean
+ setMarried()
+ setNotMarried()
+ isMarried(): Boolean

Exercise 21.3: Identify the Level of Fidelity

Identify the level of fidelity of the following class designed to represent the position of a ship in a 3D game.

ShipPosition
- x, y, z : Float
- isDead : Boolean
+ getPosition() : (float, float, float)
+ setPosition(x : float, y : float, z : float)
+ kill()
+ isDead() : Boolean

Exercise 21.4: Identify the Level of Fidelity

Identify the level of fidelity of the following class designed to represent a person's full name: first, middle, and last. Here, the name parameter is a C string, meaning characters after the null terminator are ignored. Also, `setName()` will reject names which are invalid (such as those consisting of only spaces or non-printable characters).

Name
- name : Character [8]
+ getName : String
+ setName(name : String)

Exercise 21.5: Identify the Level of Fidelity

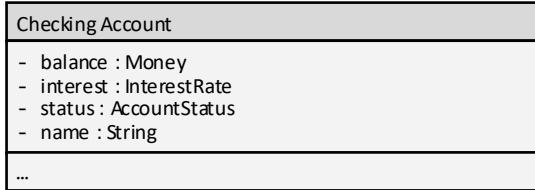
Identify the level of fidelity of the following class designed to represent a player on a basketball team. This is meant to include both the player's position (point guard, shooting guard, small forward, power forward, and center) and the player's number assigned to that position.

BasketballPlayer
- position : String
- number : Integer
...

Problems

Problem 21.1: Identify the Level of Fidelity

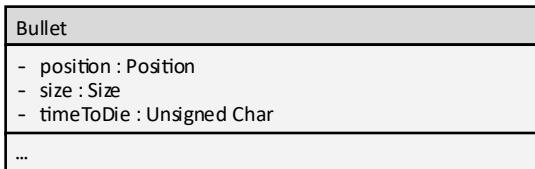
Identify the level of fidelity from the following class diagram meant to represent a checking account. This includes an account balance, interest rate, and status. You can assume that the enclosed classes (`Money`, `InterestRate`, and `AccountStatus`) have complete fidelity.



If there are fidelity defects in the above class, suggest a design to rectify them.

Problem 21.2: Identify the Level of Fidelity

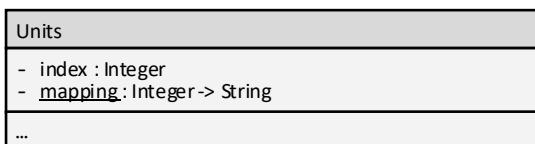
Identify the level of fidelity from the following class diagram meant to represent a bullet in a 3D game. You can assume that the enclosed classes (`Position` and `Size`) have complete fidelity.



If there are fidelity defects in the above class, suggest a design to rectify them.

Problem 21.3: Identify the Level of Fidelity

Identify the level of fidelity from the following class diagram meant to represent an ingredient in a recipe program. Here, the static member variable mapping contains the following collection: {1:cups, 2:teaspoons, 3:tablespoons, 4:ounce, 5:pound}.



If there are fidelity defects in the above class, suggest a design to rectify them.

Problem 21.4: Class Design

Create a class diagram exhibiting the highest possible degree of fidelity to match the following design concern:

A spaceship in a 3D game has several attributes: position, velocity, and orientation. It also has a ship type (one of 3 types: Beginner, Intermediate, and Advanced), a status (one of 5 types: Unharmed, Light Damage, Heavy Damage, Critical Damage, Destroyed), fuel (value from 0...100), and name.

Identify and justify the level of fidelity of your design.

Problem 21.5: Class Design

Create a class diagram exhibiting the highest possible degree of fidelity to match the following design concern:

A recipe consists of several things: a collection of ingredients, a collection of process steps, a name, and a description. It also has an estimated prepare time, estimated cook time, cost, and number of calories. The user can categorize the recipe (one of 5: Appetizer, Salad, Soup, Main Course, Dessert), specify its use (one of 4: Breakfast, Lunch, Dinner, Snack), and rate it according to quality (up to 5 stars).

Identify and justify the level of fidelity of your design.

Problem 21.6: Class Design

Create a class diagram exhibiting the highest possible degree of fidelity to match the following design concern:

A financial institution has several properties: a name, a website address, a type (Bank, Credit Card, Investment, Other), a street address, and a phone number. The user can also store a username and password combination. We will keep track of the date that the first account was opened, the most recent date of usage, and a list of accounts associated with this institution.

Identify and justify the level of fidelity of your design.

Challenges

Challenge 21.1: Credit Card Class Design

Create a class diagram exhibiting the highest possible degree of fidelity to match the following design concern:

A credit card class represents all the information necessary to represent a credit card on a personal finance software package. What would you, the user, wish to do with your credit card report on your personal financial software?

Please see Challenge 20.1 for a starting point in your design.

Identify and justify the level of fidelity of your design.

Challenge 21.2: Shopping List Class Design

Create a class diagram exhibiting the highest possible degree of fidelity to match the following design concern:

A class that represents a grocery shopping list. You may need to do some research to see what goes on a shopping list. Look for how people manage a shopping list using paper as well as other applications that automate this task for you.

Identify and justify the level of fidelity of your design.

Challenge 21.3: Video Game Class Design

Create a class diagram exhibiting the highest possible degree of fidelity to match the following design concern:

One game entity from a video game of your choice from the 1980s. You may need to conduct some research to discover attributes of the entity.

Identify and justify the level of fidelity of your design.

Robustness

Chapter 22

Robustness is the degree of resistance a class has from being placed in an invalid state or sending invalid data to the client.

Robustness is a property of class design quality, but it by no means is unique to classes. It is equally pertinent with algorithm design, modularization design quality, and even system design. Why, then, are we discussing it with classes? The reason is this: though robustness is important to all levels of software design, classes give us special mechanisms to offer superior robustness guarantees.

Robustness is a measure of how well a class avoids being a source of bugs

Robustness is the degree of resistance a class has from being placed in an invalid state or sending invalid data to the client. In other words, it is a measure of how well the class avoids introducing bugs into the code base. Though a traditional built-in data type may allow for an age to be set to -10, a class should make such nonsense impossible.

Fundamentally, robustness is not a measure of design quality but rather software quality. In other words, it considers class design, class implementation, as well as verification and validation. To ensure robustness, quality must be baked into every step of the process. As with other metrics, robustness is defined by a series of quality levels.

Levels of Robustness

Robust code is code that resists accidental or malicious attempts to place it in invalid states. Robust code can be relied upon to always produce valid output and exhibit predictable behavior. In most programming languages, classes offer mechanisms to facilitate robustness. The most important of these are access modifiers, enabling class designers to protect member variables from direct access. In effect, this builds a wall around variables, preventing them from being placed in invalid states.

The levels of robustness only indirectly refer to degree of quality of the design or the code. Instead, they refer to the degree to which the code has been validated.

Level	Description
Proven	The design was formally proven to be robust.
Resilient	Thoroughly white box tested.
Strong	Rigorously and systematically black box tested.
Tested	Ad hoc black box tested.
Fragile	Only tested in very restricted circumstances.

Fragile Robustness

Fragile is when a class can only be relied upon to function as expected in very restricted circumstances

The lowest degree of fidelity is fragile where the class can only be expected to work in very restricted circumstances. Note that the class may have a higher degree of reliability, but that higher degree has yet to be verified.

Fragile is the default degree of robustness, the degree in which we assume all software when it has first been written. This is often the case early in the development cycle. We get the code to compile and pass a simple test. "It's done," we exclaim. Probably not! At this point, it is probably just fragile. How often do we do this and never pursue a higher degree of robustness?

There are three reasons why a class may remain fragile long into the development process and perhaps even after it has been released to the customer. The first is that the class is deemed by the developer to be too simple or straightforward to be worth more effort. We call this "face validity," defined as the degree to which a programming construct such as a class appears by inspection to be correct. In other words, we are relying on experience and intuition to determine the quality of code. Not to discount the worth of a programmer's "gut check," but it is not enough to develop truly robust software.

When code appears to be correct but is not checked by experiment, we have face validity

The second reason why a class may remain fragile is that only cursory checks have been made. We have all done this from time to time, sending just a few inputs through a function or running the program a couple times before pronouncing it to be done. This is certainly a necessary step in the validation process but should not be the last one!

The final reason why a class may remain fragile is because of the difficulty of running more rigorous tests. This often happens when a class is deeply intertwined in a program with many external and internal dependencies. Isolating the class with a driver program or a more formal test harness can be difficult. Faced with no obvious path forward and faced with pressures to move on to the next task at hand, programmers can be tempted to proclaim that more formal validation cannot be done,

Though a class may be categorized as fragile without any testing, it is common for some quality activities to take place at this level. The most common of these is an activity called code review. This is useful for finding bugs and increasing quality, but does not provide the quality assurances found in the higher levels of robustness.

Code Review

A code review, also known as a peer review, code inspection, walkthrough, or simply “review,” is a careful reading of source code for the purpose of finding bugs. In many software life cycle models and as a matter of policy in many organizations, code reviews represent an important milestone in the development of software. It is often the case that code cannot be merged into the main codebase without first undergoing a code review.

There are several potential benefits to conducting a code review. These include the following:

- **Reduced number of defects:** A variety of studies have been conducted on code reviews through the years. Though the results vary wildly, most point to a defect find rate of between 40% and 60%. In other words, about half of the total number of bugs can be found through a code review.
- **Ego effect:** When developers know that their code will be read by one of their peers, they tend to be more careful about the design. This has shown to have a positive effect on reducing the number of defects in software.
- **Knowledge Transfer:** When team members and co-workers review each other’s code, their overall knowledge of the project increases. This can lead to less problems with code integration (a.k.a. making your code work with others) and increase team adaptability (a.k.a. being able to fix bugs in other people’s code).

Even when higher levels of robustness are needed for a given class, all code can benefit from code reviews.

Performing Code Reviews

Though there are several accepted methodologies governing how to perform code reviews, they all boil down to the same basic steps:

1. **Prepare the code for review.** The first step is to make sure that the code conforms to the organization’s style guidelines. Additionally, comment the code so the intent is made clear and the “why’s” are correctly explained. This step serves two purposes. First, it causes the developer to make a final pass through the code before turning it over to the reviewer so the silly mistakes are fixed. Second, it makes the reviewer’s job easier because the code is more readable.
2. **Find a reviewer.** The best code reviews occur by someone other than the code author. How do we convince others to review our code? By being willing reviewers of their code!
3. **Review small chunks of code at a time.** The general rule is that a review session should last no more than an hour and cover no more than 400 lines of code. Studies have shown that if the inspection rate is greater than 500 lines of code an hour, the number of defects detected during the review process drops sharply. Studies have also shown that “reviewer fatigue” sets in after an hour. This means that it becomes difficult for the reviewer to maintain focus after staring at code for a full hour.
4. **Use a checklist.** Each code review should have specific goals and look for specific things. Look for uninitialized variables, comments that don’t match the corresponding code, variables that change their meaning/use in a function or class, memory that is allocated but never freed, dangling pointers, performance problems, and other domain specific things.
5. **Record recommendations.** When a defect or improvement opportunity is found, it is good to communicate that with the code author. It is better to record the findings in an e-mail or some other written record. It is best to create a comment in the code itself and then check that comment into the project. When this is done, the results of the code review become a permanent part of the code’s history. It will then be easy to see how the recommendation was incorporated into the project.

Tested Robustness

A class can be considered tested when all the mainstream interfaces have been tested and all core scenarios have been validated through black box testing. This is a far more thorough testing pass than that which may have been done for the fragile stage of verification, but it still can be classified as ad hoc.

The “tested” phase of verification can be also termed ad hoc black box testing. There are two parts to this definition. First, “ad hoc” means without a formal or systematic plan. Note that ad hoc testing often finds the important bugs and results in quality software. When performing ad hoc testing, we are not claiming completeness. The second part of the definition is “black box” ...

Code is tested with all the mainstream interfaces and the core scenarios have been validated through black box testing

Black Box Testing

Black box testing is a validation and verification process where only two aspects are considered: the input into the system and the output received from the system. In this case, the internal structure and implementation of the item being tested are intentionally ignored. The goal is to make the testing implementation independent, so the same test will work regardless of design changes made by the programmer. Thus, all test cases consist of just two components: the input and the output.

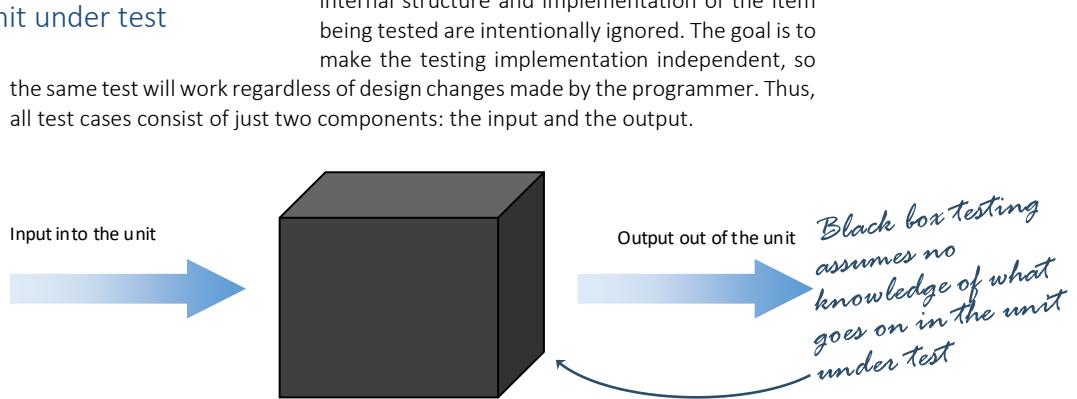


Figure 22.1:
Black box testing

Black box testing is sometimes described as “testing by contract.” Imagine a contract where certain output is expected from certain input. If the contract is honored, then both parties are satisfied. In this analogy, the implementer of the function or class is fulfilling the contract. It should be possible to completely rewrite the function or class and still pass the tests if the public interfaces remain unchanged and the contract is honored.

Black box testing is a valid form of testing because it looks at the system in the same way as the client of the class. It can also be performed by one without access to implementation details, by one unfamiliar with the technology employ to develop the system, or by one unbiased by knowledge of how the system is designed to work.

Tested classes can be relied upon to function as expected in common or mainstream uses. However, we cannot rule out the possibility that these same common and mainstream cases will move the class into an invalid state or cause it to return invalid data. The problem here is that we may miss important test cases, not probe all interfaces, and not explore all the error conditions. In other words, the probably of missing something important is rather high. With the lack of any formal quality assurance, we can certify the class is nothing better than unreliable. Therefore, we often need to move on to strong, resilient, or even proven robustness.

Drivers

To facilitate testing, it is often helpful to adapt the driver program technique used to test functions. For more details on how to use drivers, please see Chapter 15 Quality: Driver.

Drivers are helpful in the testing of classes because they allow the class to be exercised without involving the rest of the system. They allow many test cases to be run in a short amount of time, thereby increasing our confidence that we have found the important bugs. They also allow the programmer to investigate. When something is found that is “not quite right,” it is easy to dive deeper into the suspect output so the core problem can be identified. All of this can be done without recompiling the code!

Drivers are functions or classes designed to exercise other functions or classes. An example of a driver class exercising another class is the following. Here, input is gathered directly from the user and fed into the public interfaces. The state of the object is then probed with the public getters.

The code is a Java driver for a CheckingAccount class. It uses a Scanner to read input from System.in and a CheckingAccount object to perform operations. Handwritten annotations include: 'It is easy to run through many test cases' pointing to the main loop; 'Show the item before and after the edit' pointing to the edit operation; and 'Just use the smallest amount of code to test the class' pointing to the overall driver structure.

```
Java
public class Driver {
    public static void main(String[] args) {
        CheckingAccount account = new CheckingAccount();
        Scanner in = new Scanner(System.in);

        while (true) {
            // display the entire account at the beginning
            account.display();

            // prompt for one item to edit
            System.out.print("> ");
            int index = in.nextInt();
            account.item(index).display();
            account.item(index).edit();

            // display the results after the edit
            account.item(index).display();
            System.out.println();
        }
    }
}
```

Figure 22.2:
Driver function to establish
tested robustness

It is a common practice to leave drivers in the codebase even when the class under test has been validated. This is done to facilitate retesting of the class when the inevitable changes need to be made to the class under test. When drivers are left in the codebase, make sure to mark them as debug so they never find themselves in shipping code.

Strong Robustness

A class exhibits strong robustness when it has been rigorously verified through black box testing. This occurs when a thorough test plan has been developed and has been systematically executed. In most cases, this involves a test plan and a test harness.

A test plan is a document containing a collection of experiments which, taken as a whole, produces a certification of a degree of quality for a unit of software

Test Case for details on this) or a more formal test plan document (please see Chapter 42 Quality: for more details).

It is possible to certify that a given class is strongly robust by manually running all the tests in a test plan. It is more common to write debug code to automatically conduct these tests. When the class is simple and the tests are straightforward, these can be done in a single function. Often the complexity of the class makes this single-function approach prohibitive. In those cases, a suite of functions can be written with a unifying framework to handle setup and reporting duties.

When a test plan has been carefully developed and executed, all public interfaces will be thoroughly verified. This gives us a strong assurance of robustness that no user input will put an object in an invalid state and no unexpected output will come from the class.

Tests in the
code line
up with
experiments in
the test plan.

Python

```
# Test 1: Empty String
name = UserName("")
if name.last != "":
    print "Test 1 failed"

# Test 2: Only one name
name = UserName("Jack")
if name.last != "":
    print("Test 2a failed")
if name.first != "Jack":
    print("Test 2b failed")

# Test 3: Two names
name = UserName("First Last")
if name.last != "Last":
    print("Test 3a failed")
if name.first != "First":
    print("Test 3b failed")
```

A class can be designed to have strong robustness by permitting no direct access to member variables, ensuring complete initialization to all member and local variables, and thorough validation of all input parameters. These best practices do not ensure strong robustness, but are common attributes of classes that exhibit it.

Strong robustness is when a class has been rigorously and systematically verified through black box testing

Figure 22.3:
Automation to establish
strong robustness

Resilient Robustness

A class has been validated through thorough white box testing

Resiliency is a high degree of robustness characterized by thorough white box testing. For most applications, resiliency is the gold standard, there being little practical reason to invest in higher levels of quality. When a class is resilient, then there is no known way to entice the class into an invalid state or to return invalid data.

White Box Testing

White box testing is a validation and verification process utilizing detailed knowledge of how the system is implemented. Here, the testing process is highly coupled with the code being tested. If the developer chooses a different implementation, it is expected that the white box tests will fail and must be rewritten. This is particularly relevant to class verification because we need to not only verify how the methods work, but also how the class stores data. This is especially important when the data are stored in a nontrivial way.

White box testing utilizes detailed knowledge of how the system works

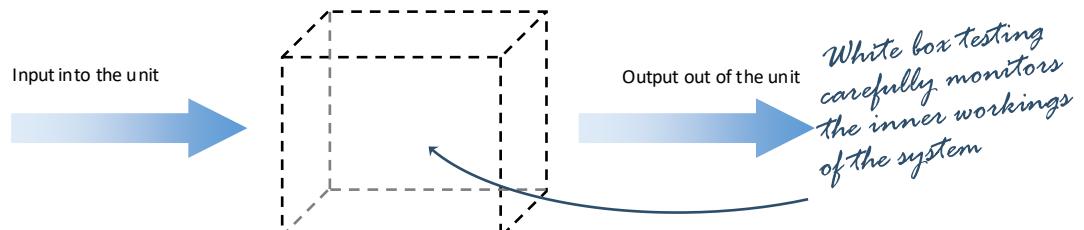


Figure 22.4:
White box testing

When working with classes, white box testing involves validating the private member variables contain data in the expected format. This usually comes in two parts: setting member variables before the test is run and validating the state of the member variables after the test is run.

The first part of a white box test usually involves setting the class under test into a given pre-condition state. While it is possible to do this through utilizing public setter methods, the risk exists that there may be an interaction between the two methods. In other words, if both the getter and the setter have a related bug, then the test may pass but a bug may remain. To minimize this risk, white box tests usually set the member variables themselves before the test is run. This way, an anomalous behavior in the method under test will be precisely measured.

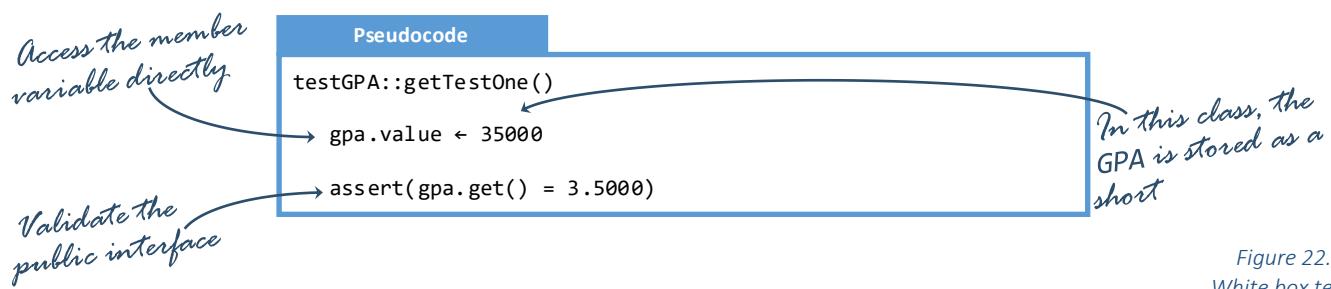


Figure 22.5:
White box test
setting the pre-condition

The second part of a white box test usually involves validating that the state of the member variables is what was expected after the test was run.

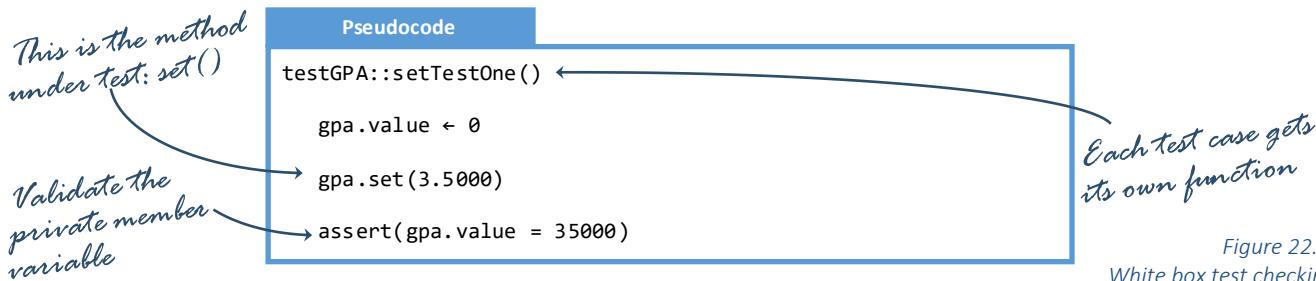


Figure 22.6:
White box test checking
the post-condition

Resiliency builds from strong robustness. A thorough test plan needs to be developed, and a suite of automation tests needs to be created to execute the plan. Resiliency adds another layer to this through the white box component. Here, the tester has expectations how the system will represent various constructs, and these expectations can be checked. This is particularly important to classes because fundamentally classes are designed to represent things. With white box testing, the tester can not only verify that the class behaves the way that was expected (black box testing), but also that the internal data representation is what was expected (the white box component).

In the following example, the set abstract data type stores a collection of integers. With access to Set's private member variables we can verify not only that the set contract is honored (the black box component), but also that the data structure is correctly storing the inserted elements in sorted order. This implementation detail has no effect on the functionality of the class because the client is never exposed to that detail. However, it does have important performance implications.

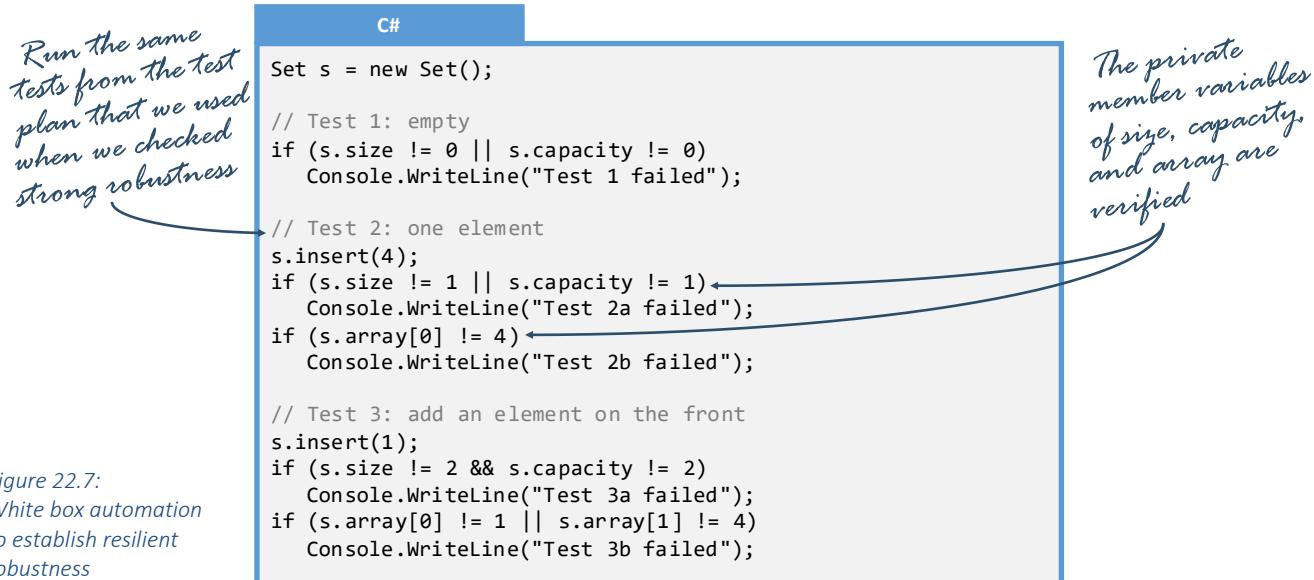


Figure 22.7:
White box automation
to establish resilient
robustness

When a class has reached resilient certification, then we can be assured that it is a dependable and reliable part of the system. This has become such a critical part of many software development methodologies that a framework for developing white box tests has been created. This is called unit testing and will be described in subsequent chapters (please see Chapter 25 Quality: Unit Test for more details).

Proven Robustness

Proven is a highest degree of robustness characterized by reducing a class to a mathematical model, applying formal correctness proofs on the public interfaces, and then validating that the implementation meets the mathematical model.

This is an extremely arduous and time-intensive process, beyond the capability of many software engineers. Why would anyone do such a thing? Well, there are several reasons.

Libraries often need to be proven

First, some components to a system are so heavily used that even subtle and highly improbable bugs can affect the overall stability of the application.

System libraries are a great example of this. Every function or class you use from the C++ Standard Template Library, the .NET framework, or the Java Class Libraries has been mathematically described and proven. There is so much code that depends on these and other core libraries that we simply cannot tolerate bugs. If you are every called upon to work on or develop a core library such as this, then serious consideration should be taken to make the robustness proven.

Second, some bugs are extremely difficult to fix. This includes embedded systems, networking protocols, and data synchronization components. Programmers are often willing to go through extreme lengths to avoid fixing bugs in these environments. In cases such as these, proven robustness can turn out to be a large time saver.

Difficult-to-debug code often benefits from proven reliability

Finally, there are a collection of mission-critical systems that simply cannot fail. These are common in the aerospace, security, and medical applications. In cases such as these, proven robustness can be a contractual or legal obligation. There are also times when your personal reputation may be on the line. A high-stakes change can have serious and immediate consequences if things go bad. This is very common when a product is near release time. Spending a couple hours mathematically proving the code change would not only be worth the time, but will also significantly reduce your stress level!

Mission-critical systems such as aerospace, security, and medical applications can have legal or contractual obligations to be proven

Proven robustness is when a class has been mathematically proven to be correct for all input

Designing for Robustness

It is far easier to design a class with robustness in mind than it is to try to “test-in” robustness. If a class will need to be strong, resilient, or proven, there are a few things that can be done to make this process much less painful.

Best Practice 22.1 Start the test plan early

It is generally a good idea to make your first draft of the test plan at the very beginning of the project. The more you think through the test cases, the more likely it will be that you will handle them in the first draft of your algorithm rather than at the end. There are two reasons why this is the case.

First, most bugs are caused because the programmer is ignorant about how the code will be used and all the corner cases it will be expected to handle. As the adage goes, “you are never as ignorant about your project as you are this very moment.” This is true, of course; the longer we work on a project, the more we learn about it. By identifying test cases early in the development cycle, we become more informed and thus less likely to introduce bugs.

Second, it is much easier to fix a bug shortly after you introduce it rather than at the end of the project. We thus want to minimize the gap between introducing a bug and fixing it. By having many test cases laid out early in the project, we can run through them as soon as we have functional code.

Best Practice 22.2 Produce well-defined output, but accept poorly formed input

The data flowing out of a class should always be in a well-defined state. It should be predictable and reliable. However, do not expect the input into a class to be the same. Your class should work reliably regardless of what it is given. This best practice should be expressed as: “have higher expectations for yourself than you do for others.” Perhaps this best practice also applies to life in general.

Best Practice 22.3 Maintain the test plan as you go

As you learn more about the project and how your code will be used, you naturally discover test cases. Rather than trying to remember them, add them to your test plan and codify them into the automation.

Best Practice 22.4 Design white box tests as you implement the code

The best time to write code that validates the data structures of your class is when you implement the data structures of your class. At that moment, all the nuances are in your head. Don’t let them fade away; codify them!

Best Practice 22.5 Use formal representations whenever possible

It is easy to use layman’s terms to describe how an algorithm or class works. It is also easy to gloss over important details. The more precise our language, the less wiggle room we leave for ambiguity.

Try to represent an algorithm in the most formal terms available to you. Not only does this help reveal bugs, but it also gets you one step closer to proven reliability.

Examples

Example 22.1: Fragile Robustness

This example will demonstrate how to identify fragile robustness.

Problem

Certify that the following class is at least fragile.

GPA
- gpa : Short
+ GPA() + get(): Float + set(input : Float)

Solution

To demonstrate that a class is fragile, we can do one of three things: perform no robustness analysis, execute the program in a small number of trivial inputs, or give the code a visual inspection. This third option is called face validity. We will pursue this third option by inspecting the following C++ implementation with a code review.

C++

```
// a class to store a GPA
class GPA
{
private:
    unsigned short gpa;
public:
    // default constructor sets the value to zero
    GPA(float input = 0.0) { set(input); }

    // retrieve a float gpa from the short
    float get() const
    {
        return float(gpa) / 10000.0;
    }

    // sets the gpa to a new value
    void set(float input)
    {
        input *= 10000.0;
        if (input >= 0.0 && input <= 40000.0)
            gpa = (short)input;
        else
            gpa = 0;
    }
};
```

Because everything looks as expected, we call the code reviewed.

Example 22.2: Tested Robustness

This example will demonstrate how to identify tested robustness.

Problem

Provide an assurance that the GPA class from Example 22.1 is “tested.”

Solution

The most convenient way to establish unreliability is to create an interactive driver program. Since there are two public methods, we will have two components to our driver program.

C++

```
// simple driver program
int main()
{
    cout.setf(ios::fixed | ios::showpoint);
    cout.precision(3);

    while (true)
    {
        float input;
        GPA gpa;

        cout << "> ";
        cin >> input;
        gpa.set(input);

        cout << " Input:" << input
            << " Output:" << gpa.get() << endl;
    }

    return 0;
}
```

Next, we will run through mainstream test cases to validate that the class works as expected.

Output

```
> 3.5
Input:3.500 Output:3.500
> 2.2
Input:1.234 Output:1.234
> -1
Input:-1.000 Output:0.000
> 5
Input:5.000 Output:0.000
> 0
Input:0.000 Output:0.000
> 4
Input:4.000 Output:4.000
```

Because everything runs as expected for these inputs, we can call this “tested.”

Example 22.3: Strong Robustness

Problem

Provide an assurance that the GPA class from Example 22.1 and 22.2 is “strong.”

Solution

We start with a test plan, recognizing that there are five fields of numbers: those below the valid range, the lower boundary condition, those in the valid range, the upper boundary condition, and those above the valid range. Of these, there are three ranges (lower invalid, valid, and upper invalid), each of which should have more than one probe point (near the lower limit, the middle, and near the upper limit). This gives us several test cases:

Test Case	Rationale
Large number	Upper bounds of upper invalid range
10.0	Middle of upper invalid range
4.0001	Lower bounds of upper invalid range
4.0000	Upper boundary
3.9999	Upper bounds of valid range
1.2345	Middle of valid range
0.0001	Lower bounds of valid range
0.0000	Lower boundary
-0.0001	Upper range of lower invalid range
-10.0	Middle of lower invalid range
Large negative number	Lower bounds of lower invalid range

Next, we will create a function to run through all these cases.

```
C++  
void verifyGpa()  
{  
    struct  
    {  
        float input;  
        float output;  
    } values[11] = {  
        { 3.4e38, 0.0000}, { 10.0, 0.0000}, { 4.0001, 0.0000},  
        { 4.0000, 4.0000}, { 3.9999, 3.9999}, { 1.2345, 1.2345},  
        { 0.0001, 0.0001}, { 0.0000, 0.0000}, {-0.0001, 0.0000},  
        {-10.000, 0.0000}, {-3.4e39, 0.0000}  
    };  
  
    cout << " input      output      expected\n";  
    for (int i = 0; i < 11; i++)  
    {  
        GPA gpa(values[i].input);  
        cout << setw(10) << values[i].input // provided input  
            << setw(10) << gpa.get() // output  
            << setw(10) << values[i].output // expected  
            << endl;  
    }  
}
```

If the output matches the expected values in all cases, the code exhibits strong robustness.

Example 22.3: Resilient Robustness

This example will demonstrate how to identify resilient robustness.

Problem

Provide an assurance that the GPA class from Example 22.1–22.3 is “resilient.”

Solution

A class is resilient only after it is first shown to be strong. Please see Example 22.3 for that.

Next, we will develop a white box comment to verify that resiliency. To do this, we will create a class (**GpaTest**) which has access to GPA’s private attributes. This class will have a method (**setTest()**) which exercises **GPA::set()**. It will run through the same test cases as Example 22.3 but will also monitor the value of the member variable **gpa**. If the GPA is ever different than expected, an assertion will be thrown.

```
C++  
void GpaTest::setTest()  
{  
    struct  
    {  
        float input;  
        unsigned short expect;  
        float output;  
    } values[11] = {  
        { 3.4e38, 0, 0.0000}, { -3.4e39, 0, 0.0000},  
        { 10.0, 0, 0.0000}, { -10.000, 0, 0.0000},  
        { 4.0001, 0, 0.0000}, { -0.0001, 0, 0.0000},  
        { 4.0000, 40000, 4.0000}, { 0.0000, 0, 0.0000},  
        { 3.9999, 39999, 3.9999}, { 0.0001, 1, 0.0001},  
        { 1.2345, 12345, 1.2345},  
    };  
  
    for (int i = 0; i < 11; i++)  
    {  
        // SETUP: create a GPA object  
        GPA gpa(0.0);  
  
        // EXERCISE: call the set() method  
        gpa.set(values[i].input);  
  
        // VERIFY: make sure the output is what was expected  
        assert(gpa.gpa == values[i].expect);  
        assert(gpa.get() == values[i].output);  
  
        // TEARDOWN: trivial, do nothing!  
    }  
}
```

Running through our test cases, we see that not only is the input and output what we expect, but the internal representation of our member variable is also as expected. With both white box and black box tests passing, this class is classified as resilient.

Example 22.4: Proven Robustness

Problem

Swift

```
func binary(a: [Int], value: Int, range: Range<Int>) -> Bool {
    let mid = (range.lowerBound + range.upperBound) / 2
    if range.lowerBound >= range.upperBound {
        return false
    } else if a[mid] > value {
        return binary(a, value: value,
                     range: range.lowerBound ..< mid - 1)
    } else if array[mid] < value {
        return binary(a, value: value,
                     range: mid + 1 ..< range.upperBound)
    } else {
        return true
    }
}
```

Solution

$$\text{binary}(\text{array}, \text{value}, \text{min}, \text{max}) = \begin{cases} \text{TRUE} & \text{value} = \text{array}[\text{min}] \\ & \text{value} < \text{array}[(\text{min} + \text{max})/2] \\ \text{binary}(\text{array}, \text{value}, \text{min}, \lfloor (\text{min} + \text{max})/2 \rfloor) & \text{value} > \text{array}[(\text{min} + \text{max})/2] \\ \text{binary}(\text{array}, \text{value}, \lfloor (\text{min} + \text{max})/2 \rfloor, \text{max}) & \text{min} > \text{max} \\ \text{FALSE} & \end{cases}$$

Assumption: The array is sorted. Thus, if $a < \text{array}[i]$, then $a < \text{array}[j]$ for all $i < j$.

Assertion: Let $P(\text{min}, \text{max})$ be the assertion that $\text{binary}(\text{array}, \text{value}, \text{min}, \text{max})$ works for all inputs. If we can prove that $P(\text{min}, \text{max})$ works for all numbers of min, max , then it works for all values.

Base Step: Let $n = \text{number of valid elements between } \text{min} \text{ and } \text{max}$. If the range contains no elements, then $n = 0$ and the function returns *false* which is correct.

Inductive Step. Let $\text{mid} = \lfloor (\text{min} + \text{max})/2 \rfloor$. There are three cases:

Case 1: $\text{value} = \text{array}[\text{mid}] = \text{value}$. Clearly the function works correctly

Case 2: $\text{value} < \text{array}[\text{mid}]$. Since array is sorted, value must be in an index less than mid or in the range $\text{min} \dots \text{mid} - 1$. This will be a recursive call with an n smaller than that of the current function call.

Case 3: $\text{value} > \text{array}[\text{mid}]$. Since array is sorted, value must be in an index greater than mid or in the range $\text{mid} + 1 \dots \text{max}$. This will be a recursive call with an n smaller than that of the current function call.

Because all cases in the inductive step work and the inductive step brings us closer to our base step, we can conclude that $P(\text{min}, \text{max})$ works for all values of min and max .

Thus, by math induction, we have proved that the algorithm is correct.

Exercises

Exercise 22.1: Define the Level of Robustness

From memory, recite the definition of each level of robustness:

Level	Definition
Proven	
Resilient	
Strong	
Unreliable	
Fragile	

Exercise 22.2: Identify the Level of Robustness

Identify the level of robustness from the following examples:

Validation	Level
Using an in-depth test plan, thorough black box testing was conducted on the class.	
A mathematical model of the class was created, and formal proofs were conducted.	
Some rudimentary quality checks were performed once implementation was complete.	
A combination of white box and black box testing was carefully and systematically conducted.	
We just completed a lot of ad hoc black box testing on the class using an interactive driver program.	

Exercise 22.3: Identify from Example

Consider a class designed to represent linear distance. This class can work with imperial units (inches, feet, miles, etc.) or metric units (meters, kilometers, etc.). What level of robustness is the following code meant to establish?

Pseudocode

```
testLength()
  WHILE true
    PROMPT for distance in meters
    GET meters
    distance.setMeters(meters)
    PUT meters, distance.getFeet()
```

Note that a thorough set of test cases accompany this code.

Exercise 22.4: Identify from Example

Consider a class designed to represent linear distance. This class uses an integer as millimeters for the underlying data representation. What level of robustness is the following code meant to establish?

Pseudocode

```
DistanceTest::getMetersTest()

  distance.mm ← 1000

  meters ← distance.getMeters()

  assert(meters = 1)
```

Note that this is one of several tests derived from a thorough set of test cases.

Problems

Problem 22.1: Fragile Position Class

Consider a **Position** class representing a position on a chessboard.

Position
- position : Char
+ Position()
+ getRow() : Integer
+ getColumn() : Integer
+ set(row : Integer, column : Integer)
+ display()

Here, the internal representation of the position is a single character. Note that a character has 256 possible values, but a chessboard has 64 possible values. There are therefore plenty of bits to represent this position. To accomplish the mapping between the row and column values and the internal representation, the following code is provided:

Pseudocode

```
getRow()
    RETURN position / 8

getCol()
    RETURN position % 8

set(row, column)
    position ← row * 8 + column
```

Implement this class in a programming language of your choice. When you are finished, establish that your implementation has fragile robustness.

Problem 22.2: Tested Position Class

From the **Position** class of Problem 22.1, establish that the class has tested robustness. If any bugs are found in this process, please fix them.

Hint: you may need to write a driver program to solve this problem.

Problem 22.3: Strong Position Class

From the **Position** class of Problem 22.1, establish that the class has strong robustness. If any bugs are found in this process, please fix them.

Hint: You may need to create a simple test document and write some simple automation to solve this problem.

Problem 22.4: Resilient Position Class

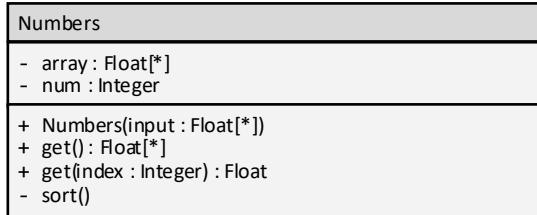
From the **Position** class of Problem 22.1, establish that the class has resilient robustness. If any bugs are found in this process, please fix them.

Hint: You may need move your automation function into the **Position** class.

Challenges

Challenge 22.1: Fragile Numbers Class

The following class diagram describes a class that contains an array of numbers.



These numbers are maintained in sorted order. The member function `sort()` will sort the array and is called by the constructor. This function will use the selection sort. The code for the insertion sort is the following:

Pseudocode

```
Numbers :: sort()
FOR iPivot ← num-1 ... 1
    iLargest ← 0
    FOR iCheck ← 1 ... iPivot-1
        IF array[iLargest] ≤ array[iCheck]
            iLargest ← iCheck
        IF array[iLargest] > array[iPivot]
            swap array[iLargest], array[iPivot]
```

Implement this class in a programming language of your choice. When you are finished, establish that your implementation has fragile robustness.

Challenge 22.2: Tested Numbers Class

From the **Numbers** class of Challenge 22.1, establish that the class has tested robustness. If any bugs are found in this process, please fix them.

Hint: you may need to write a driver program to solve this problem.

Challenge 22.3: Strong Numbers Class

From the **Numbers** class of Challenge 22.1, establish that the class has strong robustness. If any bugs are found in this process, please fix them.

Hint: You may need to create a simple test document and write some simple automation to solve this problem.

Challenge 22.4: Resilient Numbers Class

From the **Numbers** class of Challenge 22.1, establish that the class has resilient robustness. If any bugs are found in this process, please fix them.

Hint: You may need move your automation function into the **Numbers** class.

Challenge 22.5: Proven Numbers Class

From the **Numbers** class of Challenge 22.1 - Challenge 22.4., establish that the class has proven robustness. If any bugs are found in this process, please fix them.

Hint: Focus on the `sort()` method. Create a proof using mathematical induction.

Convenience is how easy it is for a client to use a class in an application.

In many ways, object-oriented programming requires a paradigm shift. Think not about applications, but rather about tools. Think not about the lone programmer, but rather about a member of a large team. Here, there are two players: the client and the provider. The client is the user of the class and the provider is the one who creates the class for the client. This client-provider relationship is the heart of object-oriented programming.

Convenience is the degree in which properties are presented to the client in a format that facilitates use in an application

The designer of a class must always keep the client's interests in mind. She needs to do everything possible to understand the client's needs, know how the class will be utilized in the application, and try to make it as easy as possible for the client to integrate the class into the application. There are two parts of this definition which are important: "properties presented to the client" and "used in an application." Convenience is about the public interfaces and how they fit into the application the class is meant to serve. In other words, one cannot evaluate the convenience of a class by simply looking at the public interfaces. Instead, it needs to be considered in the context of the application in which it is built.

There is a relation between the encapsulation notion of convenience and the modularization notion of coupling. They are similar in how they both describe the interconnection between programming entities. The difference is that coupling illustrates the complexity of a single interface whereas convenience is about how well the class fits into the application. For example, a complex or document interface may be seamlessly convenient if the client has such an object on hand.

Levels of Convenience

There are five levels of convenience. Each are expressed in terms of the effort required by the client to use the class to fulfill a design concern. In other words, all measurements of convenience are relative to the application domain. A class may exhibit seamless convenience in one application and convoluted in another.

Level	Description
Seamless	Perfectly aligned with the needs of the application.
Easy	No extra work is required to manipulate.
Straightforward	Any data manipulation can be readily performed.
Convoluted	Much needs to be done to use the class.
Prohibitive	More work is required than it is worth.

Prohibitive Convenience

Prohibitive convenience is when more work is required to use the class than benefit is derived from the class

A class can be categorized as prohibitive when more work is required to manipulate data traveling into and out of the class than the benefit the client receives from using the class. This can happen when a client tries to use an inappropriate tool for a given

job, reusing an existing class that does not really help with the job at hand. It can also happen when the provider has an incomplete or erroneous understanding of the client's needs. This could be the result of the client's needs changing, imprecise requirements, the provider failing to understand the requirements, or a simple communication breakdown.

For example, consider a class to write data to a file. This class handles the intricacies of various file systems as well as error handling. Unfortunately, it is designed to work only with integers. Our program requires us to save a checking account transaction to a file with this class. To do this, we need to serialize all the transaction data (convert to a series of integers) when we save a file, and deserialize the data (convert it back to useful text) when we read from a file.

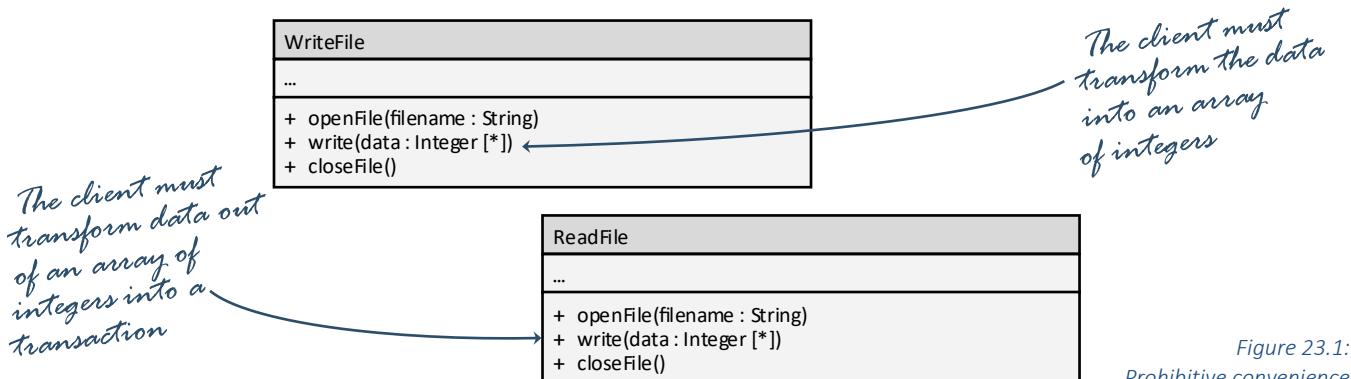


Figure 23.1:
Prohibitive convenience

When a class is prohibitive, it is usually best to start over and develop a more appropriate tool. While this may seem obvious, it is often not the chosen path. For political or social reasons, programmers are often compelled to use prohibitive tools. In cases like this, the best thing that can be done is to create a façade.

A façade is a function or a class whose job is to translate one interface into another. This serves to shield both the provider and the client from each other's interfaces and implementation details. Of course, the façade's interface should be as seamless as possible—that is the whole point of writing it! However, never forget that the very existence of a façade is a good indication that a convenience problem lurks elsewhere. Façades are merely coping strategies for dealing with poorly designed classes.

Façades are merely coping strategies for dealing with poorly designed classes

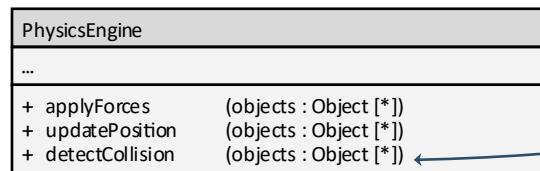
Convolved Convenience

A class can be categorized as convoluted when a great deal of work needs to be done to manipulate data going into and out of the class, often requiring a layer or adapter to be written to facilitate the process. This is typically because the client uses one paradigm and the provider uses another.

For example, consider a video game where the physics engine is written using spherical coordinates (r : distance, θ polar angle, and ϕ azimuthal angle) but the game is written in Cartesian coordinates (x , y , z). For the most part, this can be readily accomplished with a single function call. However, since there are many spherical coordinates to describe the Cartesian (0,0,0) origin, we could have two coordinates that were the same in the Cartesian system but are not in the spherical system. Having to deal with this complexity makes the interface convoluted.

Convolved convenience is when a great deal of work needs to be done to manipulate data so you can use a class

Figure 23.2:
Convolved convenience



Every object needs
to be in spherical
coordinates, but the
game is Cartesian

In another example, consider a recipe program that needs to send recipes to the printer. The program uses the red-green-blue (RGB) color space to describe color, whereas the printer class uses cyan-magenta-yellow-key (CMYK). As with the coordinate transformation issue, this color space transformation can be accomplished with a simple function call except that there is not a 1:1 mapping between RGB and CMYK. Thus, two colors which may have been equal before the transformation may not be after.

Everything is in
CMYK color,
even when there is
no CMYK color
corresponding to a
given RGB color

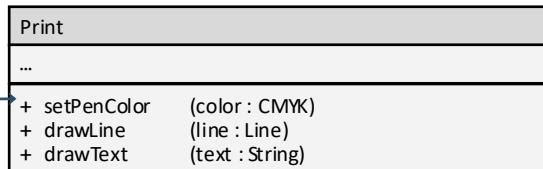


Figure 23.3:
Convolved convenience

As with prohibitive convenience, often the best way to handle convoluted classes (aside from rewriting them) is to create a façade so the conversion is more seamless to the client. This is the same thing what we do when we are forced to work with a prohibitive class.

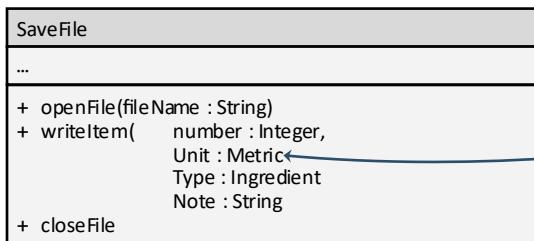
Straightforward Convenience

Straightforward convenience is when any data manipulation can be readily performed by all clients

A class can be categorized as straightforward if any data manipulation can be readily performed in all applications using the class. Here, all the necessary tools to perform conversion are present and available. Usually these tools are provided in a few functions.

It is not hard to find examples of straightforward convenience. Consider, for example, a recipe program presenting the user with imperial units for American audiences. Here, everything is in teaspoons, tablespoons, cups, pints, quarts, and gallons. The file format, however, is in liters. Every interface with the file will need to convert imperial units to metric or vice versa. This conversion is straightforward by calling a readily-available function, but still requires work on the part of the client.

Figure 23.4:
Straightforward
convenience



We need to convert to metric every time we write

Another example would be a video game using a third-party physics engine. This engine requires everything to be in physical units (meters, kilograms, and seconds) but the game works in virtual units (pixels, mass units, and frames). This conversion must be done with every interaction with the physics engine but is straightforward to conduct.

Must convert to PhysicalUnit every time we call this metric

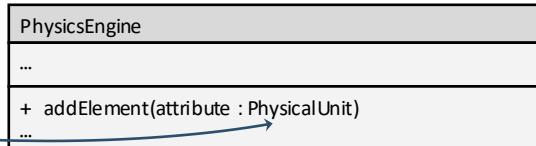


Figure 23.5:
Straightforward
convenience

Note that straightforward convenience is not bad. In fact, it can be unavoidable. Generic tools utilized by many applications typically provide generic interfaces. We often need to perform trivial transformations when using them. This is expected. The only time straightforward convenience is bad is when a class was written specifically for a purpose and still requires the client to perform data transformations.

Easy Convenience

A class can be categorized as easy if no extra work is required by the client to manipulate data going into or out of the class. The format of the data utilized by the application is the same as that used by the class' public interfaces. Methods can thus be called without any extra work.

The hallmark of easy convenience is the lack of any data conversion by the client. Consider a class designed to update the position of a spaceship in a video game. To apply inertia, the client needs to know the ship's current velocity. This is accessible from the class' own `getVelocity()` method. It is therefore very easy to provide the needed information: just ask the class to provide it itself!

The client does not convert the data, it just passes it along from the class method

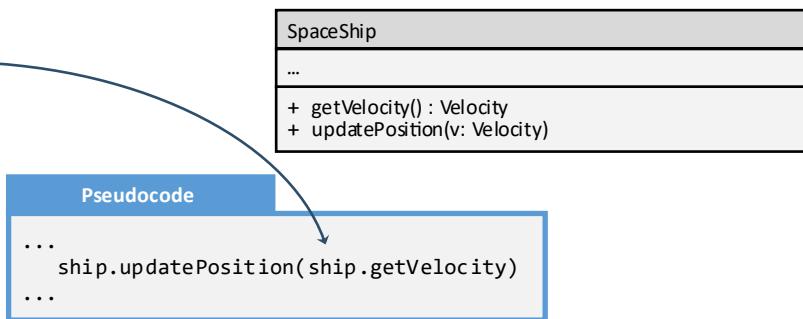


Figure 23.6:
Easy convenience

Note that though a class has easy convenience, there might still be convenience improvement opportunities. For example, consider the file interface provided by the standard template library of C++. The following example shows how to write the number 42 into a file.

```
C++
void writeAnswerToFile()
{
    ofstream fout("theAnswer.txt");
    if (fout.fail())
        return;
    fout << 42 << endl;
    fout.close();
}
```

Figure 23.7:
Simple interface but complex interaction between the methods

Because the interface asks for parameters which are easily provided, then the convenience level can be considered easy. However, the sequence of method calls and the required error checking requires work on the part of the client. Thus, the code cannot be called seamless.

Seamless Convenience

Seamless convenience is when the properties and attributes of the class are perfectly aligned with the needs of the application

not uncommon for a class to remain unchanged yet the degree of convenience to change with time.

For a provider to achieve seamlessness, she needs to offload as much work as possible from the application into the class. This allows the client to work at a higher level of abstraction, freeing him or her from having to manage the data of the class.

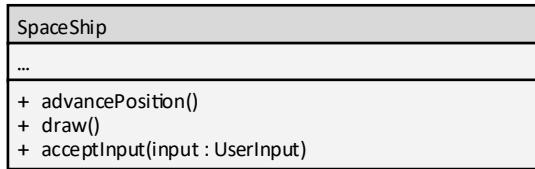


Figure 23.8:
Seamless convenience

Back to our checking account example, we would like the class to be as self-sufficient as possible from the rest of the application. To do this, such common actions as saving data to a file, drawing the account on the screen, and handing edit events should just happen.

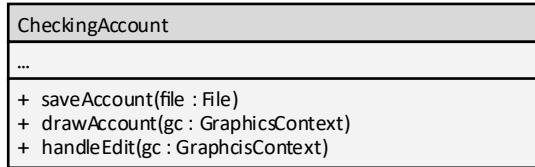


Figure 23.9:
Seamless convenience

Designing for Convenience

There is no hard-and-fast methodology that one can follow to achieve seamless convenience with every class. There are a few considerations, however, which make the job much easier.

Best Practice 23.1 Ask the client many detailed questions

There is a technique called “precision questioning” where the interviewer starts with very general questions and works to more detailed ones. With each interviewee response, the interviewer zeroes in on details and nuances and launches into another round of clarifying questions. This process continues until a point is reached where the interviewee’s knowledge is exhausted. Utilizing a technique such as this is helpful for a provider to gain a deep understanding of the client’s needs.

Best Practice 23.2 Monitor how the class is used

When the client starts to utilize the class in the application, the provider should carefully study how it is used. If there are any signs of data transformations, then the class may be straightforward or worse!

When the client starts to use the class in a different way than was initially discussed, don’t be surprised. It is common for the client’s needs to change as the application is developed. In these cases, another conversation with the client might be in order.

Best Practice 23.3 Try to be overly helpful

Imagine two people preparing a meal together in the kitchen: the cook and the helper. It is useful if the helper is responsive to the cook’s requests for help. It is much better if the helper is proactive about finding opportunities to help. A proactive helper is constantly on the lookout for tasks that the cook can offload, thereby making the cook’s job easier. This analogy holds with the client/provider relationship. The provider should constantly be on the lookout for tasks the client can do but would be more appropriate if handled in the class. The more complexity can be removed from the application into the class, the better!

Examples

Example 23.1: Prohibitive Convenience

This example will demonstrate how to identify prohibitive convenience.

Problem

Consider the following application context:

A video game needs to keep track of how long it takes for a player to clear a given level. An updated timer is presented on the screen as the player works through the level, and a scoreboard is maintained in a “top scores” file.

To help the client with this, a provider created the following class:

Date-Time
...
+ get() : String
+ set(input : String)

Solution

Two things need to be considered with every convenience measure: the public interfaces and the needs of the client. The public interfaces here are a simple getter and setter working with strings. In other words, the client provides a textual date string like “Sunday the 7th of December 1941 at 8:00 am.”

The needs of the client are quite different. Absolute date or time is not needed, instead elapsed time. Furthermore, the client does not need to set a given time. Instead, the client needs to add a fixed amount to the time that has previously elapsed. For the client to use this class, she will need to do many nontrivial things:

- Manually turn absolute time into time elapsed by maintaining some sort of epoch (starting time).
- Provide the logic to increment the current time by a fixed amount.
- Parse the date-time string to determine the number of minutes and seconds. Also generate a date-time string to match a given time.
- Write the display code so it works for the user interface.

Taking all these things into account, it would be easier to start over than use this class. The class is thus prohibitive in the context of this application.

Example 23.2: Convoluted Convenience

This example will demonstrate how to identify convoluted convenience.

Problem

Consider the following application context, the same as Example 23.1:

A video game needs to keep track of how long it takes for a player to clear a given level. An updated timer is presented on the screen as the player works through the level, and a scoreboard is maintained in a “top scores” file.

To help the client with this, a provider created the following class:

Date-Time
...
+ get(): String
+ getMinutes(): Integer
+ getSeconds(): Integer
...
+ setTime(String)
+ setMinutes(Minute: Integer)
+ setSeconds(Seconds: Integer)

Solution

The public interfaces ask for and provide time units as integers. If a provided value is outside the valid range (such as 61 seconds), then an error is thrown and the time is not updated.

The client needs are unchanged from the previous example: to increment the number of seconds it took to pass a level. To accomplish this, the following work needs to be done.

- Compute the number of seconds that have transpired since the level was started. This is done by getting the hours, minutes, and seconds from the default constructor and the same from when the level started. The application will have to compute the number of seconds that has transpired.
- Write the display code so it works for the user interface.

Taking all these things into account, the class does provide some benefit to the application. However, the application needs to do most of the work. As a result, the class is convoluted in the context of this application.

Example 23.3: Straightforward Convenience

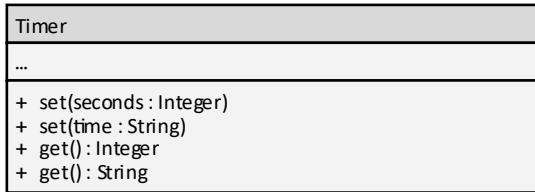
This example will demonstrate how to identify straightforward convenience.

Problem

Consider the following application context, the same as Example 23.1:

A video game needs to keep track of how long it takes for a player to clear a given level. An updated timer is presented on the screen as the player works through the level, and a scoreboard is maintained in a “top scores” file.

To help the client with this, a provider created the following class:



Solution

The first thing to notice is that this is no longer a time/date class, but a timer class. This is much better aligned with the needs of the client than the previous design. There is a set and get method working with seconds, aligning with the need of the game to track how long it takes to play a level. There is also a set and get method working with text. This is useful for reading and writing the scores file. It also facilitates generating a text time value for display on the screen.

It is straightforward for the client to use this class. All the public interfaces are expressed in terms that are easy for the client to consume and to produce. There remains quite a bit that the client needs to do to integrate this into the application:

- The client needs to interface with the system clock. This includes remembering when the level began as well as finding the current time to see how many seconds have transpired. It would be better if the class handled this detail but is easy to use the class' interfaces once the information is obtained.
- Write the display code so it works for the user interface.
- Write the file code so the time string can be written to a file and read from a file.

Taking all these things into account, the class is very straightforward to use: data manipulation can be readily performed by the client.

Example 23.4: Easy Convenience

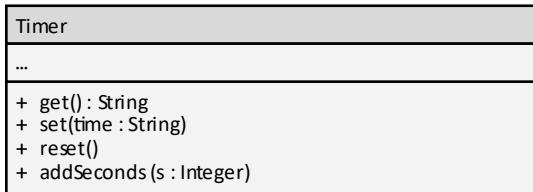
This example will demonstrate how to identify easy convenience.

Problem

Consider the following application context, the same as Example 23.1:

A video game needs to keep track of how long it takes for a player to clear a given level. An updated timer is presented on the screen as the player works through the level, and a scoreboard is maintained in a “top scores” file.

To help the client with this, a provider created the following class:



Solution

There are four public interfaces, none of which works with a traditional time-stamp. The `get()` and `set()` work with strings. None of the interfaces use absolute time; they all work with elapsed time instead.

When performing a timing operation on a game, the time of day is less interesting than the time since the game/level began. This interface works with elapsed time, being much more convenient than working with absolute time. There are still a few things which the client is required to do to integrate this into the class:

- The client needs to know how many seconds have transpired since seconds were last added to the timer. This may be through an interface with the system clock or it may have to do with frame counting.
- Write the display code so it works for the user interface.
- Write the file code so the time string can be written to a file and read from a file.

Taking all these things into account, the class is very easy to use: no extra work is required by the client to manipulate data going into or out of the class.

Example 23.5: Seamless Convenience

This example will demonstrate how to identify seamless convenience.

Problem

Consider the following application context, the same as Example 23.1:

A video game needs to keep track of how long it takes for a player to clear a given level. An updated timer is presented on the screen as the player works through the level, and a scoreboard is maintained in a “top scores” file.

To help the client with this, a provider created the following class:

Timer
...
+ startLevel()
+ stopLevel()
+ read(file : File)
+ write(file : File)
+ display(gc : GraphicContext)

Solution

Two of the public interfaces (`startLevel()` and `stopLevel()`) correspond exactly with events that occur in the game. Better yet, no parameters are required! The class integrates with the system clock to be able to perform its own timing. This offloads a great deal of complexity and work from the client.

Another two of the interfaces (`read()` and `write()`) handle the scores file. The client only needs to provide a file handle and the class does the rest of the work. Again, this removes a lot of complexity from the client’s code.

The final interface handles the display work by drawing the timer on the screen. If every game entity knew how to draw itself, imagine how much easier it would be for the client to add new elements to the game!

Since virtually all the housekeeping surrounding the class has been integrated into the class, and since all the public interfaces are perfectly aligned with the needs of the client, this is seamless convenience.

Exercises

Exercise 23.1: Define the Level of Convenience

From memory, recite the definition of each level of convenience:

Level	Definition
Seamless	
Easy	
Straightforward	
Convoluted	
Prohibitive	

Exercise 23.2: Identify the Convenience Level

Identify the level of convenience from the following examples:

Validation	Level
It is more work than it is worth; it is easier to start from scratch than to use this class	
Any data transformation that the client will need to make is very easy to accomplish	
No extra work is needed to transform input into a format that the class requires	
The class author has anticipated the needs of the client perfectly!	
Worth the effort, but so complicated that I will need to write a façade to make this work	

Exercise 23.3: A Class for an Unknown Purpose

Is it possible to determine the convenience level of a class without considering the application using the class? Explain why or why not.

Exercise 23.4: Façade

Which level(s) of convenience is/are characterized by the need (or benefit) of creating a façade? Justify your answer.

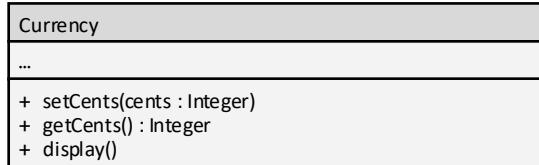
Problems

Problem 23.1: Categorize Currency

Consider the following scenario:

A financial package uses a **Currency** class to manage money. This class cannot have rounding errors so, for fidelity reasons, it uses an integer as the internal representation.

From this scenario, the following class diagram is developed:



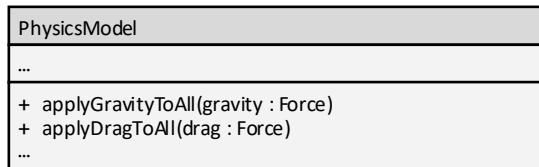
Classify the level of convenience for this class.

Problem 23.2: Categorize Physics

Consider the following scenario:

A video game uses one class to manage all applications of physics. Everything in the game uses a **Force** object to monitor force: force due to gravity, force due to thrusting, force due to drag, and force due to collisions.

From this scenario, the following class diagram is developed:



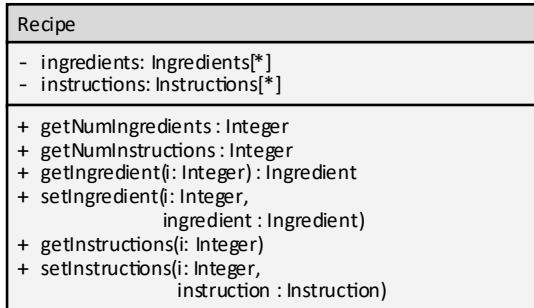
Classify the level of convenience for this class.

Problem 23.3: Categorize Recipe

Consider the following scenario:

A recipe program has a scaling option where the user can specify the number of servings needed for a given meal.

From this scenario, the following class diagram is developed:



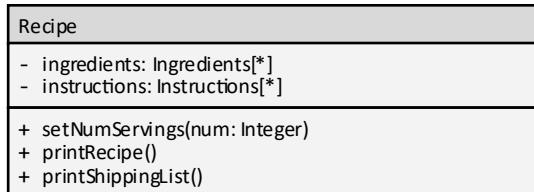
Classify the level of convenience for this class.

Problem 23.4: Categorize Recipe

Consider the following scenario:

A recipe program has a scaling option where the user can specify the number of servings needed for a given meal.

From this scenario, the following class diagram is developed:



Classify the level of convenience for this class.

Challenges

Challenge 23.1: Budget

Create a class diagram of easy or seamless convenience to match the following description:

A personal finance system needs to have a budget feature. This feature will allow the user to identify budget fields, set spending targets, and link budget fields to categories of spending set elsewhere in the product. The feature will also display two types of reports: the current budget settings and a simple table showing the user how this week's spending met the targets.

Identify the level of convenience of your **Budget** class and provide a justification for your assessment.

Challenge 23.2: Homework List

Create a class diagram of easy or seamless convenience to match the following description:

A list management mobile application has a homework feature allowing users to enter upcoming homework items. Each item has a due date, subject, and a hyperlink to the full item description. The feature will display a sorted list of upcoming assignments and provide the ability for the user to check items off the list as they are completed.

Identify the level of convenience of your **Homework** class and provide a justification for your assessment.

Challenge 23.3: Spaceship Fuel

Create a class diagram of easy or seamless convenience to match the following description:

A 3D video game has a ship which flies through a series of obstacles. One component is a fuel class. This class keeps track of how much fuel is remaining and displays a warning message when the fuel level gets too low. Fuel is added when the ship flies through a fuel game element, and gets reduced by one unit every second of gameplay.

Identify the level of convenience of your **Fuel** class and provide a justification for your assessment.

Abstraction

Chapter 24

Abstraction is the process of shielding users from unnecessary implementation details. It is a metric of class quality pertaining to how much the client needs to know about implementation details to use a class effectively.

Managing complexity is one of the most difficult challenges of developing software. While you may be able to internalize all the code of the one- or two-thousand-line program you completed last week, it is simply impossible to internalize a ten-million-line program. Even if you were able to accomplish this amazing feat, more code would be added every day than a single programmer can possibly learn. On the surface, this may appear to be a hopeless cause. Fortunately, it is not.

Programmers have many tools at their disposal to manage software complexity. One of the most powerful is abstraction. In a broad sense, abstraction is the process of shielding users from unnecessary implementation details. A driver does not need to know about the inner workings of the engine to drive a car. A cook does not need to know how a microwave works in order to thaw out frozen chicken. In the context of software, abstraction is a metric of class quality pertaining to shielding clients from complexity. Abstraction is the amount the client needs to know about the implementation details of the class in order to use it effectively.

Abstraction is the amount the client needs to know about the implementation details of the class in order to use it

A programmer can manage overall system complexity through careful application of the principle of abstraction. Consider an entire system developed using object-oriented programming where most of the code resides inside classes. If each of these classes utilizes complete abstraction, then each class is required to know very little about the other classes of the system. In this way, the overall system complexity is kept to a minimum.

Levels of Abstraction

There are four levels of abstraction. Each are expressed in terms of the amount of implementation details that are revealed to the client and how much the client must take these details into account when using the class.

Level	Description
Complete	No implementation details are revealed to the client.
Opaque	Unimportant implementation details are revealed to the client.
Porous	Knowledge of implementation are very helpful to the client.
Critical	The client must deeply understand the implementation details.

Complete Abstraction

A class can be classified as exhibiting complete abstraction when there are no clues to the client how the class is implemented. This makes it possible for the provider to completely change the implementation details of a class without any change to client code.

This is the gold standard of abstraction, enabling the class author (provider) to radically change implementation without fear of side effects it the rest of the application. There are many benefits to complete abstraction:

- **Reduced development time.** The amount of time it takes a client to utilize a class is greatly reduced when there is little the client needs to know. This impacts not only training time, but also how many things the client needs to keep in mind when using the class.
- **Greater program comprehensibility.** If each class in a program has complete abstraction, then one class will need to know very little about another class in order to work properly. This means that the amount of knowledge one needs to possess to change a given line of code is, for the most part, limited to the class in which the programmer is working.
- **More isolation of changes.** If two classes honor exactly the same interfaces in exactly the same way, then they are interchangeable. This means that any change within a class is accessible as long as it honors the class' interface.
- **More stable code.** A codebase has less bugs when the tester can isolate a behavior, the developer is able to isolate a code change, and the other members of the team are able to understand the code.

It is impossible to claim with certainty that a class exhibits complete abstraction. After all, there may always be an unforeseen clue revealing implementation details. The best we do is state that “there is no known clue.” Consider the stack class:

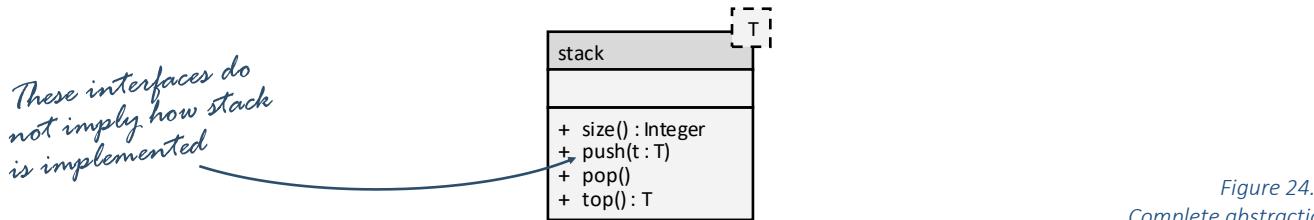


Figure 24.1:
Complete abstraction

This class can be equally implemented in as an array, a deque, or a linked list. As long as the stack contract is honored, it is up to the provider how the code works.

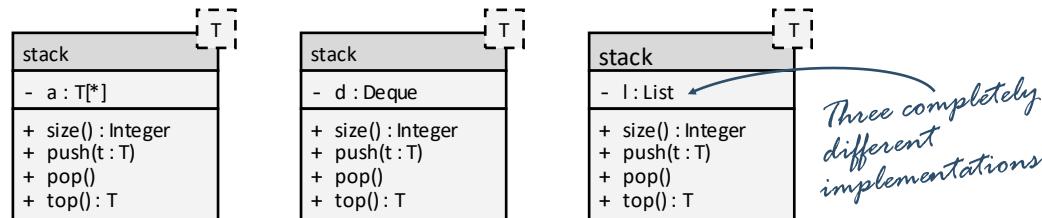


Figure 24.2:
Stack's three different implementations

Opaque Abstraction



Figure 24.3:
An abstraction with
a back door

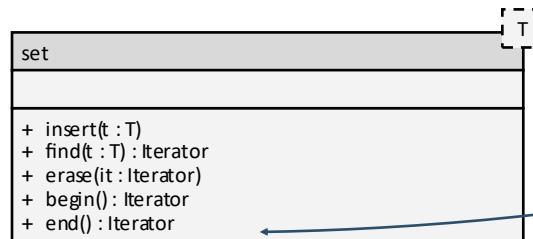
A class can be categorized as having opaque abstraction when there exists a very small number of non-mainstream cases where implementation details of the class can be revealed to the client. Thus, the class designer has succeeded in shielding the client from most of the complexity contained within, but an expert or heavy user might benefit from knowing some of the implementation details.

Opaque abstraction is when a small number of nonmainstream cases reveal implementation details to the client

Perhaps this is best explained by analogy. Consider a class as a castle and the attributes as the treasures contained therein. Access to the castle's assets occur through the gates which are the public operations. If the castle architect did a good job, then the only way to access the assets are through these gates. The problem occurs when there is a gap in the wall or a tunnel under a gate. While most of the townsfolk will still access the castle through the gates, some will not. The class designer needs to look a class in much the same way as the architect looks at the castle. Are there gaps or tunnels? Are there any ways to deduce the inner workings of the class which may cause troubles later?

The great danger of opaque abstraction is that a client may come to depend on an implementation quirk. This may not seem like a big deal; who cares if they rely on some esoteric detail? The problem arises when the class needs to be changed. If a performance enhancement, bug fix, feature addition, or a reimplementation is carried out, then either the quirk will need to be replicated or the client code will need to be changed. A diligent provider will introduce checks in the class to warn against such dependencies or will periodically monitor how the various clients are using the class. It would be better still to close those loopholes and make the abstraction complete.

There are many possible causes of opaque abstraction. The most common are boundary/limit conditions, performance quirks, and error conditions. In other words, things work predictably in most cases. It is the corner cases that cause trouble.



The iterators will reveal the order in which the class keeps the set items

Figure 24.4:
Class exhibiting
opaque abstraction

In the above example, we have a set class which contains a collection of items. The main interfaces are `insert()` which adds an element to the set, `erase()` which removes an element from the set, and `find()` which determines whether an element exists in the set. These all behave as one would expect, regardless of how the provider chose to implement the set. The problem comes from the `begin()` and `end()` methods which return an iterator. With these, it is possible to iterate through each element in the set. These methods will reveal an implementation detail: whether the provider chose to implement the set as sorted or not. If a client came to depend on this implementation detail, then we could have trouble when the provider modifies the class. This reveals opaque abstraction.

Porous Abstraction

Porous abstraction is when many implementation details are obvious to the client

A class can be classified as having porous abstraction when implementation details are obvious to the users of a class, and knowledge of these details makes it possible to more effectively use the class.

You know your class is porous when you feel compelled to educate clients. This could be when you provide documentation of the inner workings, when you write comments for your member variables or private methods which are directed towards the client, or when you find yourself fielding many client questions when the client is using your class. A sure-fire indication that a class is porous is when there is public access to member variables or when the getters and setters provide direct access to the member variables.

Porous abstraction is usually the result of a programmer casually throwing variables into a class without considering how the class is used by the application.

Many programming languages have array implementations which exhibit porous abstraction. They rely on the programmer to ensure that the indices are within the valid range. If the programmer fails to do this, then unexpected behavior will result. Consider a program manipulating a two-dimensional array. We happen to know that the compiler treats this like a single dimensional array. We can compute the two-dimensional index with the two-dimensional row and column variables with the following: `index = row * numCol + col`. The array implementation is therefore porous.

Pascal

```
program porous(input, output, stderr);
{$rangeChecks on}
var
  index: integer;
  ticTacToe: array[0..2, 0..2] of integer;
begin
  for index := 0 to 7 do
  begin
    ticTacToe[0, index] := index;
  end;
end;
```

*Though this is a
two-dimensional
array, we are
treating it like
a single array*

Figure 24.5:
Class exhibiting
porous abstraction

Critical Abstraction

A class can be classified as having critical abstraction when any client must possess a detailed understanding of implementation details of the class in order to effectively use it. For example, consider pointers as used in C or C++. Before a new programmer has any hope of using them correctly, a great deal of training is required. Usually pictures are drawn, examples are demonstrated, and problems are worked through. Even after hours of training, new programmers usually don't "get" pointers until they have been used in code for months or even years. Pointers are an example of critical abstraction because the programmer needs to deeply understand many implementation details.

Critical abstraction is when the client must possess a detailed understanding of the implementation details to use it effectively

Money is removed from Sue's account because the pointer refers to her account variable

```
C
{
    float accountSam = 2751.13;
    float accountSue = 3004.97;

    float * pointer = (accountSam > accountSue) ?
        &accountSam : &accountSue;

    *pointer -= priceDinner;
    *pointer -= priceMovie;
}
```

Programmers need to be cautious using pointers, fully aware of how they work and what they represent. Another example, also with pointers, deals with memory allocation. With normal variables (and allocated variables in languages such as Java and Python), the language handles many of the implementation details such as how they are created and destroyed. With lower-level languages such as C++, the programmer handles those details.

Figure 24.6:
Class exhibiting critical abstraction

Figure 24.7:
Critical abstraction

```
C++
{
    char * name = new char[32];      // allocate space for name
    cin.getline(name, 32);

    cout << "Hello, " << name << ", how are you?\n";
}                                     // name is never deleted!
```

The programmer is required to free all memory that was allocated

We should generally avoid critical abstraction. Modern programming languages such as Python, Swift, and Kotlin do not allow the programmer to handle raw pointers, thereby freeing them from having to manage details such as these. This moves the level of abstraction to porous or even opaque.

Designing for Abstraction

There are no hard-and-fast rules for guaranteeing complete abstraction in a class design. Instead, there are a collection of tips which make better design more likely.

Best Practice 24.1 Design public interfaces before considering implementation details

This is related to the property of convenience where public interfaces are designed to be as useful as possible to the client. This should be the first step. Next, design the private member variables with an eye towards fidelity. If we do it the other way around, then a key implementation detail (how we store the data) is likely to be revealed to the client.

Best Practice 24.2 Be mindful of edge conditions and error states

It is often our corner cases that betray us. Usually these are easier to handle early in the design process and more difficult late into implementation. In other words, an inexperienced class designer can paint himself or herself into a corner with error states and edge conditions if they are not considered early in the development process.

Best Practice 24.3 Be wary of performance traps

It is easy to get fixated on efficiency and performance. This is not a bad thing; performance often has a huge impact on the practicality and usability of our code. Over-fixation, on the other hand, can be a problem. Clients often say, “I can make this so much more efficient if you just give me a hook into your code.” Be wary of such requests. While it may yield performance gains, it will almost certainly result in porous abstraction.

Engineering is a discipline of trade-offs. There are few hard-and-fast rules; instead there are a myriad of considerations. Most engineers, when properly aware of all the factors influencing a decision, will make the appropriate selection. We get into trouble when we make decisions in a state of ignorance, not considering all the factors that come to play.

Performance gains are tangible and easily measured. The cost of poor abstraction is less obvious. It influences the overall complexity of the system, maintainability of the codebase, and stability of the final product. Each of these benefits is hard to measure directly, but they are important nonetheless. The astute programmer will take these into consideration when the temptation arises to decrease abstraction.

Examples

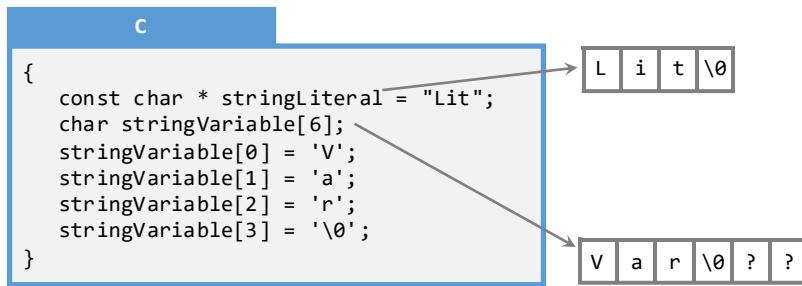
Example 24.1: Critical Abstraction

This example will demonstrate how to recognize critical abstraction.

Problem

Identify the level of abstraction of the following programming construct:

A C string is the default string implementation for the C programming language. A C string is defined as an array of characters terminated with a special null character '\0'. The programmer can specify a string literal with double quotes: "string literal." The programmer can specify a string variable by declaring an array of characters and manually inserting the null character.



Solution

The string literal is a constant pointer to a character. There are many implementation details contained in that single variable declaration, each of which the user of the variable must master before it can be used effectively.

The string variable is an array of characters. Notice that the programmer needs to manually insert the null character, or the C string will not be well-formed. The programmer also must make room for the null character in the array declaration, not skip any indices when initializing the string, recognize that the first index is 0, and not exceed the bounds of the array. If any of these details are not realized by the programmer, then the code will malfunction.

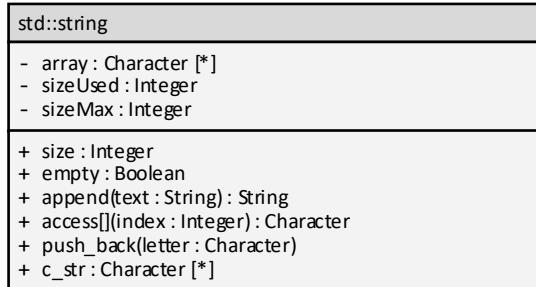
The fact that the programmer must be aware of so many implementation details surrounding the C string construct makes this abstraction critical.

Example 24.2: Porous Abstraction

This example will demonstrate how to recognize porous abstraction.

Problem

Identify the level of abstraction of the C++ `std::string` class. An abbreviated class diagram of the string class is the following:



Solution

For the most part, the public interfaces do exactly what the client expects: they manipulate strings. The client does not need to concern himself or herself with how the string is implemented. In fact, many never do.

The problem comes from the last public method: `string::c_str()` which gives the client direct access to `array`. This constitutes a huge breach in the castle wall. With the resulting pointer, the client can manipulate the underlying array so the `sizeUsed` member variable is no longer consistent with the number of characters in the string. This pointer can also allow the client to circumvent all the public methods. The client can even hold onto that pointer and then, when the string class reallocates the associated buffer, dereferencing the pointer can cause the program to crash. The existence of this single method gives the class a porous level of abstraction.

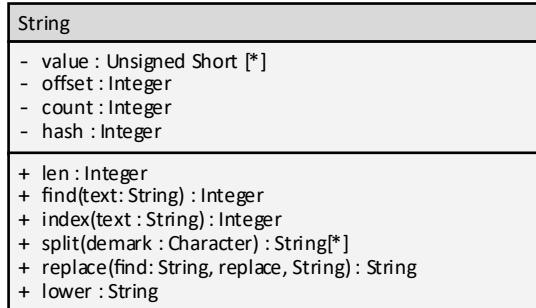
As an aside, the developers of this class certainly took this into account when the class was designed. This method was probably added for performance reasons so some clients could have more direct access to the buffer. It may have been added so the new string class could work with existing C string functions. Whatever the reason, there was some perceived benefit offsetting the abstraction problems it introduced.

Example 24.3: Opaque Abstraction

This example will demonstrate how to recognize opaque abstraction.

Problem

Identify the level of abstraction Java's `String` class. An abbreviated class diagram of the string class is the following:



Solution

On the surface, there appears to be nothing about the public interfaces which reveal the inner workings of the `String` class. There is, however, one minor chink in the armor. Java Version 6 attempts to optimize on the case when many substrings are drawn from a single string. In these cases, the return value of `substring()` returns a new `String` sharing the value with the original. The only difference is the `offset` and `count` variables are different. Normally this is great: one less buffer to copy! However, when we then assign the result back to the original string, now we have an unnecessarily large buffer.

```
Java
{
    // create a huge string containing all the data from a file
    String s = readLargeFile();

    // we only care about a 16 character sequence at the front
    s = s.substring(8, 24);

    // the string still has a huge capacity!
}
```

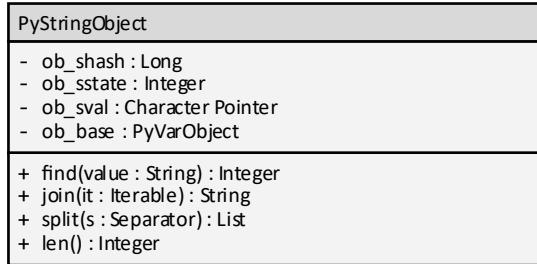
Only in subtle cases such as these do implementation details leak out. Because mainstream scenarios are strongly abstract and only unconventional scenarios reveal implementation details, this would be considered opaque.

Example 24.4 Complete Abstraction

This example will demonstrate how to recognize complete abstraction.

Problem

Identify the level of abstraction Python's `str` class. An abbreviated class diagram of the string class is the following:



Solution

As mentioned before, there is no reliable way to identify complete abstraction. The best we can do is to probe the interface from as many angles as we can. The deeper we go, the more confident we are that the class' abstraction level is complete.

In the case of the Python string implementation, we are never given access to the underlying implementation. We cannot tell whether the underlying data structure is an array of characters, a linked list, or a hash. We cannot tell whether text is stored as ASCII values, Unicode, or UTF-8. Because these implementation details are completely shielded from the client through the provided public interfaces, a case can be made that this class is complete abstraction.

To further strengthen the case that Python's `str` class has complete abstraction, the internal implementation was changed from Version 2 to Version 3. In Version 2, `str` was a one-byte string. In Version 3, it became a Unicode two-byte string. This change occurred without necessitating widespread overhaul of Python programs.

Exercises

Exercise 24.1: Define the Level of Abstraction

From memory, recite the definition of each level of abstraction:

Level	Definition
Complete	
Opaque	
Porous	
Critical	

Exercises 24.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Critical abstraction is better than complete abstraction	
To improve abstraction, define private attributes before public operations	
It is impossible to prove that an abstraction is complete	
Opaque abstraction is easier for the client to use than porous abstraction	

Exercise 24.3: Iterator Scenario

From the given scenario, identify the corresponding level of abstraction.

An iterator allows the client to traverse a collection of items. It works the same whether the underlying data store is a linked list, a tree, or an array. In fact, the author of the class can change the underlying data structure and there is no way that the client can know.

Exercise 24.4: Linked List Scenario

From the given scenario, identify the corresponding level of abstraction.

A linked list is a collection of structures where each node has two components: a data element and a pointer to the next node in the list. When you add a new node to the linked list, you must allocate a new node and carefully hook up the pointers so the chain remains unbroken. If you remove a node, then you must get the previous node in the linked list to point to the node after the one being removed. If you fail to do this exactly right, then the linked list is broken.

Exercise 24.5: ASCII Scenario

From the given scenario, identify the corresponding level of abstraction.

A single character of text is stored in one-byte data type called a character. Before 1963, there were over sixty standards describing how to map text to numbers. Today, ASCII is the standard encoding for the first seven bits of one-byte text. ASCII maps the Latin alphabet (a-z) in both uppercase and lowercase, the digits (0-9), a collection of symbols (&@...), spaces (space, newline, and tab), and other special characters. When working with text, it is important to know which characters were in the ASCII table as well as their relative order. For a long time, most developers would have the ASCII table on display in their office because they needed to reference it so often.

Problems

Problem 24.1: Year Class

Classify the level of abstraction of the following class which stores a year as an integer.

Year
- y: Integer
+ display()
+ add(num : Integer)

Pseudocode

```
Year::: add(num)
      y += num

Year :: display()
      IF year > 0
          PUT year AD
      ELSE
          PUT -year BC
```

Hint: What happens when you subtract 2030 from today's year?

Problem 24.2: Date-Time Class

The Unix operating system represents time using the POSIX format. Here, time starts on the 1st of January 1970. Time is stored as a 32-bit integer, representing the number of seconds since that date. Classify the level of abstraction of the following class implementing POSIX date/time:

DateTime
- seconds : Integer
+ getYear : Integer
+ getMonth : Integer
+ getDay : Integer
...

Pseudocode

```
DateTime:::getYear
      RETURN seconds / 31,577,600
```

Hint: What happens on the 19th of January 2038?

Problem 24.3: File Name Class

Classify the level of abstraction of the following class which stores a file name.

FileName
- name: String
- extension: String
- type: FileType
...
+ isGreater(rhs: FileName) : Boolean
...

Pseudocode

```
FileName :: isGreater(rhs)
    RETURN name > rhs.name
```

Note that the `isGreater()` method is used to sort files by their name so they are presented to the user in alphabetical order.

Hint: What happens when the user tries to list “`a.txt`” and “`B.txt`” in the same directory?

Problem 24.4: Chess Piece Class

Classify the level of abstraction of a class designed to store a chess piece on a chessboard. The member variable is a single character where ‘`r`’ corresponds to a white rook and ‘`R`’ corresponds to a black one.

Piece
- value: Character
+ getPossibleMoves() : Move [*]
+ isBlack() : Boolean
+ move(Move)
+ getValue() : Character
...

Note that the `getValue()` method returns the character corresponding to each chess piece.

Problem 24.5: Angle Class

Classify the level of abstraction of a class that stores an angle. This allows the client to work equally with radians (where 2π is a complete loop around a circle) and degrees (where 360° is a complete loop).

Challenges

Challenge 24.1: VB Integer

Research how Visual Basic (VB) represents the `Integer` type. Classify the level of abstraction and justify your reasoning.

Challenge 24.2: Python Integer

Research how Python 2.0 represents the `<class 'int'>`. Classify the level of abstraction and justify your reasoning.

Challenge 24.3: Playing Card

A playing card has the following properties:

There are 52 possible playing cards. Each playing card has two attributes: a suit and a rank. There are 4 possible suits: Spades, Diamonds, Hearts, and Clubs. There are 13 possible ranks: Ace, King, Queen, Jack, 10, 9, 8, 7, 6, 5, 4, 3, and 2.

Design a playing card with complete abstraction. Implement this class in the language of your choice. Create another playing card class with the same public interfaces but with a completely different implementation. Demonstrate that the interface exhibits complete abstraction by showing that the two implementations are completely interchangeable.

A unit test is code written and maintained by developers to exercise a single function, method, or class. The purpose is to identify bugs early in the development cycle, quickly isolate bugs when code is changed, and provide assurances of quality.

Imagine working on a large project with thousands of classes and hundreds of thousands of lines of code. Your job is to add some new functionality and, perhaps more importantly, not break anything. When the change is made and the code is compiled, things don't work quite right. Is the problem the code you just wrote, some of the code you are using, or perhaps how the code is integrated into the overall system? It is very hard to know where to start.

A unit test is white box developer-written automation used to validate a single function, method, or class in isolation of the rest of the program

Unit tests are designed to be a safety net, helping developers identify bugs early in the development cycle, quickly isolate bugs when code is changed, and provide assurances of quality. Though the concept of unit testing has been around since the dawn of the Information Age, and though the name "unit testing" can also be traced back to the 1950s,

there is not a general agreement amongst academics or practitioners as to what exactly constitutes a unit test. Some claim that any testing done on isolated units of software can, by definition, be called a "unit test." In this chapter, we will use a more mainstream definition. Unit tests are white box automation written by developers to validate isolated units of software.

Characteristics of Unit Tests

Unit testing is performed early in the development life cycle, typically before quality assurance engineers first interact with the code. Though there may be black box components to a unit test, they are fundamentally white box tools used to verify that code behaves the way it was designed. They are a form of automation, code written and maintained by developers that resides in the codebase alongside production code. Finally, unit tests are designed to test small units of software, usually an individual function, method, or class.

Characteristic	Description
White box	Validates implementation details.
Automation	Test cases are represented with code in the codebase.
Developer written	The software developer writes and maintains the tests.
Isolated	Validates isolated units of software, not integrated.

Unit testing is one of many types of testing done on a project. Tests are created to validate code at the smallest possible level where individual entities are verified. Test are also created at the highest possible level where the functionality of the entire system is validated.

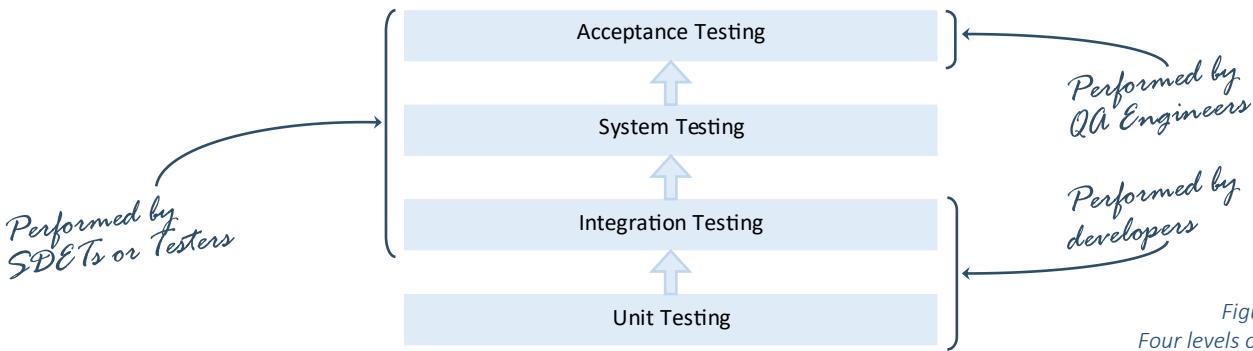


Figure 25.1:
Four levels of testing

Acceptance Testing

Acceptance testing is the final stage in the testing process whose purpose is to determine whether the system is fit for use. It is a form of black box testing and is usually done by quality assurance (QA) engineers. In most organizations, they provide the final "sign-off" that indicates that the project is complete and meets the client's needs.

Acceptance testing is to determine whether the system is fit for use

System Testing

System testing is a holistic review of the product for the purpose of finding defects

System testing is the process of looking at the software system as a whole. This is conducted by software development engineers in testing (SDET). The purpose of system testing is to find defects for

the purpose of making improvements. System testing is a form of black box testing where the SDETs have access to the system, its components, and its requirements specification but cannot view the internal workings of the components themselves.

Integration Testing

Integration testing is the process of checking the interplay between the various units within a single component. This can be performed by developers, but is most often performed by SDETs. Integration testing occurs within the codebase, where individual units are combined with special testing code to observe how they interact. This can be done in a black box way where only public interfaces are tested, or this can be done in a white box way where the internals are monitored during testing. The purpose of integration testing is to discover bugs to improve the overall quality of the product, not to ascertain the quality of the design.

Integration testing is where individual units of software are combined and tested as a group for the purpose of finding defects

Unit Testing

Units are the smallest aspect of a software system that can be individually tested. For procedural, functional, and OO programs, units are usually functions or methods. In OO components, it is common for a unit to be a single class.

Note that a robust set of unit tests is not enough to ensure a high-quality product; integration and system tests are also needed. Please see Chapter 42 Quality: V-Model for a comprehensive description of how to build quality into every stage of development.

Creating a Unit Test

Recall from Chapter 15 Quality: Driver that an automation function implements all the test cases required to verify a function under test. This automation function is called from a test runner which invokes all the automation for the entire project.

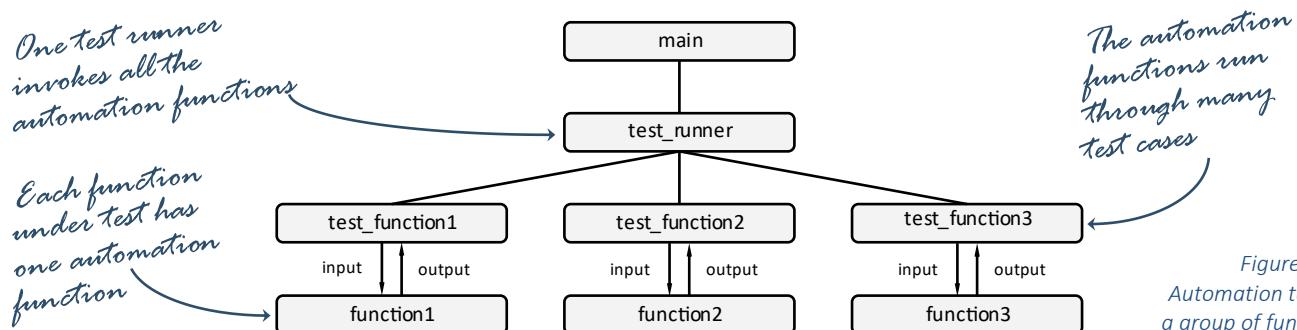


Figure 25.2:
Automation testing
a group of functions

Unit tests operate under the same general framework as automation with one important exception: they are designed to test classes rather than functions. Unit tests are executed by test runners which exercise all the test cases for the entire project. All the tests for a given class are encapsulated in a testcase class, analogous to the automation function which contains all the tests for a function.

The big difference between automation and unit tests is that each test case is contained in a method in the testcase class. This is done because the behavior of a method depends on not just the parameters passed, but potentially on the attributes contained in the class itself. Also, the method under test may modify the class' attribute values in addition to returning a value. Therefore method unit tests are often more complicated than unit tests for functions; a test case typically cannot be expressed in a single line of code as it can with a function test. As a result, unit tests for OO code consist of three components: a test method representing a single test case, a testcase class representing all the test methods needed to validate a single class under test, and a test runner which invokes all the test methods of all the testcase classes in the project.

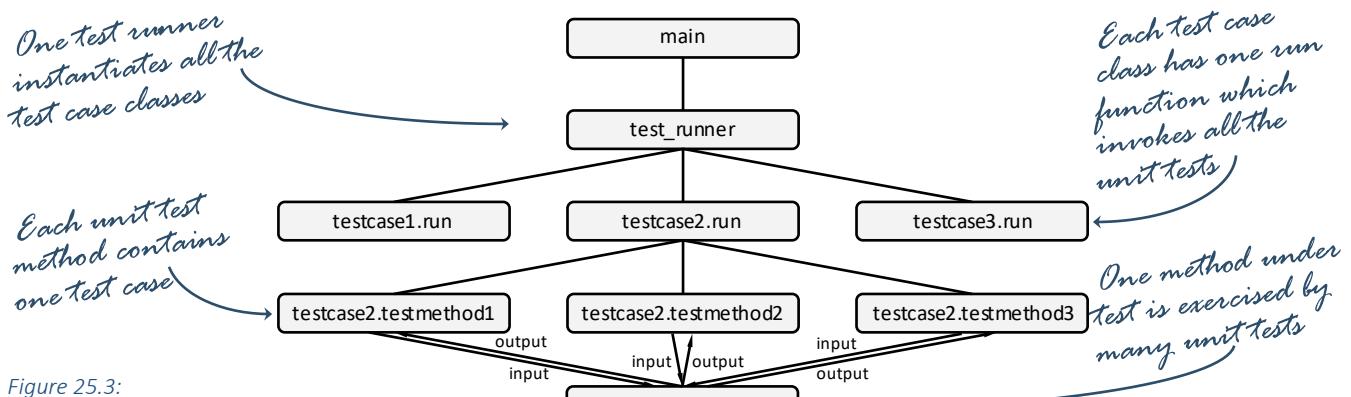
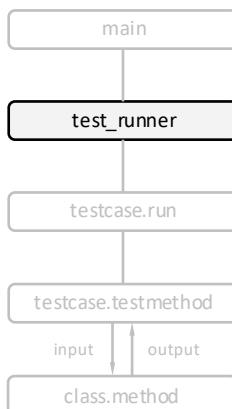


Figure 25.3:
Unit tests testing
a group of classes

Test Runner



In its purest form, the test runner's job is to instantiate a `testcase` object for every `testcase` class in the project. The runner then asks the `testcase` class to execute all the test methods in the class. The runner is invoked by `main()`, usually before any system code is executed. Of course, release builds of the project do not include the test runner so care must be taken to not compile it in those circumstances.

The first unit test framework was built in 2002 for the Java language and is called JUnit. Since this initial release, the unit test framework has been ported to virtually every modern programming language. These frameworks are collectively called xUnit. For example, the Python framework is called PyUnit and the C++ framework is called CppUnit. These frameworks include a test runner which can be executed from the command line or even provide a graphic user interface allowing the developer to select the unit test to be executed.

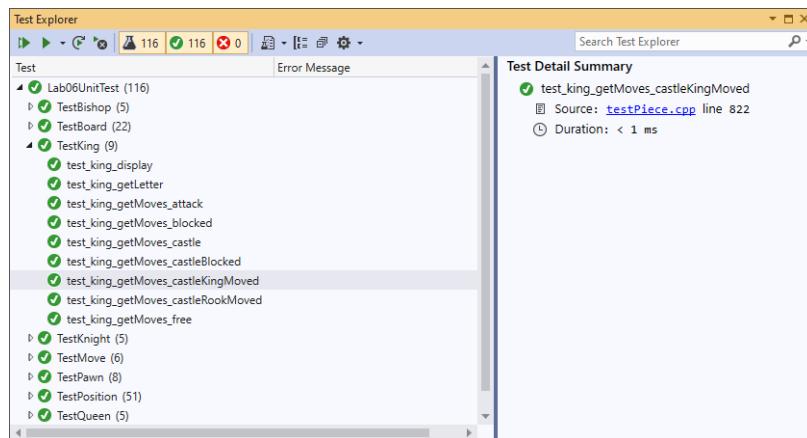


Figure 25.4:
The output of a test runner

In those cases when a unit test framework is not available, an improvised test runner can easily be created. This involves instantiating a `testcase` object for each `testcase` class and invoking the `run()` method.

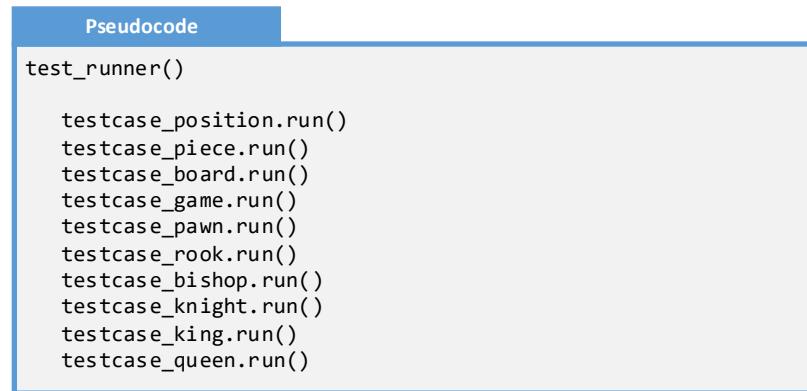
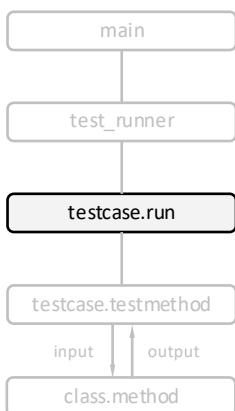


Figure 25.5:
Pseudocode of an
improvised test runner

The biggest problem with the improvised test runner approach is the possibility of missing a test. This would cause the unit tests for an entire class to be skipped. Unit test frameworks in the xUnit family do not have this problem. Because every `testcase` class inherits off the `testcase` base class, the framework code iterates through all derived classes and invokes the `run()` method on each one.

Test Case Class



All the unit tests for a given class are encapsulated in a single `testcase` class. This class always has a `run()` method and many test methods. The `test_runner` function does not directly call the test methods from the `testcase` class, instead they are called through the `run()` method. This can be done one of two ways: either by asking `run()` to invoke all the test methods in the test case class, or by asking for a specific test to execute.

To minimize code duplication in the testing code, it is not uncommon to add utility methods to the `testcase` class to facilitate creation of test methods. The most common methods are `setUp()` which prepares a test to be run, and `tearDown()` which cleans up the system after a test has completed. The following test case class exercises the `Position` class with eight test cases. All the tests are executed by invoking the `run()` method when no parameters are specified. A specific test can be invoked with the second `run()` method.

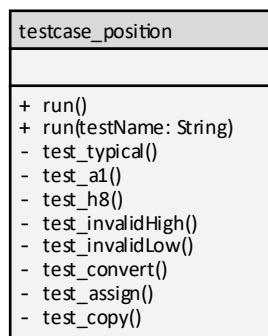


Figure 25.6:
Class diagram of a
test case class

When using an improvised `testcase` class, create one or more `run()` methods to execute the tests. These methods simply enumerate through all the test methods in the class similar to that of a test runner. This technique is called *test enumeration*.

Pseudocode

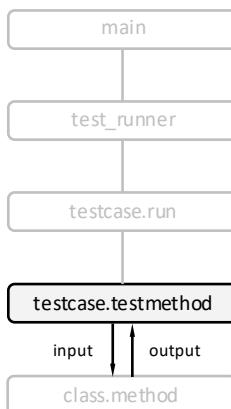
```
testcase_position.run()
  test_typical()
  test_a1()
  test_h8()
  test_invalidHigh()
  test_invalidLow()
  test_convert()
  test_assign()
  test_copy()
```

Figure 25.7:
Pseudocode of
the `run()` method of
a test case class

When using an xUnit framework, things are a bit more sophisticated. First, a single `testcase` object is created for each test to be executed. This is done to ensure that one test does not influence another. Thus, the test runner will always invoke the `run()` function that accepts a test name as a parameter. This uses the factory design pattern described in more detail in Chapter 37 Strategy: Object Creation.

Second, test cases can be added to the `testcase` class by simply creating a method with the word “`test`” at the beginning of the test method name. The xUnit framework utilizes the reflection feature available in most OO languages to enumerate through all the methods in the class. Each test is annotated such as beginning a method with the word “`test`”. This technique is called *test method discovery*. When test method discovery is utilized, the developer can be assured that all the unit tests are invoked.

Test Method



The heart of the unit test process is the test method: a method which executes a single test case. Recall from Chapter 15 Quality: Driver that testing a standalone function involves two steps: passing input to the function and verifying that the output is what was expected. Testing a method is more complex because the method's behavior can be sensitive to the state of the object and the method can also change the state of the object. To account for these differences, test methods have four distinct steps.

Step	Description
Setup	Prepare the test to be run by setting all the pre-conditions that the test needs.
Exercise	Perform the action that is to be verified. This involves calling the method under test.
Verify	Ensure that the output and the new system state is what is expected.
Teardown	The system should be reset so another test can be conducted.

For example, consider a class representing the position on a chess board. This class internally stores position as an `unsigned char` where 0 is the bottom-left corner of the board (position "a1") and 63 is the top-right corner (position "h8"). The method under test is `setRow()` which accepts an integer as input.

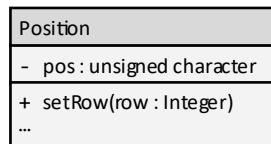


Figure 25.8:
Class diagram of the
class to be tested

The test case will start in position b3 ($\text{pos} = (\text{row}*8+\text{col}) = (2*8+1) = 17$) and move to b5 ($\text{pos} = (\text{row}*8+\text{col}) = (4*8+1) = 33$).

Name	Before	Input	Output	After
Row 3 → 5	$\text{pos} = 17$	$\text{row} = 5$	\emptyset	$\text{pos} = 33$

The test method has four steps, each annotated with a comment. The test method itself never takes a parameter, does not return anything, and fires an assert if an error has been detected. All test methods follow this simple pattern.

Notice how the unit test needs access to the class' private member variables

Pseudocode

```

testcase_position.test_setrow_typical()
# Setup
position.pos ← 17
# Exercise
position.setRow(5)
# Verify
assert(position.pos = 33)
# Teardown
  
```

This unit test does not require a teardown

Figure 25.9:
Pseudocode of
the test method

Observe how the test method has access to the class under test's private member variables. The setup step sets the `pos` member variable and the verify step reads the `pos` member variable.

Setup

The setup step in a test method involves creating a test fixture. A fixture is everything that needs to be in place for the test to run. When testing a standalone function, no fixture is required. When testing a method, an object containing the class to be tested usually needs to be instantiated. If the test is complex, the process of creating a test fixture can be long and involved.

A fixture is everything that needs to be in place for a test to run

It is frequently the case that more than one test case utilizes the same fixture. To avoid unnecessary duplication of code, the fixture creation process can be moved into an auxiliary method. The resulting fixture can be returned from the auxiliary method or stored in the member variables of the `testCase` class.

When all test methods in a testcase class need the same fixture, the fixture creation code can be added to a `setUp()` method. xUnit frameworks guarantee that the `setUp()` method is invoked before the test method is invoked. This is called *implicit fixture setup*. An advantage to using implicit fixture setup is that all test methods in a testcase class have exactly the same fixture. A disadvantage is that the tests themselves are more difficult to understand because the pre-conditions of the test are not readily apparent.

Exercise

The exercise step involves calling the function or method under test. In almost all cases, this is a single line of code. If more than one function or method is called, then we are probably exercising more than one part of the system. While this may be desirable for an integration or system test, tests such as these are not unit tests.

Verify

The verify step checks that the function or method did what was expected. This includes validating the output of the function or method, as well as checking the member variables of the class under test. This is normally accomplished with a collection of asserts.

Verification occurs with asserts

Most verify steps use standard asserts. If the same sequence of asserts is used for many test methods, it may be worthwhile to create a utility method for the test class that contains the assertion sequence. If you choose to create such a method, it should have no parameters and contain no loops or IF statements. The creation of these assertion wrapper methods reduces code duplication but makes each specific test less obvious.

Teardown

The teardown step is to return the system to the same state it was in before the test case was executed. For functions, nothing should happen here unless the function somehow altered global state. For classes, the object from the class under test should be destroyed as well as anything else involved in the fixture.

When the fixture teardown is non-trivial and used by more than one test method, it is common to move the teardown code into an auxiliary function as we did with setup. When every test method uses the same teardown procedure, that shared code should be put in a `tearDown()` method. This method is guaranteed to be called by the xUnit framework much like the `setUp()` method is called.

Best Practices

Most of the best practices presented in this book describe what should be done to maximize the chance that the project will live up to its potential. The unit test best practices are a little different. The best practices containing the word “must” should be considered as commandments rather than as suggestions.

Scope

The first set of best practices relate to the scope or size of the unit being tested and of the test itself.

Best Practice 25.1 The scope of a unit test must be a single function or method

It should be the goal of every unit test author to isolate the class or function under test to the greatest extent possible. When a unit test does too much, it becomes an integration or system test. These tests are certainly important parts of the overall quality plan for the project. However, they are often better written by SDETs and QA engineers. A test method that exercises more than one test case or more than one function/method is called an *eager test*.

Best Practice 25.2 Tests should depend on unit requirements, not on implementation details

This best practice is often worded as “favor testing through the front door over the back door.” Testing through the front door involves using public interfaces. This gives the unit test exactly the same access to the class as everything else in the codebase. Testing through the back door involves accessing the class under test’s private member variables and private methods. This is sometimes required to adequately isolate the method being tested. Since unit tests are white box tests, they have access to all the class’s private methods and variables. However, it is always better to use the front door.

The purpose of unit tests is to verify that the unit under test functions how it is meant to operate. In other words, the unit tests are to verify the unit’s contract, not the implementation details. If we rely too much on back door methods, then we can have *over specified tests*. This means the tests are tied too closely to implementation details. A common side effect of over specified tests is that too many tests need to be changed when making simple adjustments to the code. For example, one would not expect a test to have to change when the developer decided to change the name of a private member variable.

Best Practice 25.3 Try to test every codepath in the unit under test

A test suite with good coverage ensures that at least one unit test exercises every line of code. This increases the chance that a mistake in the unit under test will be caught by a unit test. However, if codepaths are not covered, then the developer has little assurance that that code works as expected. If there is a codepath that is not currently covered with the suite of unit tests, consider adding tests to get full coverage. Note that some unit test frameworks include code coverage reports that can highlight production code not adequately covered by existing unit tests.

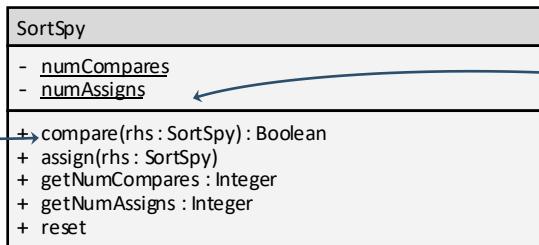
Unit testing on a class can get complicated when the class depends on other classes or functions. These are called collaborators. To better isolate a class from the collaborators, it is often necessary to create a stand-in class. These stand-ins are sometimes called “doubles,” a term borrowed from the movie industry where it may be necessary to incorporate a stunt double or a body double in order to complete a challenging scene. These doubles are passed to the unit under test instead of actual classes used in the program to better the isolation. There are five types of doubles:

Types of Doubles	Description
Dummies	An empty class containing the same interfaces as the class it is meant to replace. All the public methods contain asserts to verify they are not called.
Fakes	A simplified version of the class it is meant to replace that avoids a complex operation (such as file I/O).
Stubs	A simplified version of the class it is meant to replace that always provides a fixed output.
Spies	A stub that also collects information about how it is used. This is particularly useful for instrumentation, counting the number of times it was accessed or used.
Mocks	Sophisticated stubs that replay fixed answers from expected inputs.

Dummies, fakes, and stubs are easy to understand, but spies and mocks are more subtle. To illustrate how a spy works, consider a function that sorts a collection of values. You could create a spy class that keeps track of the number of times a value is copied or compared. The unit test would then call on the sort function with a collection of spy values. When the sort is finished, the spy will report on how many times a comparison or a value copy was needed to sort the collection.

Each time this is called, numCompare is incremented

Figure 25.10:
Class diagram of
a spy double



Store the number of times the assignment or compare function was called

Mocks also can be difficult to understand. Imagine a class called **Calendar** that puts a calendar on the screen. This calendar takes a **Date** object as a parameter. As part of this **Date** class, there is a function which knows whether a given year is a leap year (a necessary function for determining how many days there are in a month). Of course, we would like to test the **Calendar** class independently of the **Date** class. To accomplish this, we will create a **Date_Mock** class. This class will hard-code a few responses to **Date_Mock.numDaysMonth()**. Now we can verify that **Calendar** works as expected without relying on a fully-functional **Date** class.

Very simple implementation with no external dependencies

Pseudocode

```

Date_Mock :: numDaysMonth(year, month)
IF year = 1999 and month = 2
  RETURN 28
IF year = 2000 and month = 1
  RETURN 31
IF year = 2001 and month = 12
  RETURN 31
ASSERT
  
```

Only works for a very small number of cases

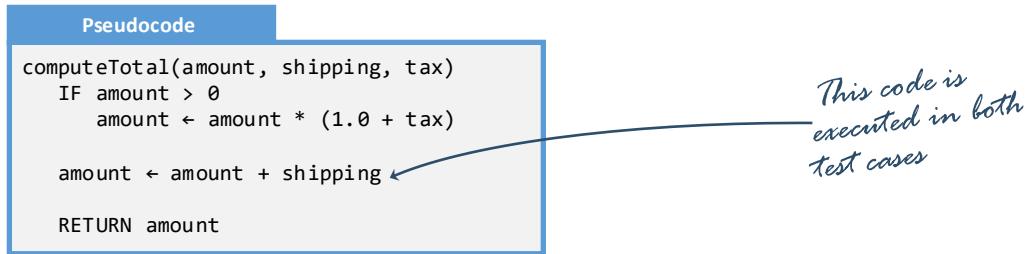
Figure 25.11:
Pseudocode of
a mock double

Best Practice 25.5 Minimize test overlap

This best practice can be somewhat counterintuitive. One might be tempted to think that more test cases results in better test coverage. However, similar test cases tend to fail at the same time. Thus, duplicate test cases do not provide a meaningful increase in test coverage. Duplicate tests simply increase the cost of maintaining the test code.

Ideally, there should be one unit test per test case, and one test case per feature or condition in the unit under test. This means that changing a line of production code primarily impacts only one unit test, and each unit test maps to a single part of the unit under test. Of course, this ideal is not possible or practical in many situations.

Figure 25.12:
Pseudocode of a function
with some test overlap



In figure 25.12, we will need at least two test cases: one that uses a positive value in the `amount` parameter, and one that uses a negative value. Both test cases unavoidably utilize the line of code that adds the shipping charge. Thus, a small amount of overlap is going to have to exist when testing this function.

One of the main advantages of unit tests is that it quickly notifies the developer of an introduced bug. A comprehensive suite of unit tests help provides that assurance. Another great advantage of unit testing is that it simplifies debugging. When there is minimal test overlap, one developer mistake should trigger only a single test failure. This should make it abundantly clear where the bug is located. Test overlap complicates debugging efforts.

Test Creation

The next set of best practices relate to how to generate unit tests.

Best Practice 25.6 Unit tests must be written by the developer who wrote the code to be tested

It was not many years ago when developers were discouraged from testing their own code; that task was left to specialists. The thinking was that there is an inherit conflict of interest: developers want the code to be finished rather than find bugs which could postpone the finish date. By separating the testing task to another individual who is motivated by finding bugs, a more objective and thorough job would be done.

Developers create & maintain unit tests

Unit testing follows a different philosophy. Since unit tests are designed to be white box, only the developer can craft tests that correctly validate the inner workings of the system. It is therefore the developer's job to create unit tests in conjunction with the software that is under test, and to maintain the tests when the design changes.

Best Practice 25.7 Unit tests must be reliable, accurately reporting whether there is a defect in the code

There are four possible outcomes: true negative, true positive, false negative, and false positive.

Possible Outcomes	Description
True Negative	Nothing reported, nothing wrong. This is the ideal outcome of a unit test run.
True Positive	A test failed and there is a defect in the code. Of course, the code should be fixed as soon as possible.
False Negative	Nothing reported but a defect exists. These are so hard to find! An important goal of unit testing is coverage, to minimize the chance that a false negative lurks in the code.
False Positive	A failure is reported but no defect exists. Since testing code is not as high-quality as production code, this is common. Here, the test needs to be refined so false positives are reduced.

Of course, we are happiest when we get a true negative report. However, we should rejoice when we get a true positive. Finding a bug through a unit test is much better than finding it through integration testing, system testing, acceptance testing, or from user feedback. The longer we wait until the bug is found, the harder it is to fix. False negatives are the thing that keep QA engineers up at night. Most engineers would vastly prefer a false positive over a false negative.

The longer we wait until the bug is found,
the harder it is to fix

The most common causes of unreliable tests are the following:

- **Interacting tests:** Two unit tests that share a fixture or depend on each other in some way. This typically happens when a unit test changes system state or when a unit test is sensitive to system state.
- **Resource leakage:** A unit test has a missing or incomplete teardown step, leaving allocated memory or open file handles. Note that this is a special type of the interacting tests problem in that the system state is changed by the unit test.
- **Concurrency dependency:** Some fixtures or systems assume they are the only unit running at a given moment in time. Classes with static member variables or code that accesses global variables are two common culprits. These tests can be unreliable if two unit tests are run concurrently.
- **Nondeterministic test:** Ideally, the exercise step should only be dependent on the class instance under test and any needed fixtures. When this is not the case, nondeterministic tests are the result. An example is testing a function or method calling a pseudo-random number generator.

Best Practice 25.8 Make the unit tests obvious

It is widely known that understandability is a critical quality metric for production code. This is doubly true with test code. There are two reasons for this. First, test code needs to be updated at least as often as production code. When test code contains a bug, it needs to be fixed. When production code contains a bug, both it and the accompanying test code frequently needs to be fixed. For this reason, there is often a great deal of churn in the test code. When the test code is difficult to understand, the cost of maintaining the unit tests can become oppressive.

Second, test code should point to the production code it is meant to test. If a unit test reports a failure, it should be obvious what part of the production code needs to be fixed. Obvious unit tests make the investigation part of fixing a bug much easier. When a unit test is deducible, misleading, or puzzling, then the cost of fixing a bug increases.

One hindrance to obvious tests is called the *mystery guest*. A mystery guest is some mechanism that influences the fixture in a way that is not clear from the test method. Ideally, every detail of the fixture should be completely apparent from the setup step of the test method. When the fixture depends on system state or some outside influence, the fixture and the test itself becomes unpredictable. Mystery guests should be avoided; if one is discovered, the test should be rewritten so the setup process of the fixture is more obvious and consistent.

Best Practice 25.9 Test methods must not have loops.

Every test method must have a completely linear algorithm, progressing from the setup step to the exercise, and on through to the verify and teardown steps. We may be tempted to put a loop in the verify step of a test method to ensure that all the elements in an array are greater than zero. If one element was negative, our assert will notify us. However, it will not tell us which element was zero until we investigate further.

If this assert fires,
which element in
the numbers array is
negative?

Pseudocode

```
testcase_primes.test_getPrimes_positive()
# Setup
# Exercise
numbers ← getPrimes(3)
# Verify
FOR number in numbers
→ assert(number > 0)
# Teardown
```

Figure 25.13:
Pseudocode of a test
method with a loop

Creating a sequence of asserts makes the debugging process much easier. This process is called loop unrolling.

Now there is no
question which
element is at fault

Pseudocode

```
testcase_primes.test_getPrimes_positive()
# Setup
# Exercise
numbers ← getPrimes(3)
# Verify
assert(numbers[0] > 0)
assert(numbers[1] > 0)
assert(numbers[2] > 0)
# Teardown
```

Figure 25.14:
Pseudocode of a test
method with a loop
unrolled

Best Practice 25.10 Test methods must not have conditionals

Since the fixture is completely predictable, there should be no IF statements or other conditional logic. We may also be tempted to use an IF statement to verify that a container is not empty before accessing its elements.

Test will not be run if the numbers array is empty

Pseudocode

```
testcase_primes.test_getPrimes_first()
# Setup
# Exercise
numbers ← getPrimes(1)
# Verify
if numbers.size() > 0
    assert(numbers[0] = 2)
# Teardown
```

Figure 25.15:
Pseudocode of a test
method with a
conditional statement

A better way to accomplish this is to insert a *guard assert* which verifies that the container has elements. After this guard assert, we can access the elements without concern that the test will crash.

This guard assert will fire first, preventing the program from crashing if the array is empty

Pseudocode

```
testcase_primes.test_getPrimes_first()
# Setup
# Exercise
numbers ← getPrimes(1)
# Verify
assert(numbers.size() = 1)
assert(numbers[0] = 2)
# Teardown
```

Figure 25.16:
Pseudocode of a test
method with a
guard assert

Usage

The next set of best practices relate to how to incorporate unit testing into the software production process.

Best Practice 25.11 Unit tests must be executed frequently

Unit tests are meant to be executed often. They should be executed before code is changed to ensure that it starts off in a known good state. By following this practice, you can be assured it was something you just did rather than a pre-existing problem when a test fails after your code modification. Unit tests should also be executed immediately after a modification is made. A change is not “done” until all the unit tests pass. Additionally, they should also be executed periodically as part of a larger test pass through the entire system. This gives the development team assurance that the code is in a good working state. If a unit test requires manual intervention, if the unit test depends on any external entity (such as the time of day or an external file), if the unit test depends on unpredictable state (such as a random number generator), or if the unit test takes too long to execute, then it cannot fulfill these purposes.

Best Practice 25.12 Unit tests must require no effort or user intervention to run

If it is difficult to run unit tests, then it is unlikely that the developers will run them as often as they should. It is good if it is easy to run unit tests. It is better if unit tests are run every time the application is run. The best possible solution is to make it difficult to not run the tests with every execution.

Best Practice 25.13 Unit tests must run quickly

The longer it takes to run unit tests, the less often they will be run. Therefore, a key aspect of a project's unit test strategy is to keep execution time of unit tests short. There are times when this is impossible.

Large projects with thousands of units may have tens of thousands of unit tests. Even if each individual unit test runs quickly, the overall cost of running the suite of tests can be prohibitively expensive. One strategy to mitigate this problem is to have two sets of tests: those to be run frequently, and those to be run on special occasions. The frequent set are a small subset of the total collection of tests chosen to touch on all aspects of the project. For example, one unit test for each unit may be in the frequent subset. This results in a suite that is a mile wide and an inch deep. Therefore, such tests are often called breadth tests. The second set is to be run before each check-in and during overnight builds. They can also be run on-demand if a developer seeks extra assurance that a given change does not break anything. In each case, the faster the tests, the more often they will be executed and the shorter the turnaround between an introduction of a bug and when the bug is fixed.

Best Practice 25.14 Unit test should exist in the codebase alongside the code it is meant to verify

Unit tests are written directly in the codebase, often alongside the code they are meant to test. In the case of testing classes, unit tests must have access to the private member variables and private methods. For this reason, unit tests must be private methods within a class or granted access through the friend mechanism.

A screenshot of a C++ code editor showing a class definition. The code is as follows:

```
C++
#ifndef DEBUG
class TestcasePawn;
#endif // DEBUG

class Pawn
{
public:
    ... code removed for brevity ...

    // Give the pawn unit test access to the Pawn's privates
#ifndef DEBUG
    friend class TestcasePawn;
#endif // DEBUG
    ... code removed for brevity ...
};
```

Handwritten annotations are present:

- A blue arrow points from the text "All reference to the testcase class should be debug only." to the word "DEBUG" in the first preprocessor directive.
- A blue arrow points from the text "When the testcase class is a friend, then it has access to the class under test's privates" to the word "friend" in the friend declaration.

Caption: Figure 25.17: Code for granting access to a testcase class to the privates of a class under

Of course, you should make sure that the code is not included in release versions of the software; use pre-compiler directives or similar mechanisms to ensure this does not happen.

Best Practice 25.15 Make the production code easy to test

One problem developers often encounter is that their existing code or their current design is difficult to verify with unit test. The underlying problem is not with the unit tests, but with the design. Well-designed code (good maintainability, high cohesion, loose coupling, seamless convenience, and complete abstraction) is almost always easy to verify with unit tests. When any of these principles are not honored, things can get quite complicated. Developers knowing that they will have to write unit tests often write better code. Because it is easier to write unit tests with well-designed code, developers tend to design code carefully when they know unit tests need to be written.

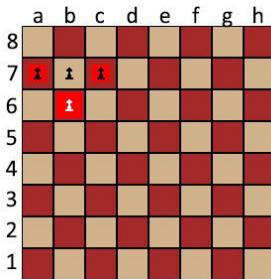
Examples

Example 25.1: Unit Test in C++ Visual Studio

This example will demonstrate how to create a Visual Studio unit test.

Problem

Create a unit test to verify that the `Pawn::getMove()` method in a chess program will work correctly when a pawn can capture two other pieces.



Solution

Visual Studio's unit test utility recognizes any class defined with the `TEST_CLASS` macro to be a test case class and any method defined with `TEST_METHOD` to be a test method. Observe how a stub for the `Board` class is used to create a simplified board for the purpose of isolating this test case from the rest of the system.

C++

```
namespace ChessUnitTest {
TEST_CLASS(TestPawn) {

    TEST_METHOD(test_pawn_getMoves_capture) {
        // SETUP
        BoardStub board;
        Pawn * pawn = new Pawn(5, 1, true /*isWhite*/);
        board.placePiece(new Pawn(6, 0, false /*isWhite*/));
        board.placePiece(new Pawn(6, 1, false /*isWhite*/));
        board.placePiece(new Pawn(6, 2, false /*isWhite*/));
        board.placePiece(pawn);

        // EXERCISE
        pawn->getMoves(m, board);

        // VERIFY
        Assert::AreNotEqual(m.end(), m.find(Move("b6a7p")));
        Assert::AreNotEqual(m.end(), m.find(Move("b6c7p")));
        Assert::AreEqual(2, (int)m.size());
    } // TEARDOWN
}
}
```

Example 25.2: JUnit in Java

This example will demonstrate how to create a JUnit test in Java.

Problem

Create a JUnit to test an `isLeapYear()` method within a class containing the notion of a year.

Year
- year
+ isLeapYear() : Boolean
...

Solution

Java requires us to include the JUnit library to implement our unit tests. Note that JUnit will collect all methods marked with the `@Test` annotation. Also, the testcase class needs to derive from the `TestCase` base class.

```
Java
package junittestcase;
import junit.framework.TestCase;

public class TestcaseYear extends TestCase {

    @Test
    public void testIsLeap2000() {
        Year y = new YearImpl();           // SETUP
        y.year = 2000;
        boolean result = y.isLeap();       // EXERCISE
        assert(result == true);          // VERIFY
    }                                     // TEARDOWN

    @Test
    public void testIsLeap2001() {
        Year y = new YearImpl();           // SETUP
        y.year = 2001;
        boolean result = y.isLeap();       // EXERCISE
        assert(result == false);          // VERIFY
    }                                     // TEARDOWN

    @Test
    public void testIsLeap2004() {
        Year y = new YearImpl();           // SETUP
        y.year = 2004;
        boolean result = y.isLeap();       // EXERCISE
        assert(result == true);          // VERIFY
    }                                     // TEARDOWN
    ... code removed for brevity ...
}
```

Example 25.3: Test Case Class in C++

This example will demonstrate how to create an improvised test case class using test enumeration.

Problem

Create a testcase class in C++ to exercise the unit tests for a `Pawn::getMove()` method.

Solution

To test a `Pawn` class, we will create a `TestcasePawn` class that inherits from the `Testcase` base class. This class will be made a friend in `Pawn` so it can have access to `Pawn`'s privates.

```
C++  
#ifndef NDEBUG  
*****  
* TEST CASE PAWN  
* Test the Pawn class  
*****  
class TestcasePawn public Testcase  
{  
    public:  
        void run()  
        {  
            TestGetMoves_StandardSimple();  
            TestGetMoves_InitialAdvance();  
            TestGetMoves_Capture();  
            TestGetMoves_EpPassant();  
            TestGetMoves_Promotion();  
        }  
  
        void TestGetMoves_StandardSimple();  
        void TestGetMoves_InitialAdvance();  
        void TestGetMoves_Capture();  
        void TestGetMoves_EpPassant();  
        void TestGetMoves_Promotion();  
};  
#endif //NDEBUG
```

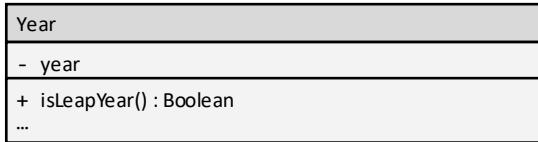
Notice that the unit test is surrounded by `#ifndef NDEBUG` to ensure it will not find itself in shipping code. When a `TestcasePawn` object is created, the test runner will have to call `TestcasePawn.run()`. This method will then call all the unit tests associated with `Pawn::getMoves()`. One final thing to notice. This testcase is a class even though it does not have any member variables. It could just as easily have been done as a collection of functions. If this approach was followed, however, every single unit test would have to be specifically mentioned as a friend in the class under test. Because of this, most unit tests for classes are classes themselves.

Example 25.4: NUnit in C#

This example will demonstrate how to create a NUnit test in C#. Note that NUnit is the xUnit name for all languages using the .NET framework. This includes C#, VB, and F#.

Problem

Create a NUnit test for a `Year.isLeapYear()` method



Solution

We will create a `TestcaseYear` class that contains two nested classes: `ValidYear` and `InvalidYear`. Though only `ValidYear` is presented here, `InvalidYear` would include negative years and other years of invalid state.

C#

```
using System;
using NUnit.Framework;

public class TestcaseYear
{
    public class ValidYear
    {
        Year y;
        public ValidYear()
        {
            y = new Year();
        }

        [Theory]
        [InlineData(1756)]
        [InlineData(1796)]
        [InlineData(1804)]
        [InlineData(1948)]
        [InlineData(1996)]
        [InlineData(2004)]
        [InlineData(2020)]
        public void WithinCenturyLeapYears(int yearTest)
        {
            y.year = yearTest;                // SETUP
            bool result = y.isLeapYear();      // EXERCISE
            Assert.True(result);             // VERIFY
        }
        ... code removed for brevity ...
    }
}
```

Notice the NUnit framework allows us to use the same test method to go through many test cases: 1956, 1796, 1804, 1948, 1996, 2004, and 2020. The next step would be to create `WithinCenturyNotLeapYear()`, `CenturyLeapYear()`, and `QuadCentury()` cases. We can get good coverage with very little code using NUnit.

Example 25.5: Spy in C++

This example will demonstrate how to create a spy double in C++.

Problem

Create a spy to compile performance metrics of a sort function.

C++

```
template <class T>
void sort(T array[], int num);
```

We wish to know the number of times the comparison and the assignment operator are called.

Solution

C++

```
class SortSpy
{
public:
    SortSpy() : value(0) { }
    SortSpy(const SortSpy & rhs) { *this = rhs; }

    // Manage the statistics
    void reset() { assign = compares = 0; }
    long getAssigns() const { return assign; }
    long getCompares() const { return compares; }

    // execute > and record the number of compares
    bool operator > (const SortSpy & rhs) const
    {
        compares++;
        return value > rhs.value;
    }

    // execute = and record the number of assigns
    SortSpy & operator = (const SortSpy & rhs)
    {
        assign++;
        value = rhs.value;
        return *this;
    }
private:
    int value;           // the value
    static long assign; // assign operator count
    static long compares; // greater operator count
};

long SortSpy :: assign = 0;
long SortSpy :: compares = 0;
```

Notice that we increment `assign` and `compares` with each call of the corresponding method. When we call `sort()` passing an array of `SortSpys`, then we can more accurately instrument the algorithmic efficiency of our function.

Exercises

Exercise 25.1: Types of Testing

In your own words, list the four types of testing. Compare and contrast each according the following properties:

- In what phase of the project is each type performed?
- Who performs each type of testing?
- Is the type of testing black box or white box?
- Is the purpose to find bugs or determine the quality of the project?

Exercise 25.2: Parts of a Unit Test

From memory, list the four parts of a unit test and explain the purpose of each.

Exercise 25.3: Isolation

Each unit test should exercise a single method or function in isolation of the rest of the program. Explain in your own words why this is fundamental to unit tests.

Exercise 25.4: Doubles

In your own words, define each of the types of doubles and explain how they can be implemented in a unit test:

Fact or Fiction	Justification
Dummies	
Fakes	
Stubs	
Spies	
Mocks	

Exercise 25.5: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
We wish to avoid false positives	
Unit tests are most often written by testers	
Every method under test should have several unit test methods	
Unit tests typically use manual drivers	
Acceptance and system testing are the same thing	
A unit test should avoid true positives	

Problems

Problem 25.1: Compute Pay Unit Test

Consider the following pseudocode.

Pseudocode

```
computePay(hours, wage)
    IF hours < 40.0
        RETURN hours x wage
    ELSE
        RETURN (wage x 40.0) + (wage x 1.5 x (hours - 40.0))
```

In the programming language of your choice create unit tests for this function. Use the following test cases for your unit test.

Name	Input (Hour, Wage)	Output
Zeros	0, \$0.00	\$0.00
No time	0, \$8.00	\$0.00
One hour	1, \$8.00	\$8.00
No wage	1, \$0.00	\$0.00
Just under full time	39, \$10.00	\$390.00
Full time	40, \$10.00	\$400.00
Overtime	41, \$10.00	\$415.00
Double time	80, \$10.00	\$1,000.00

This pseudocode does not handle some cases correctly. Implement these unit tests and fix the algorithm so they pass.

Name	Input (Hour, Wage)	Output
Negative hours	-1, \$10.00	error
Negative wage	1, -\$8.00	error
Unreasonable hours	168, \$10.00	error

Problem 25.2: Is Prime Unit Test

In the programming language of your choice, implement a unit test for the following function which computes whether a number is prime:

Pseudocode

```
isPrime(number)
    FOR divisor ← 2 ... √number
        IF number % divisor = 0
            RETURN false
    RETURN true
```

Create test cases and implement a unit test to verify this function.

Problem 25.3: Coordinate Unit Test

Consider a class designed to represent the position of a piece on a chessboard (from Figure 20.3):

Coordinate
- value : Integer
+ Coordinate
+ getRow : Integer
+ getCol : Integer
+ set
+ display
+ input
- convertFromText
- convertToText
- isValid : Boolean

Please do the following:

1. Implement this class in the programming language of your choice
2. Enumerate a set of test cases for each of the public methods
3. Create a test harness for the class as was done in Example 25.3

Problem 25.4: User Unit Test

Consider a class designed to represent an authenticated user on a secure system where every User has a username, a password, and a level of permission.

User
- name : String
- pass : String
- permission : Integer
+ init(name : String in, password : String in, permission : Integer in) : Boolean
+ validate(name : String in, pass : String in) : Boolean
+ haveAccess(permission : Integer in) : Boolean
+ getName() : String
- permissionValid(permission : Integer in) : String

Please do the following:

1. Implement this class in the programming language of your choice
2. Enumerate a set of test cases for each of the public methods
3. Create a test harness for the class as was done in Example 25.3

Challenges

Challenge 25.1: Selection Sort Function

Create test cases for a function that takes an array as a parameter (including the length of the array if that is not provided in your language of choice and modifies the array so it becomes sorted. Note that this can sort any type of data as long as the passed data type honors the assignment and less-than operators.

Pseudocode

```
selectionSort(array, num)
    FOR iPivot ← num-1 ... 1
        iLargest ← 0
        FOR iCheck ← 1 ... iPivot -1
            IF !(array[iCheck] < array[iLargest])
                iLargest ← iCheck
            IF array[iPivot] < array[iLargest]
                swap array[iLargest], array[iPivot]
```

Please do the following:

1. Implement this function in the programming language of your choice
2. Enumerate a set of test cases
3. Create a spy double as was demonstrated Example 25.5
4. Create a suite of unit tests for this class

Challenge 25.2: Function Unit Test

Identify a complex function in any project you have completed. With this function:

1. Enumerate a set of test cases
2. Create a suite of unit tests for this class
3. If any bugs are found, then fix them and run your test cases again

Challenge 25.3: Class Unit Test

Identify a class in any project you have completed. With this class:

1. Enumerate a set of test cases
2. Create a suite of unit tests for this class
3. Create doubles as necessary to isolate this class from the rest of the code you wrote
4. If any bugs are found, then fix them and run your test cases again

Test-Driven

Chapter 26

Test-Driven Development is a software development process where tests are written before the corresponding production code is written to minimize the lag between bug introduction and removal.

The conventional software development life cycle goes from requirements elicitation to design to development to test. This is true whether the life cycle is measured in years in the traditional “waterfall” model or if it is a one-week sprint in SCRUM.

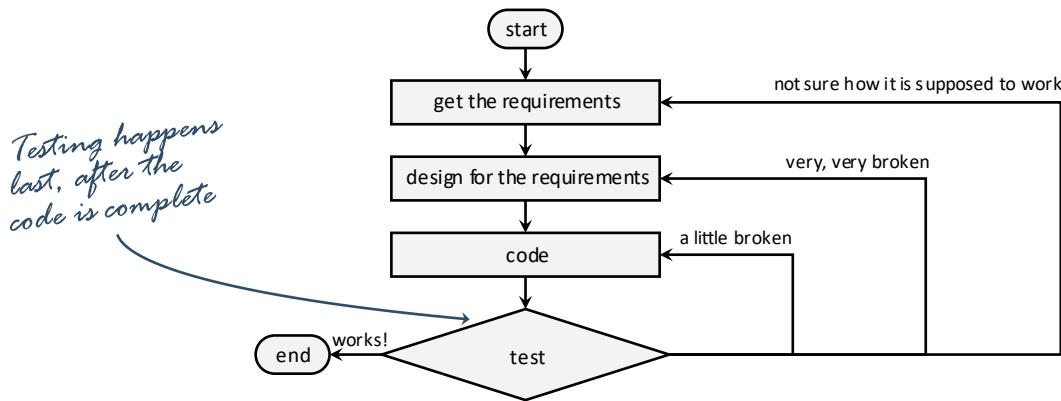


Figure 26.1:
Traditional software
development process

Test-Driven Development (TDD) represents a drastic reordering of these activities. TDD places traditional testing activities before the coding phase and before the design phase. It is, in fact, a part of the requirements elicitation phase because the unit tests are the specification.

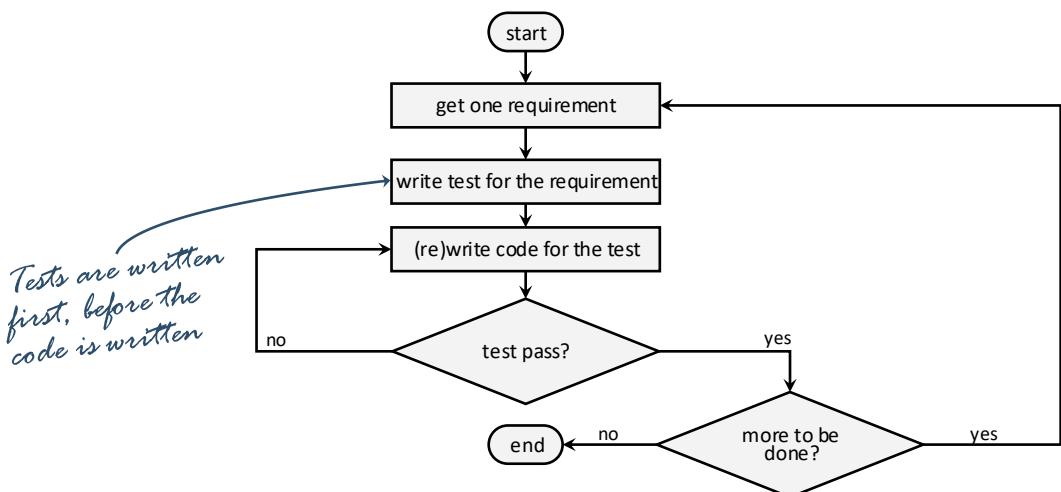


Figure 26.2:
TDD software
development process

The TDD Promises

In the early days of computer programming, before the term “software engineer” was even invented, code was written in a simple two-step process: code and fix. The outcome of this methodology was poorly designed code that was difficult to maintain. To address this defect, another step was added in the process: design, code and fix. While this was certainly an improvement, it did not scale well with large teams and complex projects. In the 1960s, a more rigid methodology was developed: document the requirements, document the design, document the code, and document the testing. Again, the problems of the previous generation were largely addressed and this methodology was common through the 1980s and into the 1990s. At the end of this era, a problem began to surface. By the time the code was written and testing began, the code was basically untestable. Furthermore, since the time was large between a defect being introduced and found, the cost of fixing bugs was very high. It was not uncommon for a one-year development effort to consist of six weeks of feature work and eight months of testing/debugging (the balance spent on requirements elicitation, design, and effort estimation). Clearly, something needed to be done.

TDD was developed during the end of this “software crisis” era. It promised to address the most serious ills of the day: poor test coverage, large lag between a defect being introduced and fixed, and overengineered code.

Eliminate Poor Test Coverage

Code coverage is the percentage of code that is activated during a test pass

Test coverage (also known as “code coverage”) is the percentage of code that is activated during a given test pass. If code is not exercised during a test pass, then it has an unknown level of quality. Clearly, the goal of a quality assurance engineer is to ascertain the quality of all the code in the system and thus achieve 100% code coverage. In a traditional black box testing environment, it is very difficult to exercise a given line of code. It is even more difficult to tell whether all possible code paths are executed. The result is very poor test coverage.

Under TDD, every single line of code has a test that exercises it. Note that the previous sentence was written in a very “code then test” way. It would be more accurate to say “under TDD, production code is not written unless a test fails that requires it.” A true TDD developer will refuse to write a single line of code unless that line of code is necessary to pass a test.

TDD promises 100% code coverage

Eliminate Overengineering

Overengineering is the process of making a design more complicated than necessary in anticipation that more functionality will be needed in the future. When developers become more fixated on the beauty of the design or the elegance of the code than they are on the needs of the client, then the temptation exists to overengineer the product. TDD strives to eliminate overengineering by making sure that no code is added to the project unless a test (and hence a client requirement) exists that demands it. Since all tests require a client requirement, it is impossible to create a test representing future need. Therefore, any overengineered code cannot have a corresponding test and should not be in the product.

Minimize Lag

Lag refers to the amount of time a bug exists in the code. We can compute lag by comparing the date and time that a bug is introduced and when it is finally fixed. Lag has some interesting properties:

Lag Duration	Consequence
a minute–15 minutes	The developer has the code in short-term memory and can localize the bug to a few dozen lines of code
15 minutes–an hour	The developer can easily recall the code and narrow the bug to a hundred lines of code
an hour–a day	It may take a few minutes to collect all the necessary information about the code where the bug resides
a day–a week	It may take the developer 15 minutes to localize the bug and remember how everything works
a week–several months	A developer who introduced a bug has little advantage over a novice who never seen the code before

From the standpoint of efficiency, it clearly makes sense to minimize lag. In the traditional “waterfall” development methodology, it is commonly several weeks between when a bug is introduced and fixed. With TDD, it is measured in minutes.

TDD minimizes lag because only one test is being “worked on” at a time. The developer is notified of the defect immediately because it is the developer who runs the test. Since the code is not finished until the test passes, a bug never exists in the system for very long.

Lag refers to the amount of time a bug exists in the code before it is fixed

Tight Development Cycle

The software development cycle is the set of activities between the initiation of a development effort and when the effort is marked as done. Ideally, “done” means the software is ready for release. In a traditional development environment, the software development cycle is measured in months or even years; the system is only ready for release for a few moments during that time!

The TDD development cycle is ultra-short, measured in minutes, not months

TDD has an ultra-short development cycle. It begins when one small requirement is taken from the queue and ends when the newly created unit test passes and the code is refactored. This is often measured in a few minutes. Thus, it is possible to release the software to the customer several times during the day; the project is always in a state that is very close to being ready.

Better Design

Because it is easier to write tests for simpler interfaces and for more cohesive modules, there is an implicit pressure on developers to favor better designs with TDD. They shy away from overly complicated designs with tight coupling, convoluted convenience, and porous abstraction because the unit tests are simply too difficult or complicated to write!

The TDD Process

In the simplest sense, TDD is a five-step process. Of course, there are some complicated interplays between these steps.

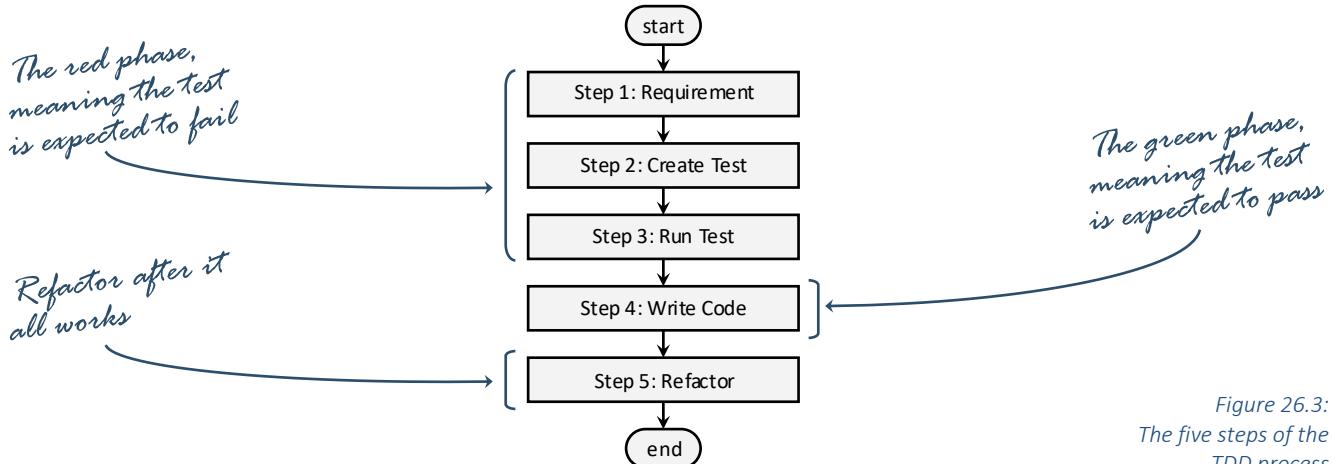


Figure 26.3:
The five steps of the
TDD process

As simple as this may seem, others adopt an even simpler model: red-green-yellow or sometimes red-green-refactor. The “red” in the red phase corresponds to a failed unit test. This is because many unit test frameworks indicate that a test failed with a red icon or red background color. This is the origin of the phrase “I got a red bar.” The red phase corresponds to Steps 1, 2, and 3 in the above flowchart.

The “green” in the green phase corresponds to a successful unit test, also originating from the user of the green color in a unit test report. This directly correlates to Step 4 in the above flowchart. Note that a green test often means that the project is fit for use, even if it is engineered poorly. One poorly engineered function or class will not bring the system down. However, it is the accumulation of these functions or classes which might. This accumulation is called “technical debt.” When technical debt becomes excessive, then the project becomes unmaintainable and expensive to enhance. This leads us to the third part: yellow or refactor.

The TDD cycle is often called red-green-refactor. Red for a failed test, green for a passed test, and the refactor step

The “yellow” in the yellow phase does not have any special meaning in the unit test world. The color is simply chosen because it is the last color in a stop light. This is why many call the process “red-green-refactor.” It is this refactor phase which helps us from accumulating too much technical debt.

Step 1: Get a Requirement

A fundamental component of TDD is that every line of code is tied to a test and every test is tied to a client’s need. Some have even preferred to call it Behavior Driven Development (BDD) where requirements are expressed in terms of behaviors. There

are four sub steps in the requirements process: 1.a requirements elicitation, 1.b subdivide the requirements, 1.c sort the requirements by dependency, and 1.d get the next requirement in the list if there is one.

Every line of code is tied to a test and every test is tied to a client’s need

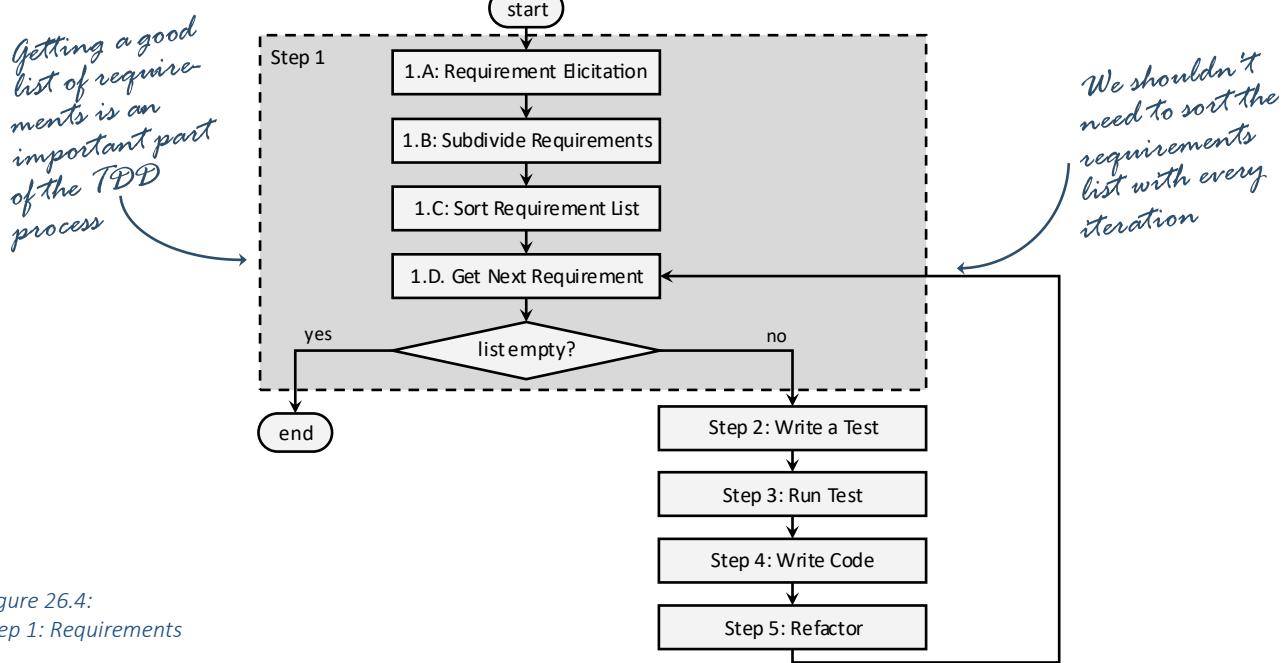


Figure 26.4:
Step 1: Requirements

1.A: Requirement Elicitation

Before beginning work on a product, feature, class, or function, a developer should seek to obtain the most complete picture possible of what is to be built. When doing so, it is necessary to keep in mind that it is very difficult for a stakeholder to express all their needs, it is even more difficult to do so in a precise manner. Therefore, all requirements gathered from the client should be understood to be rough estimations of what is to be built rather than an ironclad contract.

1.B: Subdivide the Requirements

Requirements most easily map to test cases when they are small, discrete, and precise. It is therefore necessary to translate the raw requirements obtained from the client into smaller chunks. The benefit from this process is that the client's needs are better understood the more the requirements are dissected.

In the SCRUM world, requirements begin as epic user stories. These need to be broken down to large, then medium, and finally small stories that developers can manage. TDD follows the same pattern where only small-sized requirements can be readily converted into test cases.

1.C: Sort the Requirement List

Eventually, all the requirements will be implemented. The order in which they are done, however, is usually at the discretion of the developer. When this is the case, it is important to order the requirements, completing the easiest and less complicated ones first and the corner-cases last.

1.D: Get the Next Requirement

TDD completes one requirement at a time. The developer works through the requirements list, moving from simple and mainstream cases into more complex ones. This continues until the requirement list is empty and the process begins anew.

Step 2: Write a Test

It may seem strange at first, but tests in TDD are written before the code they are meant to exercise. This is done with full expectation that they will fail the first time through. Each test is a unit test. Please see Chapter 25 Quality: Unit Test for more details on how to write unit tests if you have not read that chapter already.

Tests in TDD are written before the code they are meant to exercise. We expect them to fail the first time through

TDD tests tend to fall into the unit test framework. This means that each test cases is represented in one or more test function, and all the test functions are encapsulated in a test class. By tradition, the name of the test class is derived from the name of the class it is meant to test with the word "Test" appended on the front. Thus, the class `UserName` would be accompanied by `TestUserName`. The same is true for functions under test. The function `insertionSort()` would be accompanied by `testInsertionSort()`.

The test functions themselves have very verbose names drawn from the test case

Usually the test functions themselves have very verbose names. In fact, test functions are typically the full name of the test case. Consider the class `UserName` and the method under test called `setName()`. There may be a requirement "no spaces are allowed in the username" and the corresponding test case "check that there are no spaces in the username." The corresponding test function for this case might be named something like: `TestUserName::TestSetName_CheckForSpaces()`.

It is tempting to think about the implementation of the code when you are writing the test case. This is a mistake! Focus only on writing the best test case possible. Think about the interface you are testing as if you were the client. In other words, focus on the convenience encapsulation metric. Only after the test is written and executed do we even think about implementation. This is a drastic departure from traditional testing where we write a test for the purpose of exercising the code. Instead, we write the code for the purpose of passing the test.

Don't write a test to exercise the code. Instead, write the code for the purpose of passing the test

Step 3: Run the Test

The first time a newly written unit test runs, the developer has no expectation that it will run. In most cases, it will not even compile! This is expected and by design. Note that if, by some miracle, the test does pass, then apparently your code has unexpected functionality. It is a good idea to double-check your test code to make sure you do not have a false-negative.

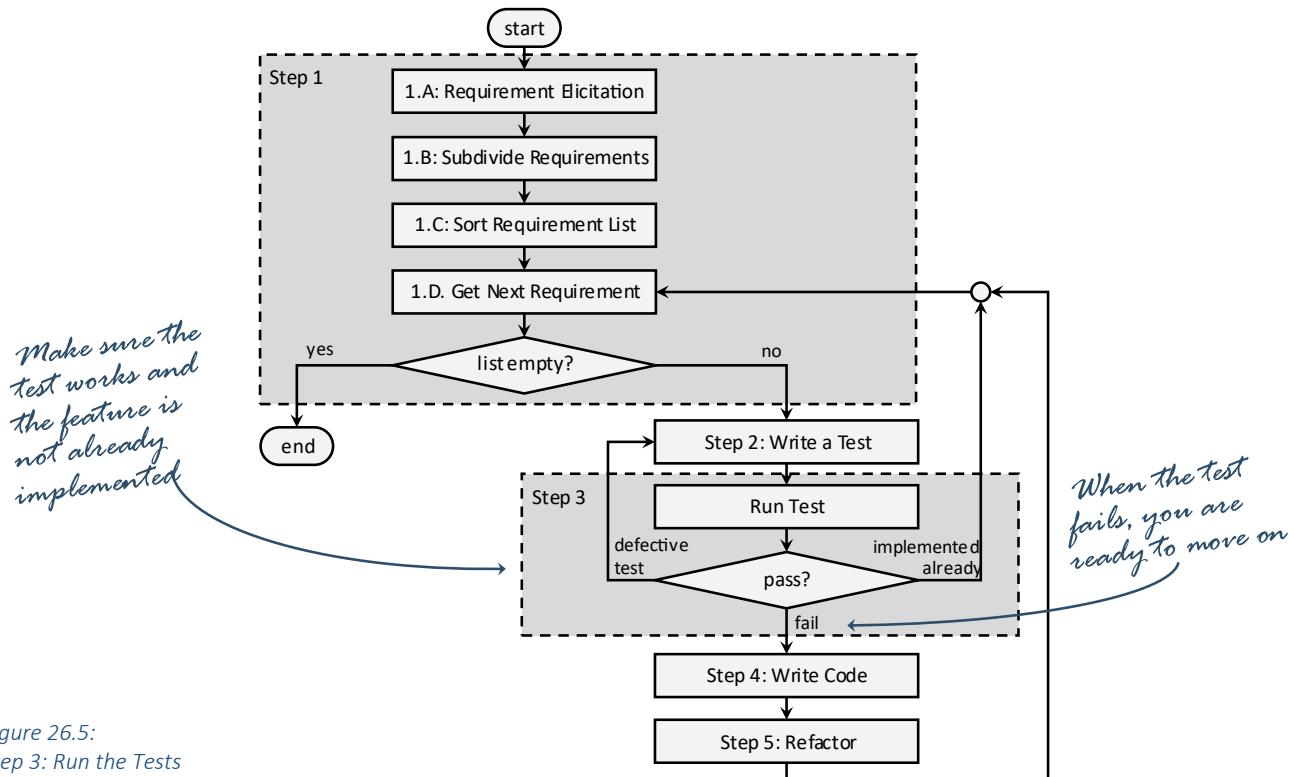


Figure 26.5:
Step 3: Run the Tests

Some may question the utility of running a test before the necessary code is written. To illustrate its purpose, imagine what would happen if the test is completed and miraculously the test passes. That was unexpected! From this singular event, two very important outcomes could be the result.

First, the feature could already be implemented. It is difficult to imagine in a small and isolated program when all the code is readily apparent. In large and complicated codebases, especially those utilizing powerful libraries, this is somewhat common. When this happens, you not only have saved yourself some work, but you have also avoided code duplication! That is a very good thing.

Second, the test itself could be defective. Since we rely on these unit tests as an important component on our overall quality process, we need to have high-quality tests as well as high-quality code.

Step 4: Write the Code

The next step is to write just enough code to pass the test. It is important to not write too much code, or optimize it for performance or maintainability or another metric. At this stage, we are just trying to get the tests to pass. First, we get the new test to pass, then we make sure that we have not broken anything. This process continues until the newly written code works.

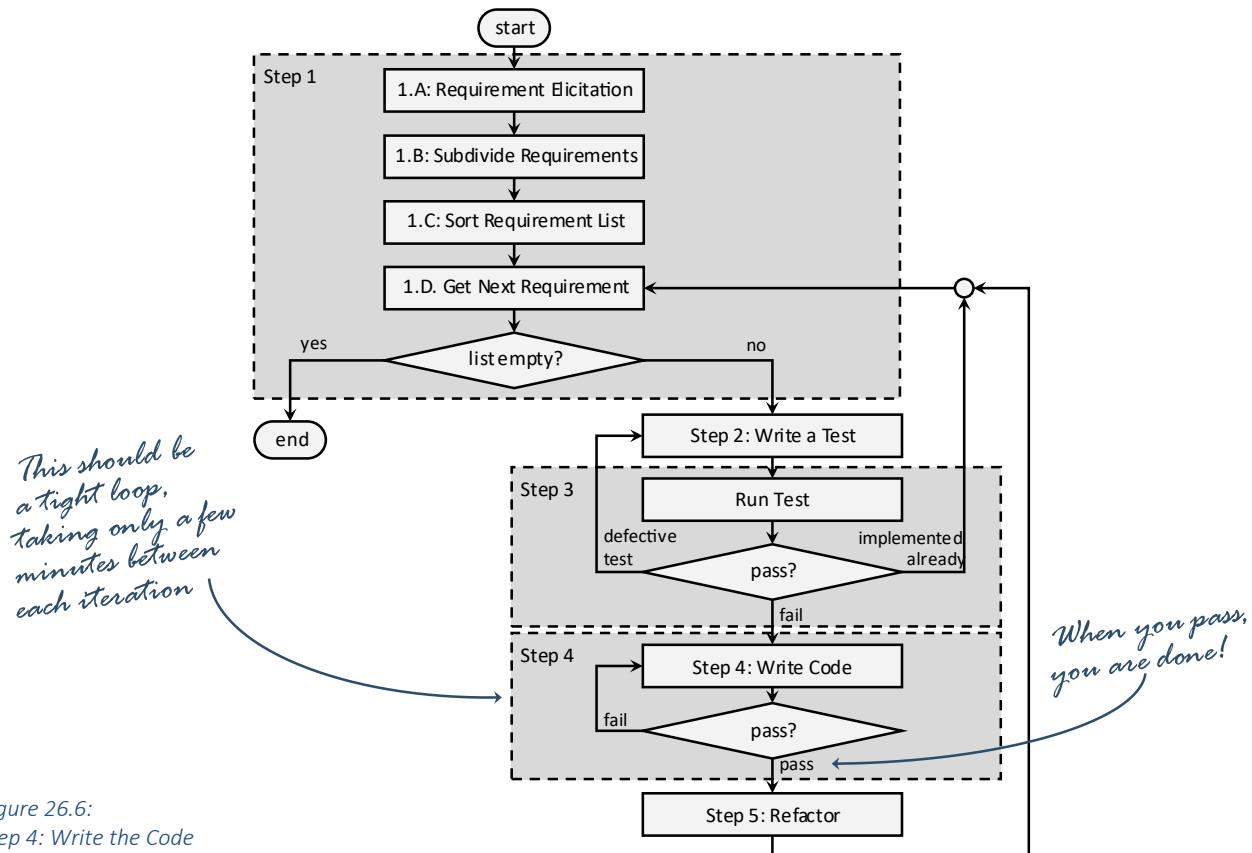


Figure 26.6:
Step 4: Write the Code

It is important to focus on writing the smallest amount of code possible to pass the associated test (and not break another test!). While it is acceptable to think about best coding practices in Step 4, do not be tempted to refactor just yet. Instead, focus only on getting the required functionality into the product.

It is a common pitfall for developers new to TDD to refactor in Step 4. Any efforts to optimize on performance or code-size will likely be lost as you just try to get the code to work. It is far better to worry about these things in a separate step after the test has passed.

Best Practice 25.1 Keep It Simple (KISS)

Kelly Johnson, a famous aircraft engineer at Lockheed responsible for developing the P-38, U-2, SR-71, and the F-117 (among other things), coined a famous design principle: Keep It Simple (KISS). The KISS design principle states that simple designs tend to work better than complex ones. At this stage of the TDD process, KISS should be honored.

Step 5: Refactor the Code

The Boy Scouts of America have a guideline: "Leave the campground better than you found it." This simple rule also applies to software development. Once a change to the code has been shown to work (through passing the associated unit tests), then it should be refactored according to the best software development practices. When finished, the code should be better than when you started.

Leave the code better than you found it

Best Practice 25.2 Before refactoring working code, make sure you have a backup

Refactor bugs can be subtle and difficult to fix. Attempting to refactor code without a robust set of unit tests is risky. Simple code inspections rarely reveal the problem. However, doing so with strong unit tests brings a certain level of confidence.

Best Practice 25.3 It is easier to re-refactor than it is to fix refactoring bugs

If you find that your refactoring caused a unit test to fail, it is most often easier to back out of the refactor and start again. It will probably take less time and certainly less angst. You will probably do a much better job the second time anyway! The ultra-short development cycle of TDD means that reverting to a backup rarely loses more than a few minutes of development effort.

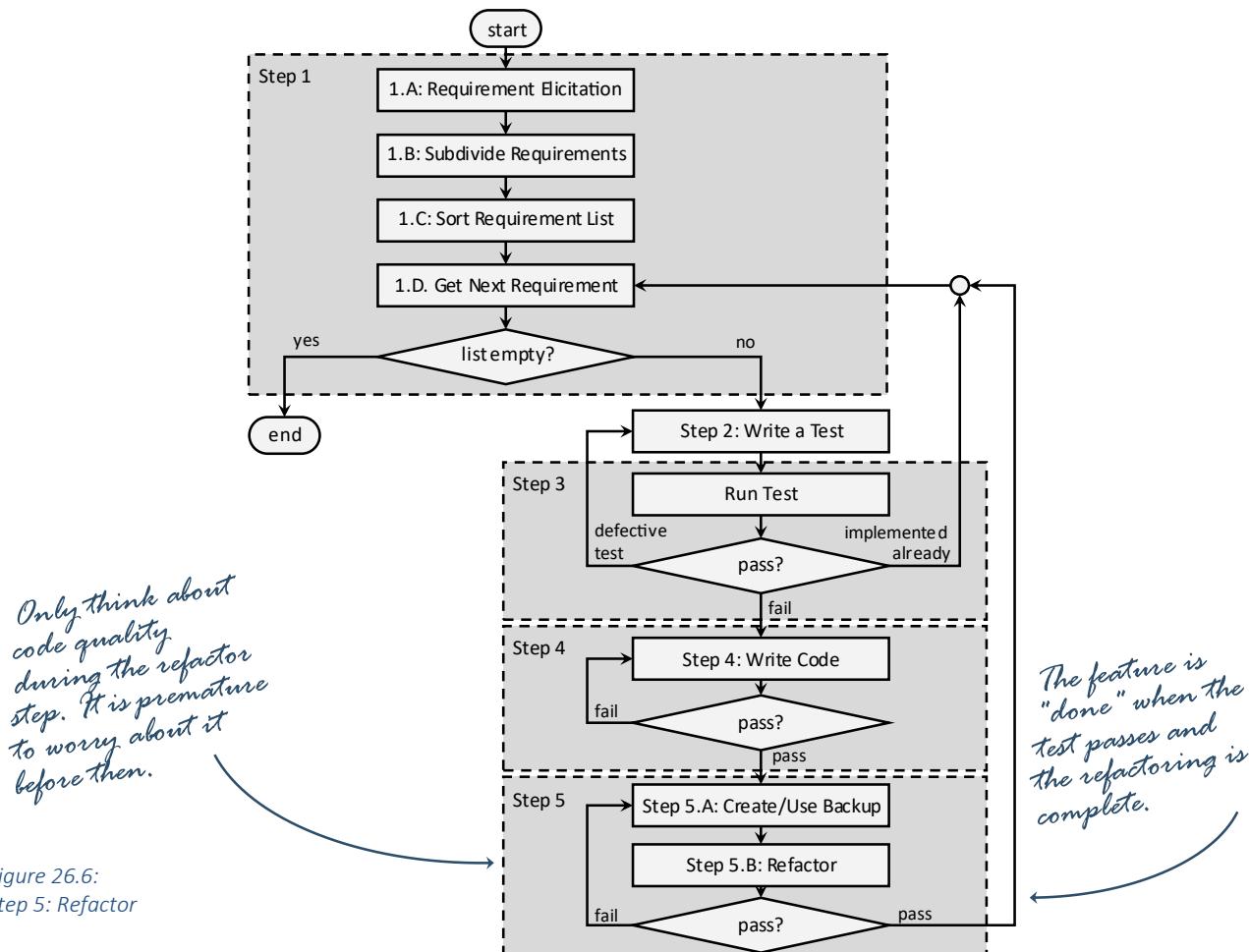


Figure 26.6:
Step 5: Refactor

Examples

Example 26.1: Requirements

This example will demonstrate how to create a requirements list using TDD.

Problem

An interview with a stakeholder has yielded the following requirement. Turn this into a requirement list suitable for TDD.

All text presented to the user must be word-wrapped. This means that white spaces are appropriately turned into line breaks in such a way that there are no more than LENGTH characters on a given line. This LENGTH value will depend on the amount of real estate available in the interface. This word-wrap functionality must handle tabs (8 characters width) and other non-printable characters.

Solution

There are four sub steps in the Requirement step of the TDD process: requirements elicitation, subdivide the requirements, sort the requirements, and get the next requirement.

The first step of the process is to elicit requirements from the client. This is given as part of the problem definition, so sub step 1.A is complete.

The next step (1.B) is to subdivide the requirements. These include:

Requirements

All tab stops are on the 8-character boundary.

A space will be converted to a newline before the line length exceeds LENGTH.

If there is a run of spaces that cross the LENGTH boundary, the last space is converted to a newline

A space on the LENGTH boundary will be converted to newline

If a newline is encountered, then the line length is reset

If there are no spaces in a run longer than LENGTH, the line will exceed LENGTH

The next step (1.C) is to sort the list of requirements, from the easiest with the fewest dependencies, to the hardest. The second appears to be the most straightforward, followed by the fourth, then the first. The sorted list is:

Sorted Requirements

A space on the LENGTH boundary will be converted to a newline

A space will be converted to a newline before the line length exceeds LENGTH

If a newline is encountered, then the line length is reset

All tab stops are on the 8-character boundary

If there are no spaces in a run longer than LENGTH, the line will exceed LENGTH

If there is a run of spaces that cross the LENGTH boundary, the last space is converted to a newline

Example 26.2: One Requirement

This example will demonstrate how to complete Step 2–5 of the TDD process by completing a single requirement.

Problem

Turn a simple requirement into working code using the TDD process:

A space on the LENGTH boundary will be converted to a newline

Solution

We will start with Step 2: Write a Test. Lacking a test framework in our programming language of choice, we will manually create the unit test. Here, the simplest test we can create is to insert a single line break.

```
C
void wordWrapTest_SpaceAtLength() {
    /* SETUP          01234567890123456789 */
    char input[]     = "a a a a a a a a a";
    char expected[] = "a a a a a\na a a a";

    /* EXERCISE */
    wordWrap(input, 10);

    /* VERIFY */
    assert(0 == strcmp(input, expected));

} /* TEARDOWN */
```

Next, we will run the test in Step 3. This will initially fail:

```
Output
a.out: example02.c:19: void wordWrapTest_SpaceAtLength(): Assertion `0 == strcmp(input, expected)' failed.
Aborted
```

The code is written in Step 4. It passes the test.

```
C
void wordWrap(char * text, int lineLength) {
    if (text[lineLength - 1] == ' ')
        text[lineLength - 1] = '\n';
    return;
}
```

Note that at this point, there is no refactoring necessary (Step 5) so this development cycle is complete.

Example 26.3: One Requirement

This example will demonstrate how to complete Step 2–5 of the TDD process.

Problem

Turn a simple requirement into working code using the TDD process:

A space will be converted to a newline before the line length exceeds LENGTH.

Solution

Step 2: We need a test case where the newline appears before the LENGTH.

```
C
void wordWrapTest_SpaceBeforeLength() {
    /* SETUP          01234567890123456789 */
    char input[]     = "a a a a xxxx a a a a";
    char expected[] = "a a a a\nxxxx a a a a";

    /* EXERCISE */
    wordWrap(input, 10);

    /* VERIFY */
    assert(0 == strcmp(input, expected));

} /* TEARDOWN */
```

Now that we have more than one test case, we also need a function to collect all our test cases.

```
C
void wordWrapTest() {
    /* A space on the LENGTH boundary
       will be converted to a newline */
    wordWrapTest_SpaceAtLength();

    /* A space will be converted to a newline
       before the line length exceeds LENGTH */
    wordWrapTest_SpaceBeforeLength();

    puts("All tests pass!");
}
```

Step 3: The test will be compiled and run. It will fail of course.

Output

```
a.out: example03.c:42: void wordWrapTest_SpaceBeforeLength():
Assertion `0 == strcmp(input, expected)' failed.
Aborted
```

Example 26.3 continues on the next page...

... Example 26.3 continued from the previous page.

Step 4: We will implement the code necessary to pass the function. It does pass.

```
C
void wordWrap(char * text, int lineLength) {
    while (text[lineLength] != ' ')
        lineLength--;
    if (text[lineLength] == ' ')
        text[lineLength] = '\n';

    return;
}
```

Step 5: The code will be refactored. Here, we are going to add some comments and put a common-sense check that we are not to go off the end of our buffer.

```
C
void wordWrap(char * text, int lineLength) {
    /* loop until a space is found. Don't go off the end! */
    while (lineLength > 0 && text[lineLength] != ' ')
        lineLength--;

    /* insert a newline if a space is found */
    if (text[lineLength] == ' ')
        text[lineLength] = '\n';

    return;
}
```

At this point, the requirement is complete and we are finished with this development cycle.

Example 26.4: One Requirement

This example will demonstrate how to complete Step 2–5 of the TDD process.

Problem

Complete the following requirement into working code using the TDD process:

If a newline is encountered, then the line length is reset

Solution

Step 2: We need a new test case.

```
c
void wordWrapTest_MultipleLines() {
    /* SETUP          01234567890123456789 */
    char input[]     = "a a b b c c d d e e";
    char expected[] = "a a\nb b\nc c\n d d\ne e";

    /* EXERCISE */
    wordWrap(input, 4);

    /* VERIFY */
    assert(0 == strcmp(input, expected));
}

/* TEARDOWN */
```

```
c
void wordWrapTest() {
    /* A space on the LENGTH boundary
       will be converted to a newline */
    wordWrapTest_SpaceAtLength();

    /* A space will be converted to a newline
       before the line length exceeds LENGTH */
    wordWrapTest_SpaceBeforeLength();

    /* If a newline is encountered,
       then the line length is reset */
    wordWrapTest_MultipleLines();

    puts("All tests pass!");
    return;
}
```

Step 3: The test will be compiled and run. It will fail of course.

Output

```
a.out: example04.c:54: void wordWrapTest_MultipleLines(): Assertion `0 == strcmp(input, expected)' failed.
Aborted
```

Example 26.4 continues on the next page...

... Example 26.4 continued from the previous page.

Step 4: This step requires a great deal of work. We need to put all the previous code in a loop and execute it more than once. This will require a new string pointer to keep track of the beginning of each line and index variable distinct from the line length variable. This eventually passes the test.

```
C
void wordWrap(char * text, int lineLength) {
    int textLength = strlen(text);
    int i;
    char * line = text;
    while (textLength > lineLength) {
        for (i = lineLength; i > 0 && line[i] != ' '; i--)
            ;
        if (i > 0) {
            line[i] = '\n';
            line += i + 1;
            textLength -= i + 1;
        } else {
            line += lineLength;
            textLength -= lineLength;
        }
    }
}
```

Step 5: The code will be refactored. There are some duplicates to be removed and code to be simplified, as well as comments and better variable names.

```
C
void wordWrap(char * text, int lineLength) {
    int textLength = strlen(text);
    int indexLastSpace;
    char * line = text;

    while (textLength > lineLength) {
        /* find index of last space */
        for (indexLastSpace= lineLength;
             indexLastSpace> 0 && line[indexLastSpace] != ' ';
             indexLastSpace--)
            ;
        /* no space is found */
        if (indexLastSpace== 0)
            indexLastSpace= lineLength;
        /* insert newline */
        line[indexLastSpace] = '\n';
        line += indexLastSpace+ 1;
        textLength -= indexLastSpace+ 1;
    }
}
```

This refactor also passes the tests. At this point, the requirement is met and we are finished with this development cycle.

Example 26.5: One Requirement

This example will demonstrate how to complete Step 2–5 of the TDD process.

Problem

When looking at the requirement from Example 26.4, we realized that a case was missing.

With each newline, the line length is reset

This can mean both that multiple newlines are needed in the output text, and that a newline might be encountered in the input text.

Solution

Step 2: We need a new test case.

```
C
void wordWrapTest_EMBEDDEDNewLine() {
    /* SETUP          01234567890123456789 */
    char input[]     = "a a\nb b c c ddd e e";
    char expected[]  = "a a\nb b c c\nddd e e";

    /* EXERCISE */
    wordWrap(input, 10);

    /* VERIFY */
    assert(0 == strcmp(input, expected));
}
```

Notice the newline at column 3 in the input buffer. Now we have two tests for a single requirement:

```
C
...
/* If a newline is encountered,
   then the line length is reset */
wordWrapTest_MultipleLines();
wordWrapTest_EMBEDDEDNewLine();

...

```

Step 3: The test will be compiled and run. It will fail of course.

Output

```
a.out: example05.c:76: void wordWrapTest_EMBEDDEDNewLine():
Assertion `0 == strcmp(input, expected)' failed.
Aborted
```

Example 26.5 continues on the next page...

... Example 26.5 continued from the previous page.

Step 4: Our old approach was to skip to the end of the line and scan backward for a space. This cannot work with this new test case: we need to scan forward, because a newline in the input buffer will change everything.

```
C
void wordWrap(char * text, int lineLength) {
    char * p = NULL;
    int lengthCurrent = 0;

    while (*text) {
        if (*text == ' ') {
            p = text;
            lengthCurrent++;
        } else if (*text == '\n')
            lengthCurrent = 0;
        else
            lengthCurrent++;
        if (lengthCurrent >= lineLength && p) {
            *p = '\n';
            p = NULL;
            lengthCurrent = 0;
        }
        text++;
    }
}
```

Step 5: The code will be refactored.

```
C
void wordWrap(char * text, int lineLength) {
    char * pLastSpace = NULL; /* remember the last space */
    int lengthCurrent = 0; /* length of current line */

    /* look at each character one at a time */
    while (*text) {
        switch (*text) {
            case ' ': /* remember the last space */
                pLastSpace = text;
                lengthCurrent++;
                break;
            case '\n': /* reset if a newline is found */
                lengthCurrent = 0;
                break;
            default: /* just advance if a space */
                lengthCurrent++;
        }

        /* wrap if longer than the line and space */
        if (lengthCurrent >= lineLength && pLastSpace) {
            *pLastSpace = '\n';
            pLastSpace = NULL;
            lengthCurrent = 0;
        }

        text++; /* advance to the next character */
    }
}
```

Exercises

Exercise 26.1: Steps of the TDD Process

From memory, explain of each step in the TDD process. Identify the order in which the step appears and any sub steps that may exist.

Step	Explanation
Write Test	
Write Code	
Get a Requirement	
Run Test	
Refactor	

Exercise 26.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
The TDD development cycle is measured in weeks	
The green phase corresponds to the code writing step	
TDD promises 100% code coverage	
You create a test immediately after you write the code	
A bug with a lag of one minute takes about as long to fix as a bug with a lag of a week	
TDD uses essentially the same steps in the same order as the waterfall methodology	

Exercise 26.3: Development Activities

For each of the following, identify where a given development activity should appear in the TDD process. For each activity, justify your answer.

Activity	Justification
Fix bugs that are causing the unit test to fail	
Create a unit test that we have every expectation will fail to pass	
Optimize the code for maintainability	
Put the easiest and less complicated requirements at the head of the list	
Optimize the code for abstraction	
Optimize the code for performance	
Implement new features	

Problems

Problem 26.1: Is Leap Year Requirements

Complete Step 1 (and the four sub steps) of the TDD process for a function which computes whether a given year is a leap year:

According to the Gregorian calendar, which began use in 1753, years evenly divisible by 4 are leap years, except for centurial years that are not evenly divisible by 400. Therefore, the years 1700, 1800, 1900 and 2100 are not leap years, but 1600, 2000, and 2400 are leap years.

Problem 26.2: Is Leap Year Centuries

Complete Step 2–5 of the TDD process for a function which computes whether a given year is a leap year. Assume that you already have a leap year function that handles the quad years (evenly divisible by 4) and unit tests to verify it. For this problem, fulfill the following requirement:

Most years that are evenly divisible by 100 are NOT a leap year

See Example 25.1 and 25.2 for an idea of what the unit tests may look like.

Problem 26.3: Is Leap Year Quad Centuries

Complete Step 2–5 of the TDD process for a function which computes whether a given year is a leap year. Assume that you already have a leap year function that handles the quad years (evenly divisible by 4) and century years (evenly divisible by 100 from Problem 26.2). Assume you also have unit tests to verify these requirements. For this problem, fulfill the following requirement:

Years that are evenly divisible by 400 ARE a leap year

Problem 26.4: Is Leap Year Complete

Complete Step 2–5 of the rest of the requirements you found in Problem 26.1. Also, put all the unit tests under a single test function so they can be executed with a single function or method call.

Problem 26.5: Coordinate Requirements

Complete Step 1 (and the four sub steps) of the TDD process for a class which represents a position on a chessboard:

A chessboard consists of 64 locations: 8 rows and 8 columns. The user can enter a location using “long algebraic notation”. This means that every column has a letter (a–h) and every row has a number (1–8). Therefore “C2” is the third column and the second row. Of course, the user can use uppercase or lowercase letters and can even get the order mixed up. Thus, “c2” means the same thing as “2C.” The location must be stored internally as a single unsigned character. There needs to be a getter for both the row and column, a setter taking a string, and a display function displaying the chess algebra on the screen.

Problem 26.6: Coordinate Valid

Complete Step 2–5 of the TDD process for a class which represents a position on a chessboard. Start with the following test case:

Lowercase a–h and digit 1–8 in that order map to valid coordinates.

Problem 26.7: Coordinate Reversed

Complete Step 2–5 of the TDD process for a class which represents a position on a chessboard. Start with the following test case:

Reverse the order of letter and digits works the same.

Problem 26.8: Coordinate Invalid

Complete Step 2–5 of the TDD process for a class which represents a position on a chessboard. Start with the following test case:

Letters outside a–h and digits outside 1–8 are ignored/rejected.

Challenges

Challenge 26.1: War

Consider the card game "War:"

War is played with two or more players. The deck of 52 standard playing cards is evenly distributed to all the players of the game. With each turn, all the players simultaneously lay down the top card from his or her stack. The player with the highest card wins the turn and collects all the cards. If there is a tie, then each player turns up one card face-down and another face-up. Again, winner takes all or the tie procedure continues. The game ends when one player has all the cards in the deck.

Implement the **Card** class for the game of War in the programming language of your choice using the TDD methodology.

Challenge 26.2: Budget

Consider a budget application:

This application will track your monthly budget. It will represent all the budget categories that are important to you. It will also allow you to update your monthly spending and give you a report of how much is left in each budget category.

Implement the **Category** class for a budget application in the programming language of your choice using the TDD methodology.

Noun Identification

Noun Identification is the process of deriving programming entities directly from requirements.

A blank sheet of paper for an author and a blank canvas for a painter can be a very intimidating thing; how does one even start? Programmers face the same challenge. Somehow a special arrangement of variables, loops, functions, and classes will produce working software that will delight the client. The noun identification process is a very simple but surprisingly effective technique for helping programmers get started on a new project or feature.

Noun identification is the process of discovering programming entities by systematically combing through requirements

Noun identification is the process of carefully combing through client-provided requirements to discover programming entities. This technique was pioneered by Russell Abbott (1983) but popularized by Grady Booch. Noun identification is based on the realization that we create software to model real-world things. The closer we can map the client's design concerns into programming constructs, the better the resulting product will meet the client's needs. In other words, noun identification is a process designed to maximize fidelity.

As with all design strategies, noun identification is just a starting point, a structured way of thinking about design. However, just as a solid methodology is no substitute for creativity and ingenuity, no amount of inspiration can compensate for poor discipline. Programmers need to be both systematic and creative.

From Parts of Speech to Parts of a Program

In the simplest terms, nouns are classes or variables and verbs are functions or methods. However, the type of noun or verb can reveal a little bit more:

Part of Speech	Programming construct	Example
Common Noun	Class	<u>Deck</u> of playing cards
Proper Noun	Object	<u>Bob's</u> poker hand
Adjective	Member variable	This card is a <u>spade</u>
Action Verb	Function or Method	<u>Dealing</u> a playing card
Having Verb	Class in a class	A deck <u>has a</u> card
Be Verb	Inheritance	A hand <u>is a</u> type of deck

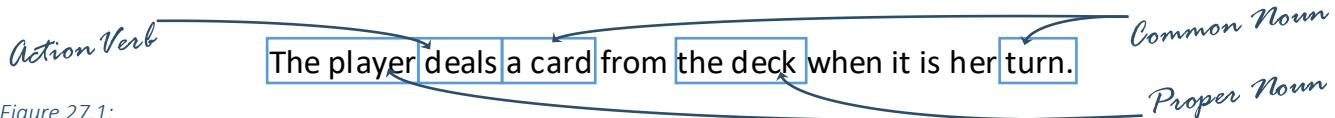


Figure 27.1:
Parts of speech in a
software requirement

Common Nouns

As we are all taught in elementary school, a noun is a “person, place, or thing.” In the context of computing, common nouns represent types of things whereas a proper noun represents an instance of a thing. For example, “girl” is a common noun while “Sarah” is a proper noun. There are many individuals in this world that are girls, but Sarah is a reference to an individual.

Common nouns represent data types, the things we use to represent an individual instance. When we encounter a common noun in a requirement, we should ask ourselves several questions:

Key questions for common nouns

- What does this data type know?
- What does this data type represent?
- What kind of information does it contain?
- How can we best represent this information?

With these questions understood, the next step is to figure out how we are going to codify this concept as a programming entity. Specifically, we can use a built-in data type, or we can invent our own.

For example, we may encounter the common noun “coordinate” which represents the position on a chessboard. This could be two integers, a tuple of two values, an array of two values, a structure with two member variables, or a class. Each of these approaches will be functionally equivalent, but one is the best. Which is it? There are no hard-and-fast answers here.

Best Practice 27.1 Before settling on a data type, enumerate the possibilities

We are usually better at selecting an option from a list of possibilities than we are to come up with the best answer “out of the blue.” It is helpful to jot down a few ideas and perhaps listing pros and cons before settling on an answer.

Best Practice 27.2 Longevity benefits from formality

When a variable’s lifespan exists completely within a single function and is never shared with a collaborator, then a lightweight solution is probably better. This is especially true if it is used in the program only once. Here, it is best to favor built-in data types such as simple variables, tuples, or arrays. However, when a variable exists within a class, when it is passed between functions, or when it is utilized many times throughout the application, then a higher degree of formality is probably better. This is especially true if more than one developer will use it. In cases such as these, favor a custom data type such as a class.

Best Practice 27.3 Only use a class when there is no better alternative

Many new object-oriented designers create more classes than is necessary. This results in an overly complicated design, poor performance, and slow development time. The inclusion of a class in the design should improve the design, and the onus is on every class to prove its worth. In other words, do not overengineer.

Best Practice 27.4 Classes represent things, not collections of verbs

A class should always represent a thing. It should never represent a collection of verbs. Every single class should have member variables.

Proper Nouns

A proper noun is a name used for an individual person, place or thing. To use a slightly more formal way of saying the same thing, a proper noun refers to a singular instance of a common noun. In most human languages, proper nouns have initial capital letters. This is not a hard-and-fast rule. If a noun refers to an individual, then we treat it as a proper noun even though it is not given a formal name.

Proper nouns usually map to variables. These could be simple variables made from a built-in data type, or they could be objects built from a class. Oftentimes, proper

nouns are hidden in a sentence. For example, one might say "I think my cat is hungry." This sentence does not appear to have a proper noun in it. The common noun "cat" is referring to an entire class of

animals, each of which has four legs and a tail. However, this sentence is referring to a specific cat, namely my cat. The sentence could just as easily have read "I think Boots is hungry." The second sentence makes it easy to pick out the proper noun, the first is more subtle. However, both sentences mean exactly the same thing.

Proper nouns are variables or objects

When a proper noun is identified, there are a collection of questions that need to be answered:

Essential questions for proper nouns

What is the data type of this variable?

When is this variable to be created?

Who uses this variable?

Does this variable ever change its value? If so, under what conditions?

When will this variable be destroyed?

The first question (what is the data type of this variable) refers to the ontology of the variable, namely how this variable relates to the rest of the program.

Best Practice 27.5 Identify the data type of variables and objects

When a variable is discovered in the requirements elicitation process, it is probably an important one. We need to have a crisp understanding of its data type and/or any related class. For example, if this variable is an object, then we need to know the associated class. If this variable is part of an operation, we need to know the function or method in which it resides. It may take quite a few questions to fully understand the context in which this variable is used.

The remaining four questions center on the life cycle of the variable. A variable residing in a simple function may have a trivial life cycle that requires very little to understand. A member variable in a class (called an attribute) or a variable that is passed between functions may be quite a bit more complex.

Best Practice 27.6 Create a DFD for important variables

If a variable important enough to be discovered in a requirement, then a thorough understanding of its life cycle is needed. We need to know how the associated data entered the program, the various transformations they went through, where they were stored, and how they were used. A DFD is a great tool to discover these answers.

Adjectives

An adjective is a word describing a noun. For example, the phrase “low fuel” contains both the adjective “low” and the noun it describes (“fuel”). When we talk about adjectives as things, we often use the term attribute. In other words, our fuel may have many attributes: quantity, quality, and type.

Recall that classes represent system state, and this state is represented by the collection of member variables contained therein. These member variables are also called attributes, in no small part because they map to the natural language notion of adjectives. When an adjective is discovered in the requirements, a few questions should be asked:

An adjective is usually an attribute in a class, but sometimes a value for a noun

Essential questions for adjectives

- What is the associated noun?
- What are the alternatives?
- Who else can use this adjective?
- Is this attribute permanent or changeable?

Every single adjective has an associated noun. Most often the noun is obvious, located next to the adjective. Sometimes it is implied, and we need to look for it.

Best Practice 27.7 When an implied noun is discovered, make it explicit

When the wording of a requirement is such that the noun associated with an adjective is not directly stated, then rewrite the requirement so it is abundantly clear. Remember that requirements are not meant to be works of literary art; they are meant to be unambiguous.

Best Practice 27.8 Enumerate the alternatives

Adjectives usually represent a broader class of possible values. In the case of our “low fuel” phrase, it would probably be a mistake to have a class attribute called “low.” Instead, we would want one called `fuel_level` which could have many possible values. Perhaps the values are Boolean (`low/not_low`), much like the low fuel light in your car. It might be nominal (`full, half, low, empty`). It might even be an interval value (`100%–0%`). Ask the client these questions and codify them in the design once you reach an agreement.

Best Practice 27.9 If an adjective can be changed, discover how

Some adjectives are permanent (often called “naming attributes”): an ace of spades, for example, will always be black (whereas diamonds and hearts are red). Here, color would be a permanent attribute. Some adjectives have changeable values: a ship may have a fuel attribute which is constantly changing. If a changeable attribute is discovered, its entire life cycle needs to be understood just as it would be for a proper noun. This probably means a DFD will need to be created to document this life cycle.

One final note about adjectives: sometimes they do not map directly to program entities. They are often used in spoken languages to clarify or emphasize. In cases like this, disregard them.

Action Verbs

A verb is a word used to describe action or state. In the context of software, there are three flavors: action verbs (describing action or something being done), having verbs (describing possession), and being verbs (describing relations). Action verbs often correspond to functions or methods.

Action verbs most often correspond to functions or methods

Verbs can be difficult to identify because they can be implied. For example, one may say “The gun’s bullet will stop when it hits a target.” There is a very important verb which is implied but not stated, “fired.” We could reword the requirement to “The gun will fire a bullet. The bullet will stop when it hits a target.” When an action verb is discovered in the requirements, a few questions should be asked:

Essential questions for action verbs

- Is this part of a larger process? Does it have subprocesses?
- What is the input and output?
- Is there a closely associated noun?

Action verbs do not automatically correspond to functions or methods. Sometimes they are part of large processes that eventually map to dozens or even hundreds of functions. For example, “The [Go] button will send an e-mail” has the doing verb “send.” This single word, however, corresponds to a complicated set of steps which will probably result in a few thousand lines of code. This could be a single function, but it probably should not.

Best Practice 27.10 Ask cohesion questions with every action verb

As soon as an action verb is discovered, immediately start asking cohesion questions. Begin the process of identifying subprocesses using a structure chart, making every function strongly cohesive.

Almost every function has input and output, be that external through interactions with the user or the file system, or internal through parameter passing. As soon as an action verb is discovered in the requirements, start the process of identifying the input and output.

Best Practice 27.11 Identify all the proper nouns and common nouns associated with every action verb

In many cases, the input and output for a function is well understood, utilizing previously described data types and variables. Oftentimes, however, an action verb will refer to something new. Be on the lookout for hidden parameters; many a class is discovered because it is needed to fulfill an action verb found in a requirement.

Best Practice 27.12 Methods can be distinguished from functions when there is a strongly associated noun

Determine whether the verb is closely associated with a noun or whether it is standalone. When there is an associated noun, then we are probably talking about a method. Hopefully, at this point the corresponding class has already been discovered and described. When there is no associated noun, this verb is probably a function. Note that there are many cases when it could go either way.

One final warning: be cautious of nouns that are masquerading as verbs. For example, “the user must be able to undo all edits.” It looks like “undo” is a verb. It will rewind the previously performed action. However, “undo” is actually a noun, referring to an undo record placed on a stack. These are very hard to spot!

Having Verbs

While action verbs describe what a noun is doing, “having” verbs describe how a noun is contained. For example, the sentence “the car has four wheels” does not imply that the car is doing anything with the wheels. It only states that the four wheels are part of a car. Thus, “having” verbs do not map to functions or methods. Instead, they describe membership.

In the simplest sense, “having” verbs describe items in an array (“I have three As and two Bs on my report card”), elements in a tuple (“The X coordinate has a zero”), or member variables in a structure or an array (“An address has a zip code”). In other words, “having” verbs describe the same types of things as adjectives, just using different terminology.

Having verbs such as “has a” describe membership

When a “having” verb describes a class, there are five different types of relations. The nature of the relation should be clarified with the client when possible.

Relation	Description
Composition	A member variable exists within a class. The class manages the creation, utilization, and destruction of the variable.
Aggregation	A member variable exists within a class. The class does not create or destroy the variable, however. It just contains a reference to it.
Nested	The data type is only relevant inside the class. Thus, it is defined within the class and no other class can create an object of this type.
Association	A method within a class has a variable of this type. The variable is not a member variable so it is not part of the class’ identity.
Dependency	One class depends on another in a way not described by the previous four in the list. Usually this means a parameter is accepted of that data type.

We will learn more about *has-a* relations in Unit 3.

Be Verbs

Be verbs are usually characterized by the phrase “is a.” They clarify the type or nature of something. For example, a mother is a woman, a woman is a person, and a person is a mammal. In each case, we are specifying a relationship where a class of people (“woman”) inherits the attributes of another class (“person” in this case). When you start looking for these types of relationships, you will quickly discover them everywhere.

Be verbs such as “is a” describe inheritance

Most programming languages can describe “is a” relations through a mechanism called inheritance. We will discuss inheritance in more depth in Unit 3.

Noun Identification Process

The noun identification process is an iterative process involving close collaboration between the client and a developer, often mediated by a facilitator. The client, also known as the subject matter expert, stakeholder, user, domain expert, product sponsor, or product owner, is the individual with domain knowledge about the system being modeled. It is the job of the developer to produce working software that represents what the client knows.

Best Practice 27.12 Speak with more than one client whenever possible

Aside from the simplest problems, it is unusual for a single individual to completely understand the problem domain. Interviewing more than one client allows for different perspectives on the system and more specialized areas of expertise.

The overall process is iterative. The bottom loop (Steps 4 through 6) represents the process of turning a single requirement into design elements using the noun identification process. For it to work, it is necessary to have a collection of high-quality requirements. This is where the top loop comes to play.

The top loop (Steps 1 through 4) represents the process of interviewing the client to generate use cases and turning these into design elements. It is an iterative process because the developer needs to continually go back to the client to clarify ambiguity and validate assumptions. This process repeats until the client can think of no more use cases and everything described has been turned into design entities.

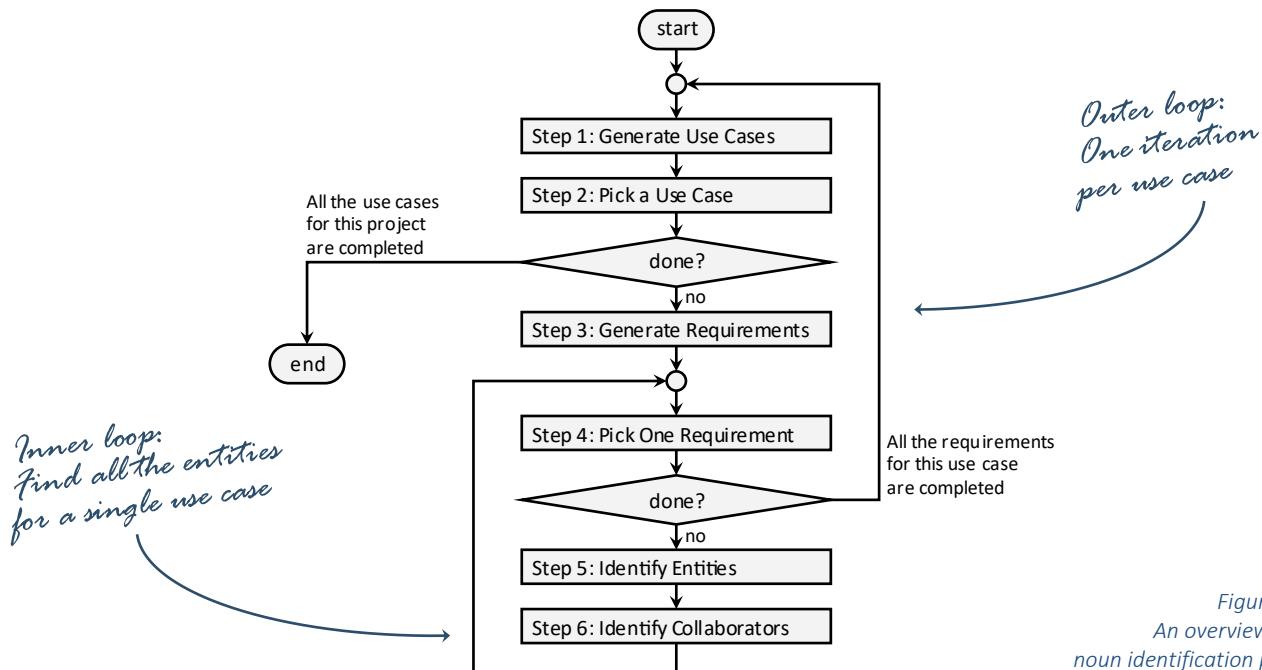


Figure 27.2:
An overview of the
noun identification process

Best Practice 27.13 Adapt, don't adopt

As with all design methodologies, the purpose of the noun identification process is to help the designer channel his or her creativity. It is not meant to be a rigid algorithm that must be fanatically followed. Adapt development methodologies to meet your specific needs; do not adopt those of another.

Step 1: Generate Use Cases

Document the client's needs with whatever mechanism is most convenient

The first step of any software development process is to understand the problem domain. In one form or another, this must involve gathering data from the client.

In rare cases, this can be presented to the development team as a complete and correct formal requirements document. Often, the team only has a client with a rough idea of what is needed. The first step of the noun identification process is to document the client's needs. This is often done with use cases.

A use case is a written description of the system written from the client's perspective. A use case is not meant to be a complete description of the system, but rather a description of one aspect of the system. To see how this works, consider a grocery-list application. The following use cases have been identified:

A use case is a description of the system from the client's perspective

Use Case List

- The user can add, remove, duplicate, or edit a grocery list
- Each grocery list has a user-specified name
- When editing a grocery list, the user can add, edit, and remove items
- An item has a name, location in the store, quantity, and price
- The user can print a grocery list
- The "grocery store view" facilitates checking items off the list

At this phase of the design process, it is understood that the collection of use cases is incomplete. The client and the developer will continue to revisit this first step until all known aspects of the system are identified.

Step 2: Pick a Use Case

With projects of any degree of complexity, it is generally not possible to work on all use cases simultaneously. Step 2 involves picking one from the list and hashing it out before moving on to the others. Of course, if the list is empty, then the design process is finished.

Which one should you start with? At this stage in the design process, it does not matter. There is no obvious way to sort the list of use cases from most important to least, from easiest to hardest, or from most complex to simplest. The only guidance is this: it is generally easiest to select a use case like the one you just completed on the previous iteration. As you work on a single aspect of the system, your knowledge of that area tends to increase. To leverage this recently gained understanding, it makes sense to stick with one aspect of the design.

Step 3: Generate Requirements

The noun identification process requires precise written descriptions of the client's needs. This means that it is probably not good enough to have high-level, domain-specific descriptions gleaned from the client. Instead, we need requirements. According to the IEEE, a requirement is "a condition or capability that must be met or possessed by a system to meet an objective." Requirements must be correct, feasible, necessary, unambiguous, and verifiable to be useful in the software development process.

From the use case "The user can print a grocery list," there are several requirements:

Requirements
The grocery list feature shall have a print command
The print command shall bring up a print dialog
The print dialog shall present the user with opportunity to select a printer
The [cancel] button in the print dialog shall return the user to the previous state
The [print] button shall send the current list to the printer
Printing shall occur in a background thread

Note that these requirements are much more specific than use case from which it came. Where did these extra details come from? In this case, the developer assumed that the client wanted this print feature to work like every other print feature. That may or may not be accurate. It is therefore necessary for the developer to validate these requirements with the client before moving on to the next step in the process.

Best Practice 27.14 Continually validate your requirements list with the client

Use cases from the client are often in vague and general terms. The developer's job is to remove ambiguity at every step in the process, as there is no room for ambiguity in the coding phase. Every requirement should be written with an eye to turning it into code. Therefore, nouns, proper nouns, adjectives, action verbs, having verbs, and be verbs should be used intentionally.

Be constantly vigilant for missing items. If something is implied but not explicitly stated, rewrite the requirement to include the implication. If there is a noun masquerading as a verb or vice versa, rewrite it so the intention is clear.

Best Practice 27.15 Be as specific as possible

When converting a use case into a requirement, be mindful of implied items. Back to our print example, nowhere does it say which grocery list is to be printed. Is it the currently viewed grocery list? What do we do when there is no current list? What if there are multiple lists which are currently being viewed? When the client and/or the developer have a simplistic view of the system, that simplistic view often fails to capture the complexity that the user may require.

Step 4: Pick One Requirement

After Step 3, one use case will often turn into several requirements. The noun identification process works best when we focus on one requirement at a time. As with selecting a use case in Step 2, it does not matter the order in which requirements are completed. It only matters that all the requirements for a given use case are finished before moving on to the next one. Only when the last requirement is done do we move back to Step 1 and revisit our collection of use cases.

Step 5: Identify Entities

Step 5 constitutes the heart of the noun identification process. This is where we map words or phrases in requirements into design entities. It is more than just underlining words and assigning function or class names to them. It is a discovery process involving asking questions, performing research, and drawing diagrams. In other words, identifying the part of speech of a word in a requirement is just the starting point in the design process. It is followed by asking the questions associated with each part of speech and then working through several design alternatives to maximize quality according to applicable metrics.

When it is hard to identify entities, it is probably necessary to refine the requirements

There are times when the entity identification process seems impossible and that no meaningful progress can be made. The cause is almost always the same: poor requirements. In general, noun identification does not work without the groundwork done in Steps 1–4. It is the iterative and systematic refinement of requirements that makes the entity identification process possible.

Entity identification is a discovery process involving asking questions, performing research, and drawing diagrams

Step 6: Identify Collaborators

The final step in the noun identification process is to identify collaborators. Collaborators are program entities that depend on each other. When working on a class, for example, a collaborator could be a function utilizing our class, an object as a member variable within our class, or even another class that contains one of our objects.

The collaborator identification step is important because it ties the various components into a single cohesive system. Without this step, we have a collection of disjoint entities. It is important to frequently take a step back to look at the big picture. Is the design evolving into a unified and integrated system, or is it looking more like ravioli – solid masses loosely connected with a sauce? I love ravioli, but certainly don't want my design to be described as such.

Frequently look at the big picture to look at the overall design quality

Best Practice 27.16 Prefer reusing existing functionality over building new functionality

With every new component added to the design, ask yourself if that functionality already exists in the system. If it does, then refactoring might be necessary.

Refactoring is the process of reworking an existing design to accommodate a new design constraint or requirement

Refactoring is the process of reworking an existing design to accommodate a new constraint. In the context of reuse, refactoring often means extending the functionality of an existing component so it can be used in more than one context. This could be to make it more general and abstract. It could be to

add new functionality that is specific to a given scenario. Whatever the need, existing components should be reused whenever possible.

Variations

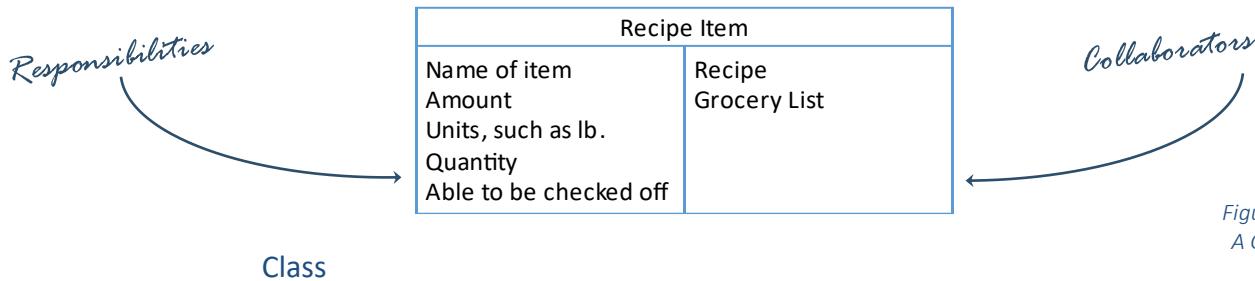
There are several variations of the noun identification process, the most popular of which is Class Responsibility Collaborator (CRC) cards.

CRC Cards

The CRC card process involves documenting candidate classes on index cards and then organizing them according to how they relate to each other. Each card has a title (the class name) and two columns: responsibility and collaborators.

Part	Description
Class	The name of the class we are building
Responsibility	The role of the class in the program
Collaborators	Other classes or components in the system

A typical CRC card might look like the following:



Each CRC card has a title, which the class being modeled. This reflects the very object-oriented bias of the entire CRC process. It is not very helpful for modeling procedural designs or algorithms.

Responsibility

The left column is called “responsibility” and corresponds to the requirements the class is meant to fulfill. This includes notes about what the class represents, the role it will play, and anything the class knows or does.

There is not a general consensus as to what goes in the responsibility column nor how it is formatted. Some list only candidate member variables. Others list potential methods. Others make general notes as to what the class is all about.

Collaborators

Collaborators include any class or function that depends on the class we are working on. It also includes anything that this class depends on. The list does not have to be complete or even accurate. The idea here is to jot down ideas that will later be codified into the final design. When the CRC cards are organized according to how they relate to each other, the collaborators column is a great starting point.

Domain Analysis

Broadly speaking, there are two main ways to look at a software system: from the outside-in or from the inside-out. The outside-in perspective focuses on the behavior and functionality of the system. This is often called the domain model. The inside-out perspective focuses on how the system is built, often called the application model. Traditionally, clients understand the domain model. They know what the software is meant to model and how it should behave. Developers understand the application model, namely how the program is built.

Domain analysis is a development methodology conducted by the developer to better understand the domain model. This can be conducted in an ad hoc way where the client is interviewed and notes are taken. It can also follow a very formal process including generation of documentation representing that which is known about the domain. Regardless of the degree of formality, the following components usually exist:

Component	Purpose
Glossary	Document the concepts and terminology of the problem domain. Some are to be used in the software itself, others are necessary to communicate with the client.
Facts and Processes	Summarize processes, rules, relationships, principles, and anything else the client believes is relevant to the project.
Users	What is known about who will use the system, including their expectations, background, and the types of tasks they will use the system to accomplish.

Formal domain analysis is critical when the client cannot be present during the development process. This is less important when the client is part of the development team and is available to answer questions as they arise.

Agile Integration of Noun Identification

Agile is a class of development methodologies first described in 2001 that has since become the standard for how development teams are organized. One of the Agile values is “working software over comprehensive documentation,” commonly resulting in a lack of a formal requirements document. Instead, the client’s needs are captured in a collection of user stories. How, can noun identification be incorporated into Agile development?

Agile’s user stories are close cousins with use cases. Here, a developer is assigned a user story from the sprint backlog which is then implemented and tested. The only necessary alteration is to work with the noun identification process is to remove the “generate use cases” step (the product owner is in charge of that) and to work with user stories instead of use cases.

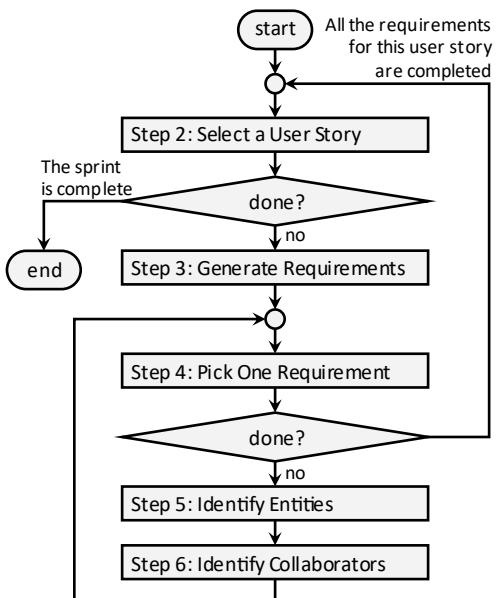


Figure 27.4:
Noun identification with
Agile user stories

Examples

Example 27.1: Requirement Identification

This example will demonstrate how to turn a use case into a set of requirements.

Problem

Consider a mobile application to manage a credit card account.

The “card-tracker” application helps users keep track of their credit card usage so they can avoid late fees and manage their spending. This functionality is available on all the popular mobile platforms where account status and spending habit reports are presented to the user the moment they are making purchase decisions.

From this product description, the following use case has been identified:

The user adds a new purchase.

Create a list of requirement mapping to this use case.

Solution

Drawing from common knowledge about how credit card transactions work, the following requirements were identified:

Initial Draft of the Requirements

A single transaction shall have a date, name, note, category, and amount.
The user shall be able to add a new transaction to the register.
The user shall be able to edit an existing transaction in the register.
The user shall be able to remove a transaction from the register.

These requirements were presented to the client for approval. The client discovered an important detail that was not apparent initially: the source of the transaction. Some transactions are hand-edited by the user, some are downloaded from the bank, and some are captured by taking a picture of a receipt.

Revised Draft of the Requirements

A single transaction shall have a date, status, name, note, category, and amount.
The user shall be able to add a new transaction to the register.
The user shall be able to edit an existing transaction in the register.
The user shall be able to remove a transaction from the register.
The user can download transactions from the bank.
The user can add a transaction by taking a picture of a receipt.
The user can change the status of a transaction in the register.

These new requirements were approved by the client.

Example 27.2: Nouns into Classes

This example will demonstrate how to turn a noun in a requirement into a class diagram.

Problem

Consider the mobile financial application from Example 27.1. Given the “user adds a new purchase” use case, the following requirement has been selected:

Requirement to be Translated

A single transaction shall have a date, status, name, note, category, and amount.

Convert this requirement into program entities.

Solution

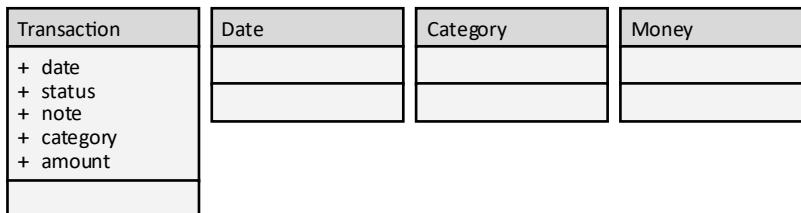
The first step is to identify the parts of speech:

Word or Phrase	Part of Speech	Program Entity
a single transaction	Proper Noun	Variable or object
have	Having Verb	Member variable
date	Common Noun	Data type
status	Common Noun	Data type
note	Common Noun	Data type
category	Common Noun	Data type
amount	Common Noun	Data type

While “a single transaction” clearly refers to one instance of a transaction, this requirement is talking about transactions in general. Thus, the important thing to note is that a transaction is a data type.

The “have” verb indicates that there is more than one aspect to the “transaction” data type, implying it is a collection. Since it will likely be passed as parameters, stored in other data structures, and kept for long periods of time, a more formal construct such as a class is preferred over an informal one such as a tuple or array.

The five common nouns (*date*, *status*, *note*, *category*, and *amount*) are all members of the transaction class. *Date*, *category*, and *amount* are clearly classes because there are verbs associated with them. *Note* appears to be just a text field, so a built-in data type seems appropriate. Finally, *status* seems to be an integer, though we may need to make it a more robust data type in the future.



Example 27.3: Verbs into Functions

This example will demonstrate how to turn a verb in a requirement into a structure chart.

Problem

Consider the mobile financial application from Example 27.1. Given the “user adds a new purchase” use case, the following requirement has been selected:

Requirement to be Translated

The user shall be able to add a new transaction to the register.

Convert this requirement into program entities.

Solution

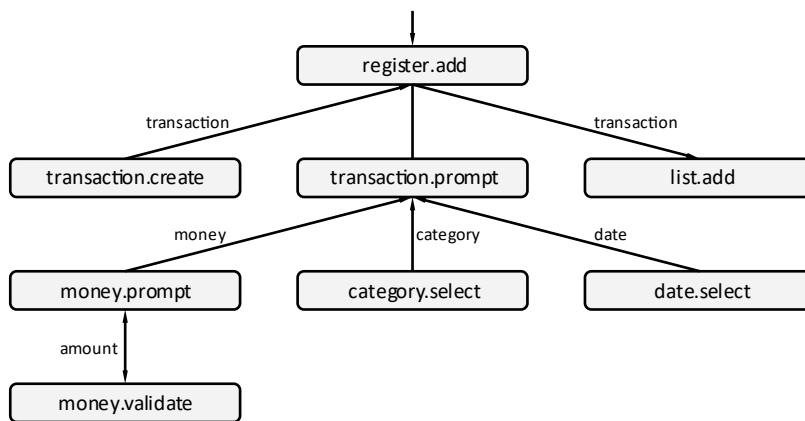
The first step is to identify the parts of speech:

Word or Phrase	Part of Speech	Program Entity
user	Proper Noun	Variable or object
add	Action Verb	Function or method
a new transaction	Proper Noun	Variable or object
the register	Proper Noun	Variable or object

“The user” in this instance refers to an entity that exists outside the program boundary. It refers to external input and output. It may also refer to a UI object if such a thing exists in our system.

“Add” is an action verb, corresponding to a function or method. Which is it? Note that an existing transaction is not being added. Thus, *add* is not a method in the transaction class. However, the transaction is being added to the register. It does make sense to include *add* as a method on the register class.

One final note: there are many subtasks to adding a transaction. The input needs to be gleaned, the transaction needs to be validated, and the new transaction needs to be added to the register. In other words, more than one function or method is needed.



Example 27.4: Proper Noun into Variables

This example will demonstrate how to turn a proper noun in a requirement into a DFD.

Problem

Consider the mobile financial application from Example 27.1. Given the “user adds a new purchase” use case, the following requirement has been selected:

Requirement to be Translated

The user can change the status of a transaction in the register.

Convert this requirement into program entities.

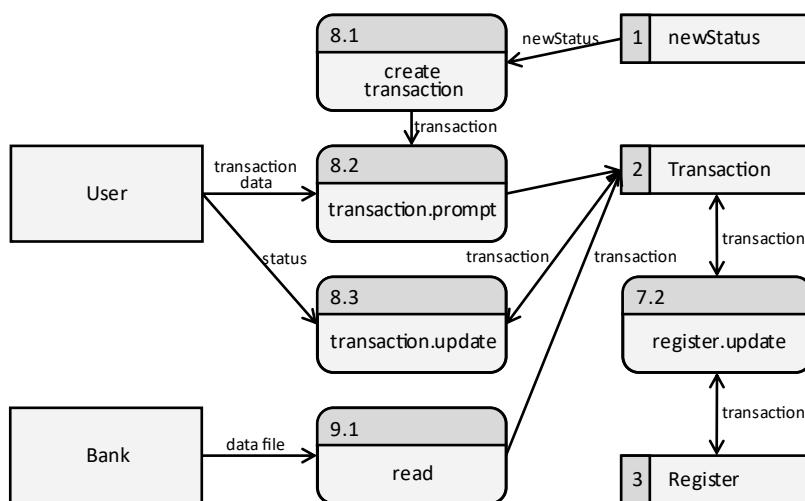
Solution

The first step is to identify the parts of speech:

Word or Phrase	Part of Speech	Program Entity
user	Proper Noun	Variable or object
change	Action Verb	Function or method
the status	Proper Noun	Variable or object
a transaction	Proper Noun	Variable or object
the register	Proper Noun	Variable or object

There are several proper nouns in the requirement that have already been identified: *user*, *status*, *transaction*, and *register*. The unique thing about this requirement is the action verb *change*. Here, the act of changing the value of the member variable *status* is not very interesting. What is interesting is how the *status* value changes through the life of the program. This needs to be better understood and documented.

Since we are discussing the flow of a value through the system, the obvious choice here is a DFD.



Exercises

Exercise 27.1: Parts of Speech

For each part of speech, identify the corresponding programming construct.

Part of Speech	Programming Construct
Action verb	
Proper noun	
Be verb	
Common noun	
Having verb	
Adjective	

Exercise 27.2: Steps in the Process

From memory, name and explain of each step in the noun identification process.

Step	Name and Explanation
Step 1	
Step 2	
Step 3	
Step 4	
Step 5	
Step 6	

Exercise 27.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
All proper nouns must be capitalized	
One requirement can become many entities	
An action verb can be a function or a method	
The order a requirement is selected is important	
Good requirements are needed for the process	
Use cases ultimately come from the client	
One use case can have several requirements	
The order a use case is selected is important	

Problems

Problem 27.1: Use Cases

Consider a grocery list mobile application that has an “in-store” feature:

The “in-store” feature facilitates the user to check items off their grocery list as they put things in their cart. It presents a list of all the times in the list and puts a checkbox next to each item. The user can touch the checkbox with her finger and the item is marked as “done.” Note that the user can sort the list alphabetically, by category (produce, meats, etc.), or by status (done, not in the store, or yet to be retrieved).

Based on this problem description, create 3–5 use cases (Step 1 of the noun identification process).

Problem 27.2: Requirements

Select a use case from Problem 27.1 and generate 3–5 requirements (Steps 2 and 3 of the noun identification process).

For each requirement, make sure to intentionally use common nouns, proper nouns, adjectives, action verbs, having verbs, and be verbs to minimize ambiguity.

Problem 27.3: Identify Entities

Select a requirement from Problem 27.2 and identify the corresponding program entities (Steps 4, 5, and 6).

For each program entity, use the best possible representation (flowchart, pseudocode, structure chart, DFD, or class diagram).

For each design element, determine the quality based on applicable metrics (efficiency, maintainability, cohesion, coupling, fidelity, robustness, convenience, and abstraction).

Problem 27.4: Use Cases

Consider the card game “war”:

“War” is a turn-based card game where the playing cards are evenly divided among the players. When a new turn begins, each player turns over the top card in their stack. The player with the highest value collects all the upturned cards. If there is a tie, then the players add two face-turned cards to the pile and then a third up-turned card. The player with the highest value collects all the upturned cards and the pile. The game continues until one player has all the cards in the deck.

Based on this problem description, create 3–5 use cases (Step 1 of the noun-elicitation process).

Problem 27.5: Requirements

Select a use case from Problem 27.4 and generate 3–5 requirements (Steps 2 and 3 of the noun identification process).

For each requirement, make sure to intentionally use common nouns, proper nouns, adjectives, action verbs, having verbs, and be verbs to minimize ambiguity.

Problem 27.6: Identify Entities

Select a requirement from Problem 27.5 and identify the corresponding program entities (Steps 4, 5, and 6).

For each program entity, use the best possible representation (flowchart, pseudocode, structure chart, DFD, or class diagram).

For each design element, determine the quality based on applicable metrics (efficiency, maintainability, cohesion, coupling, fidelity, robustness, convenience, and abstraction).

Challenges

Challenge 27.1: Space Invaders

Research the 1978 arcade game Space Invaders.

Complete the noun identification process to come up with a complete design for this application. You should have 20–30 requirements, a dozen classes, and a couple dozen functions.

Challenge 27.2: To-Do

Consider the simple to-do lists you may maintain to keep track of upcoming assignments, people to call, things to do, or items to buy. Your job is to write a simple mobile application to maintain these lists for a user.

Complete the noun identification process to come up with a complete design for this application. You should have 20–30 requirements, a dozen classes, and a couple dozen functions.

Challenge 27.2: Budget

Consider an application designed to manage a person’s finances. One feature allows the user to specify a budget. We are tasked with creating a report showing the user how they did on their budget for the past month. This will show the various spending categories, how much they should have spent, and how much they did spend.

Complete the noun identification process to come up with a complete design for this feature. You should have 20–30 requirements, a dozen classes, and a couple dozen functions.

Metaphor

Chapter 28

Metaphors and symbols help convey abstract and difficult-to-understand concepts using familiar and well-understood ideas.

In many ways, programming is the process of representing entities from the physical world with virtual symbols. The integer variable `age` stands in for how long you have lived on the earth. The function `addition` stands in for the concept of combining two things. The class `Transaction` stands in for a single credit card transaction. It is the programmer's job to find appropriate symbols and metaphors to represent design concerns.

Metaphors and symbols are tools to help communicate abstract ideas

A programmer's choice of metaphors can have far-reaching influence on the understandability of the resulting code. They can drastically increase convenience and help with abstraction goals. If they are well-chosen, then adjectives like "intuitive" and "obvious" are often applied. Poorly chosen metaphors make code difficult and worse yet misleading. The challenge to picking good metaphors is to select from a well-understood menu, to use them consistently, and to make sure that they are sufficiently powerful to capture what they are designed to represent.

All abstractions leak. This is a well-intended deception

Common Operations

If you do the same thing twice, use the same name

A surprising amount of functionality can be represented with a small number of common operations. Software can be more understandable if it uses a consistent naming convention. In other words, synonyms are wonderful literary tools, but they have no place in software. Ward Cunningham, an important contributor to wikis, extreme programming, and design patterns, perhaps put it best:

You know you are working on clean code when each routine you read turns out to be pretty much what you expected.

The following is a (hopelessly incomplete) collection of common operations:

Use	Operation
Change or retrieve an attribute	Set & Get
Work with collections	Insert, Remove, & Update
Interact with users	Display, Input, & Prompt
Respond to a specific event	On
Indicate readiness to perform an action	Can, Is, & Has

Attributes

When an attribute is stored in a class, two things are needed: to set the value and to retrieve it. These actions typically come in pairs; if one is needed, so is the other.

The “set” verb is applied to any action that updates a value in a data store. The set metaphor replaced an old existing value with a new one. Though it is used in a variety of situations, set is most commonly used in class methods. In fact, such methods are often called “setters.”

Best Practice 28.1 Use the prefix “set” for any method that updates a class’ attribute(s)

Setters, otherwise known as mutators, must take one or more parameters from the client and must write to at least one member variable. If a class has only one setter or if the intent is unambiguous, then the single word “`set()`” is appropriate for the method name. However, if more than one setter is needed, then “set” is commonly used as the prefix to the full method name: `setRow()`.

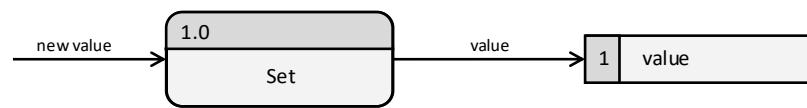


Figure 28.1:
DFD of a set method

“Get” is a verb applied to any action that retrieves a value from a data store. The retrieved value is a copy of the one fetched, leaving the original unchanged. As with set, get is most commonly used in class methods. Therefore, methods returning class attributes are often called “getters.”

Best Practice 28.2 Use the prefix “get” for any method that retrieves an attribute from a class

Getters, also known as accessors, usually take no parameters; the nature of the returned value is implied by the name of the method and by the class it is attached to. Of course, getters have a single return value: the data that the client requested.

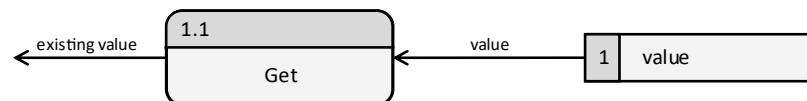


Figure 28.2:
DFD of a get method

Getters are setters usually come in pairs; where you find one you should find another. They should also be symmetric: the input parameter of the setter should be the same as the return parameter of the getter.

Getters and setters have a bad reputation with OO designers

Getters and setters often have a bad reputation among object-oriented designers. This is because novice programmers tend to create simplistic getters and setters that only move data between the client and the

member variables. These getters and setters do not add any value: they do not shield the client from the complexity of the class (abstraction), do not perform data validation (robustness), and do not translate the data into the format the client needs (convenience). There is nothing wrong with getters and setters per se, but they should be designed with all the encapsulation quality metrics in mind.

Best Practice 28.3 Input and output parameters should be in a format convenient to the client

To maximize the metric of convenience, input parameters from getters should be in a format that is fitting and appropriate for the client. Remember, it is the class’ job to serve the application, not the other way around.

Collections

Just about every class working with collections needs to provide the client with the ability to insert, remove, and update elements. If one action is present, then all three are. It is often the case that an entire suite of operations is needed:

Operation	Description
Insert	Add a new element into a collection
Remove	Remove an existing item from a collection
Get	Retrieve a value from a collection
Update	Change an element that exists in a collection
Find	Locate an element that is within a collection
Size	Determine the number of elements in a collection
Iterate	Visit each element in a collection

The first and most common verb applied to collections is *insert*.

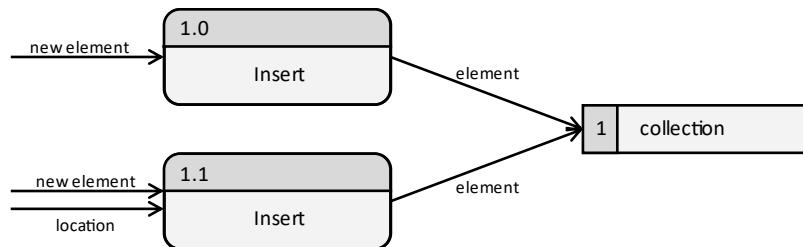


Figure 28.3:
DFD of an insert method

The *insert* verb is applied to any action that puts something into a collection. There must always be at least one input parameter: the element to be added into the collection. In the case of ordered collections such as arrays, a second parameter may be needed: the location or position of the new element in the collection. Usually there is no return value to insert methods, but it is not uncommon to return a reference to the newly inserted element or the status of whether the insertion was successful. The result of the insertion operation is that the collection is the same as before (in the case when the insertion failed or the element already exists) or larger by one. Insert should never decrease the size of the collection.



Figure 28.4:
DFD of a remove method

Remove is the opposite of insert; it takes an element out of a collection. Thus, the collection must be the same size (if the element already does not exist) or smaller by one. Note in the above DFD that there is no arrow between the remove function and the collection. An arrow does not leave the collection because the removed element is not sent back to the client. An arrow does not enter the collection because the location of the element being removed is not added to the collection. DFDs represent the flow of data, not the transfer of control as flowcharts and structure charts do.

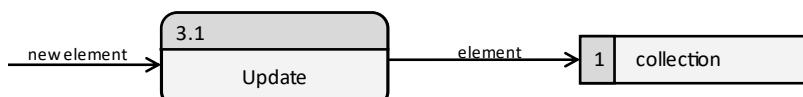


Figure 28.5:
DFD of an update operations

Update modifies a single element in the collection. It leaves the size of the collection unchanged, but one element in the collection can have a replacement value. This could mean the element within the collection is swapped out for a new instance or the value of an existing element is updated.

Interfaces

Unlike accessing attributes (using get and set) or elements in a collection (insert, remove, and update), user interface actions do not necessarily need to appear in complete sets: the presence of a display method does not imply the need of a corresponding input method. Though all interactions with users can be characterized as input, output, or a combination of the two, there are usually five different operations used:

Operation	Description
Display	Send data to the user
Input	Receive input from the user
Prompt	Display paired with an input
Interact	Multiple I/O interactions, typically in a session
Wait	Pause until a fixed amount of time has passed

Of the five operations, only four involve data flowing to or from the user.

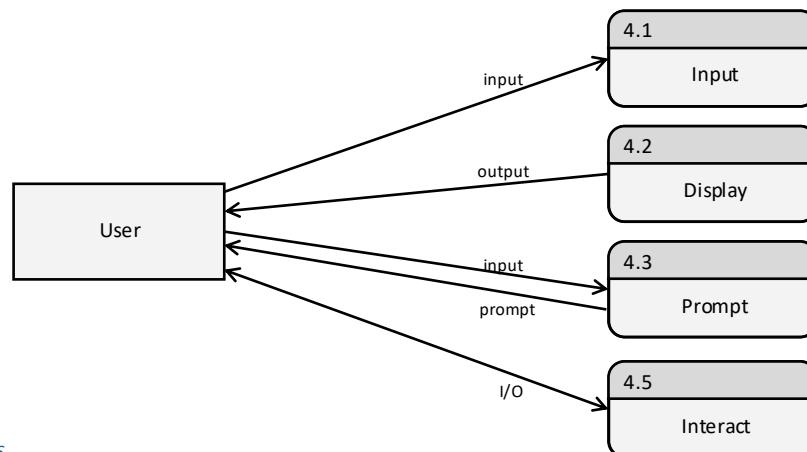


Figure 28.6:
DFD of interface operations

The *input* verb refers to any type of input, be that from a keyboard, finger, or mouse. It is commonly used for on-demand input: the program needs input from the user at this point in time and will wait until it is received. When the system is relying on demand-based input, then the prefix “on” is used. See the following page for details.

The *display* verb is applied to any action sending data to a user. This could be a dialog, a report, a screen, or even just a textual message. In each case, the data flow is one direction. Note that “draw” is also used when the output is graphics heavy.

The *prompt* verb implies both an input and a display. A dialog box would be a typical prompt. It is not uncommon for a prompt function to call a display function and an input function to complete its task.

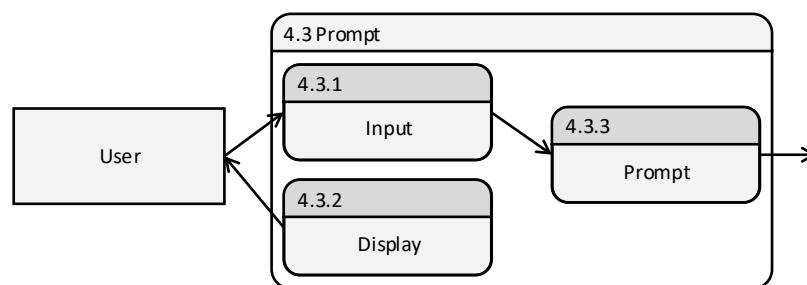


Figure 28.7:
DFD of prompt

Events

The *on* verb is used for functions that respond to specific events. Usually, the *on* function is a callback: a function provided by a caller for a callee to execute at a pre-specified event.

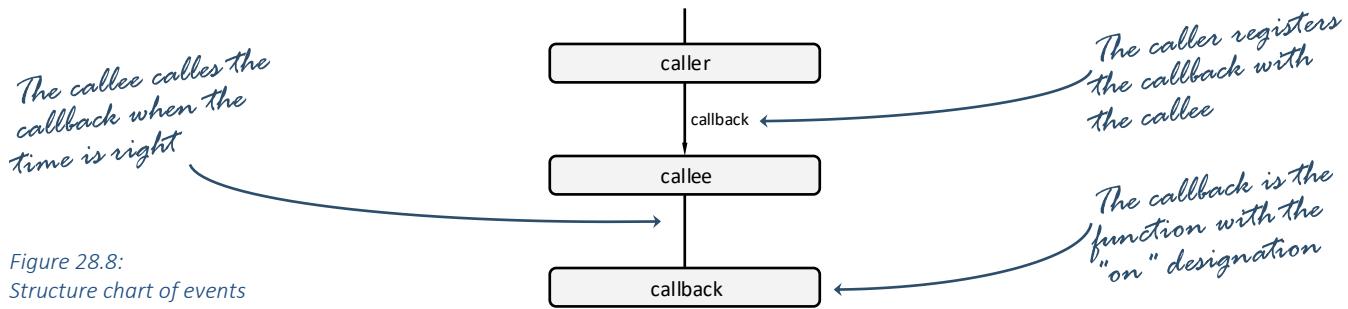


Figure 28.8:
Structure chart of events
implemented as callbacks

Perhaps this is best explained by analogy. Consider a general planning for a battle with the enemy. Up on the front lines, he anticipates that the enemy will try to attack. If this happens, he wants his army to respond a certain way. To deal with this contingency, he puts some orders in a sealed envelope for one of his lieutenants. If the lieutenant observes the enemy trying the attack, the lieutenant is to respond with the instructions in the sealed envelope. In this analogy, the general is the caller, the function initiating the exchange and providing the instructions. The lieutenant is the callee, the function receiving the instructions and the function to carry out the instructions at a pre-determined time. The instructions in the sealed envelope are the callback. These are instructions provided by the caller to be executed by the callee. While this analogy with the general, the lieutenant, and the sealed envelope may seem contrived, it is actually quite common in practice.

Consider a graphics library (such as OpenGL) enabling a program to display images on the screen and to receive keyboard events. If such a keyboard event were to occur, how would the graphics library notify the client program? The most convenient way for this to happen is for the graphics library (lieutenant or the callee) to call a function (sealed envelope or a callback) specified by the client (general or the caller). This way the program can handle the event as needed.

Best Practice 28.4 The name of the callback should reflect the type of event it is responding to

The *on* verb is a prefix; there needs to be a noun that follows describing the type of event it is respond to. One of the best examples of this metaphor is JavaScript Document Object Model (DOM) events. The callback `onchange()` is called when the associated input control has received user input. The callback `onmouseover()` is called when the user's mouse has moved over the associated control. `onclick()`, of course, is called when the user has clicked on a control.

Predicates

A function that returns a Boolean value is called a predicate. Other names include propositional functions, indicator functions, or even Boolean functions. In each case, they refer to a function that is useful in making a decision or declaring status.

Best Practice 28.5 The name of a predicates must indicate what TRUE and FALSE means

When a predicate function returns TRUE, that value must mean something. The name of the function needs to unambiguously describe what that means. For example, consider a function determining whether an individual qualifies for a child tax credit:

Pseudocode

```
childTaxCredit(income, age)
IF income < $200,000 AND age < 17
    RETURN true
ELSE
    RETURN false
```

Figure 28.9:
Predicate function with an ambiguous name

This is a predicate function: it returns TRUE or FALSE depending on if the criteria is met. However, the name of the function does not unambiguously tell us how to interpret the output. If the return value is TRUE, then we are forced to conclude that the child tax credit is a true principle. What does this even mean? It would be better to choose a name that better reflects the predicate:

Pseudocode

```
isQualifiedForChildTaxCredit(income, age)
IF income < $200,000 AND age < 17
    RETURN true
ELSE
    RETURN false
```

Figure 28.10:
Predicate function with an unambiguous name

If this function returns TRUE, then clearly the individual is qualified for a child tax credit. Now the name corresponds to what the return value means.

There are several names associated with predicate functions:

Operation	Description
Is	Whether something is true or not
Can	Readiness to perform an action
Has	Whether an item is contained therein

Operator Overloading

Operator overloading is the process of using a more convenient and human-readable notation for calling a function. Most programming languages use prefix notation. This means that the operator comes first, followed by the operands. An example of prefix notation is “add four and three” in English and “`add(4, 3)`” in a programming language. In both cases, the operator “add” precedes the operands (four and three).

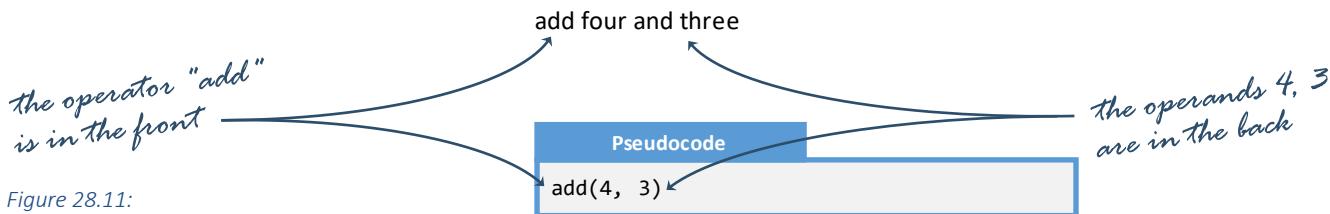


Figure 28.11:
Prefix operator

Infix notation, on the other hand, is characterized by situating the operator between the operands. Examples of infix notation would be “four plus three” or “ $4 + 3$ ”.

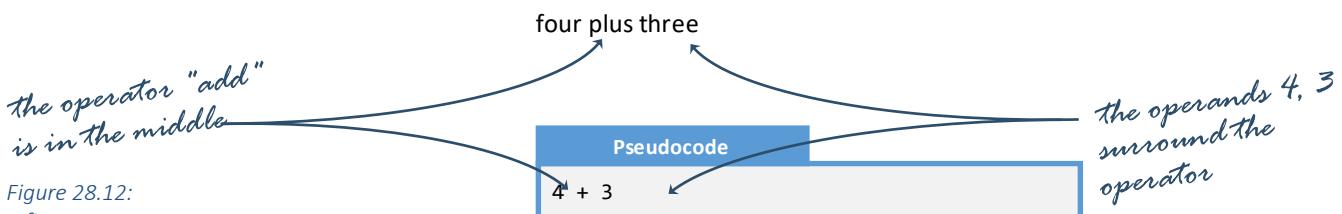


Figure 28.12:
Infix operator

The functionality of either notation is the same; the difference is only cosmetic. There is nothing you can do with infix notation that you cannot do with prefix notation. While most people find infix notation easier to understand than prefix notation, it is not without limitations. Prefix notation allows for an unlimited number of parameters whereas infix notation allows for only two. For example, “`add(4, 3, 7)`” uses only one operator while “ $4 + 3 + 7$ ” utilizes two. Similarly, there is no ambiguity in the order in which functions are called with prefix notation whereas infix notation introduces ambiguity. Does “ $4 + 3 + 7$ ” mean “ $(4 + 3) + 7$ ” or “ $4 + (3 + 7)$ ”? We need parentheses and the order-of-operations to disambiguate how to evaluate expressions in infix notation, but neither are required for prefix.

Operator overloading is the process of using common symbols such as addition (+) and assignment (=) to perform actions that are meaningful in the context of a class. Most operators use infix notation whereas traditional functions use prefix notation. When working with operator overloading, there are usually three components:

Component	Description
Operator	The symbol that stands for the function name
Right-hand side	The operand to the right of the operator
Left-hand side	The operand to the left of the operator

The goal is to make it easier and more convenient for you and others to use your code. Problem: operator overloading, used incorrectly, can result in cryptic code that is more difficult (not less) to understand. Because it has been so often abused, some seasoned developers say: “If you have a problem and choose to use operator overloading to solve it, you now have two problems.” In general, only use operator overloading when the metaphor is unambiguous.

Assignment

Conceptually, the assignment operator works the same as the copy constructor with the exception that a new object is not created. Instead, a copy of the object on the left-hand side is made to the object on the right-hand side.

For example, consider a simple class that represents a complex number (consisting of the real and imaginary parts):

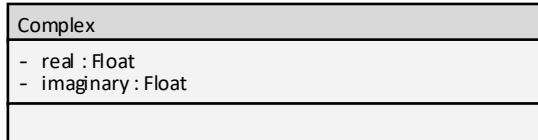


Figure 28.13:
Class diagram representing
complex numbers

If we wish to define the assignment operator for this class, we would do so just like an ordinary method except with the slightly strange `operator=` name.

```
C++  
Complex & Complex :: operator= (const Complex & rhs)  
{  
    this->real      = rhs.real;  
    this->imaginary = rhs.imaginary;  
    return *this;  
}
```

Figure 28.14:
The assignment operator

Best Practice 28.6 Use the assignment operator whenever an object can be copied

Each instance of a copy constructor should be paired with an instance of the assignment operator. Whenever the client may wish to create a copy of an object, provide an explicit assignment operator even when the default one created by the compiler does what you expect. This makes your code more explicit.

There are several variations to the assignment operator. While there might be a useful metaphor for your application, some (such as `%=`) only make sense in a mathematical context.

Operator	Metaphor
<code>+=</code>	Add onto or append
<code>-=</code>	Subtract from, remove, or reduce
<code>*=</code>	Multiply by or extend

Best Practice 28.7 Make sure that `x += y` works the same as `x = x + y`

When using a compound metaphor such as `+=`, make sure it works exactly the same as its components. This means that `x += y` should be the same as `x = x + y`. In languages like Kotlin, you can only name one. If addition (`+`) is defined, then you cannot define add-onto (`+=`).

Comparison

The equivalence and not-equals operators should be defined every time objects need to be compared. Neither changes the left-hand side nor the right-hand side.

C++

```
bool Complex :: operator==(const Complex & rhs) const
{
    return (rhs.real == this->real &&
            rhs.imaginary == this->imaginary);
}
bool Complex :: operator!=(const Complex & rhs) const
{
    return !(*this == rhs); // define != in terms of ==
}
```

Figure 28.15:
The comparison operators

It is far less common to need the relative comparison operators ($>$, \geq , $<$, \leq). These should only be used when there is a well-understood scalar relation between the items. It would make sense with currency, for example, but not with checking accounts (unless the accounts need to be sorted by name).

With complex numbers, it is unclear whether the $>$ operator makes sense. What is bigger, after all: $(0 + 10i)$ or $(10 + 0i)$? After performing a bit of research, it is discovered that the distance from the origin is the main indication of how complex numbers should be sorted. This yields the following definition:

C++

```
bool Complex :: operator>(const Complex & rhs) const
{
    float lhsDistance = this->real * this->real +
                        this->imaginary * this->imaginary;
    float rhsDistance = rhs.real * rhs.real +
                        rhs.imaginary * rhs.imaginary;
    return lhsDistance > rhsDistance;
}
```

Figure 28.16:
The greater-than operators

Best Practice 28.8 If you define greater-than, then you should define less-than

C++

```
bool Complex :: operator<(const Complex & rhs) const
{
    return rhs > *this; // a < b is b > a
}
bool Complex :: operator>=(const Complex & rhs) const
{
    return *this > rhs || *this == rhs; // a >= b is
                                         // a > b || a == b
}
bool Complex :: operator<=(const Complex & rhs) const
{
    return *this < rhs || *this == rhs; // a <= b is
                                         // a < b || a == b
}
```

Notice how these operators are defined in terms of
 $>$ and $=$

Figure 28.17:
The other operators defined
in terms of $>$ and $=$

Best Practice 28.9 Define one comparison operator in terms of another whenever possible

Less code and less redundancy result from defining one operator in terms of another. This should be done whenever possible.

Access []

The access operator, also known as the square-bracket operator, is a useful metaphor for retrieving an element out of a collection. This normally denotes read-write access (similar to “update”), but can also be used for read-only (“get”), and write-only (“insert”). In most cases, the words *update*, *get*, and *insert* are more appropriate than the access operator, but there are many situations when the client thinks of the collection as an array.

```
C++  
Transaction & Transactions :: operator[](int i) throw (char *)  
{  
    if (0 <= i && i < collection.size())  
        return collection[i];  
    else  
        throw "ERROR: Invalid index";  
}
```

Figure 28.18:
The access operator

In the above example, it is easy for the client to think of the collection of transactions as an array. Thus, the array metaphor through the `[]` operator is appropriate. In this case, the client can “get” the data from a transaction and “update” the data in a transaction in the collection because this function is return by-reference. This implementation does not offer the ability to “insert.”

Best Practice 28.10 Use the access operator only when the function is $O(\log n)$ or better

Most programmers know that referencing an element out of an array is $O(1)$. If the access operator is $O(n)$ or worse, then this would be an inappropriate metaphor application. There are many cases when the underlying data structure yields $O(\log n)$ access time. These are mostly appropriate for the access operator because $O(\log n)$ approaches $O(1)$ for very large n . In all other situations, a function name should be chosen that implies the amount of work required to access.

For example, consider a linked list:

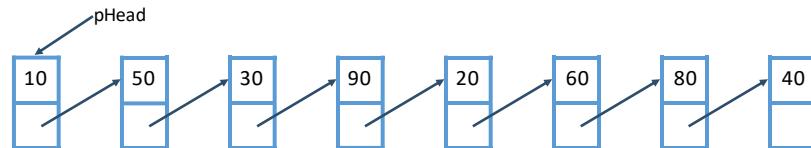


Figure 28.19:
A linked list of integers

If we were to access the i^{th} element from this list, we need to iterate through the previous $i - 1$ elements.

```
C++  
int List :: operator[](int i)  
{  
    int * p;  
    for (p = pHead; p && --i; p = p->pNext)  
        ;  
    return (p != NULL) ? p->data : 0;  
}
```

Figure 28.20:
An inappropriate use of
the access operator
because it is $O(n)$

This function is $O(n)$. Though the client may wish to have this functionality, a more appropriate function name should be chosen, such as “`getElement()`”.

Arithmetic

The arithmetic operators include addition, subtraction, multiplication, and division. It may also include integer division, modulus, exponent, increment, decrement, not, logical and (comparing two Boolean expressions), logical or, exclusive or (returning true if and only if one of the two Boolean expressions is true), bitwise-or (treating variables as arrays of bits and comparing each bit in turn), bitwise-and, and bitwise-not. In most cases, these operators only make sense with a small number of classes.

Operator	Metaphor
+	Add, append, concatenate, or combine
++	Add one, advance, or promote
*	Multiply or add multiple times
-	Subtract, find the difference, subtract from, or remove
--	Subtract one, back up, slow down, or demote
/	Divide, break up, or repeated subtractions
!	Not or opposite
&&	Logical AND, or set-wise intersection
	Logical OR, or set-wise union

The “add” metaphor not only means addition, it also means “add onto.” Thus, it is common to see “+” being used to append, concatenate, and combine. In many programming languages the string “operator” + “ overloading” yields “operator overloading”.

The “increment” metaphor not only means “add one,” but also “advance.” Thus, if we had a class referring to a rank in the military, `rank++` would mean a promotion has occurred.

The “multiply” metaphor not only means “multiplication,” but also means “add multiple times.” Thus, “crazy” * 3 yields “crazy crazy crazy”.

Best Practice 28.11 Avoid arithmetic operator overloading unless the metaphor is unambiguous and clear

Because the arithmetic operators can mean different things to different people, it is generally a good idea to avoid them. The exception is when working with a class that represents a number: currency, measurements, distance, and things like that. Here, there is little opportunity for confusion.

Examples

Example 28.1: Chess Position

This example will demonstrate how to use the access metaphor.

Problem

Consider a class that represents a chessboard. Focus on a method to return the piece in a given position. Use appropriate metaphors to make the class seamlessly convenient and completely abstract.

C++

```
Piece & Board::findPiece(int row, int col)
{
    if (row >= 0 && row <= 7 &&
        col >= 0 && col <= 7)
        return board[row][col];
    else
        throw "Invalid position";
}
```

Solution

Our first draft of this method is to call the method `board.findPiece(row, col)`. The first issue is that we are not using a standard name. A more common metaphor would be Get because we are retrieving the value held in a member variable.

C++

```
Piece & Board::get(int row, int col);
```

Note how this design both reveals an implementation detail about the `Board` class (namely that it is an 8x8 array where the indices are 0–7) and it requires the `Board` class to validate a position. We can do better by introducing a `Position` class.

C++

```
Piece & Board::get(const Position & pos);
```

A final iteration would be to use the square-bracket operator taking a position as a parameter. This metaphor is a perfect fit for retrieving a piece from a chessboard.

C++

```
Piece & Board::operator[](const Position & pos)
{
    return this->board[pos.getRow()][pos.getCol()];
}
```

Example 28.2: Add a Transaction

This example will demonstrate how to use the add-onto operator.

Problem

Consider a class that represents a checking account register. Focus on a method to add a new account to the register. Use appropriate metaphors to make the class seamlessly convenient and completely abstract.

C++

```
void Register::addNewTransaction(const Transaction & rhs);
```

Solution

The first step is to find a metaphor consistent with adding an element onto the end of a collection. Here, the insert operation appears to be applicable:

C++

```
void Register::insert(const Transaction & rhs);
```

Discuss the level of quality here. Notice here that we are always adding onto the end. This means the add-onto operator might be appropriate. This is especially common because `+=` is often used to concatenate strings in much the same way.

C++

```
void Register::operator+=(const Transaction & rhs);
```

The final implementation of this code is the following:

C++

```
void Register::operator+=(const Transaction & rhs)
{
    this->list.push_back(rhs);
}
```

Example 28.3: Direction

This example will demonstrate how to use metaphors to define a class.

Problem

Consider the following problem definition:

A physics simulator has the notion of a direction. The client may use this class with degrees (0° – 360°) or radians (0 – 2π). Note that degrees wrap. 0° is the same thing as 360° . The client will mostly want to store and retrieve degrees but will also want to rotate by a set amount. In other words, if the current direction is 180° and the client wishes to turn right by 90° , then the new angle will be 270° .

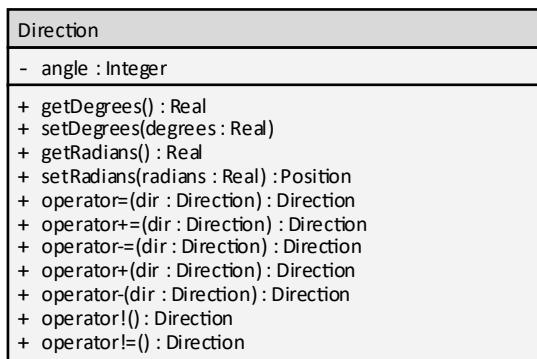
Solution

First, notice that this class represents one attribute. The client will want to both store and retrieve this attribute, meaning that both a setter and a getter are needed. However, since both degrees and radians will need to be supported, we will need two setters (`setDegrees()` and `setRadians()`) and two getters (`getDegrees()` and `getRadians()`).

This class does not represent a collection, an interface, handle events, or return any predicates. These metaphors do not apply. There are several operators which may come in handy. These include:

Operator	Metaphor
<code>+=</code>	Add an angle onto the current direction
<code>-=</code>	Subtract an angle from our current direction
<code>!</code>	Flip the angle around by 180°
<code>=</code>	Copy a direction

Based on these observations, the following class diagram is described.



Exercises

Exercise 28.1: Metaphors

From memory, explain the context when each of the following metaphors would help with code clarity and convenience.

Metaphor	Use
Insert	
Assignment	
Get	
Add	
Input	
Prompt	
Set	
Display	
Remove	
Equivalence	
On	
Add Onto	
Is	
Access	
Can	
Increment	

Exercise 28.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
It is possible to have a perfect abstraction	
Using standard metaphors makes code more understandable	
Operator overloading always increases code clarity	
If you do the same thing twice, it is best to use the same name	

Problems

Problem 28.1: Position

Consider the following problem definition:

An orbital simulator has a notion of a position. This represents the two-dimensional location of an orbital body in the simulation. Note that you can represent the position in meters or in pixels.

Using appropriate metaphors, create a class diagram describing a class to satisfy this problem.

Problem 28.2: Orbital Elements

Consider the following problem definition:

An orbital simulator keeps track of many elements, each of which responds to the force of gravity. This collection is reduced as elements fall to earth or increased when new items are added to the simulation.

Using appropriate metaphors, create a class diagram describing a class to satisfy this problem.

Problem 28.3: User Interface

Consider the following problem definition:

An orbital simulator allows the user to control various elements in the simulation. A satellite, for example, can thrust to a higher orbit. Also, the user can add new elements to the simulation. All the while, the simulator displays the current status to the user.

Using appropriate metaphors, create a class diagram describing a class to satisfy this problem.

Problem 28.4: Velocity

Consider the following problem definition:

An orbital simulator has a notion of velocity. A velocity is the speed and direction (from Example 28.3) of movement. It is possible to increase velocity by a fixed amount (such as add 5 m/s to the speed), increase velocity by a factor (such as double the speed), reverse velocity (move the opposite direction), and copy a velocity object. It is also possible to combine two velocities, equivalent to a nudge given to a marble rolling down a slope.

Using appropriate metaphors, create a class diagram describing a class to satisfy this problem.

Problem 28.5: Element

Consider the following problem definition:

An orbital simulator models the interactions of various orbital elements. These elements are part of a larger collection of elements (described in Problem 28.2). Each individual element has a position (Problem 28.1), velocity (Problem 28.4), and has a variety of status elements (this includes whether it is dead and needs to be removed from the simulation, where it can accept user input, and whether it possesses subelements), and can handle a collection of events (when two elements collide, when it is time to draw, when it is time to move). Using appropriate metaphors, describe a function to handle such events.

Using appropriate metaphors, create a class diagram describing a class to satisfy this problem.

Challenges

Challenge 28.1: HTML Escape

Consider the following problem definition:

HTML treats some symbols as special, most notably less-than, greater-than, and ampersand. You would like to create a class to represent text. It can be displayed either as plaintext or as HTML-encoded text. The client will be able to treat this as a normal built-in string class but will always output encoded text.

Using appropriate metaphors, please do the following:

1. Identify the interfaces that would be useful for such a class.
2. Create a class diagram for this class that uses appropriate metaphors.
3. Implement this class in a programming language of your choice.
4. Create unit tests to validate that your class works as expected.

Challenge 28.2: Weights

Consider the following problem definition:

A weights class can handle a variety of units of weight (gram, pound, kilogram, stone, ounce, ton, etc.). This class will be used in a recipe program that needs the ability to easily combine or scale recipes.

Using appropriate metaphors, please do the following:

1. Identify the interfaces that would be useful for such a class.
2. Create a class diagram for this class that uses appropriate metaphors.
3. Implement this class in a programming language of your choice.
4. Create unit tests to validate that your class works as expected.

Data protection is a strategy to maximize robustness by offering assurances that an object is always in a valid state

In the early days of computers, hardware was so constrained that performance was an overriding design consideration. With the advent of more powerful computers, software was able to model more complex things and fulfill a greater number of applications. Computer languages followed this trend. The first generation of languages were low-level, allowing the programmer to specify even the smallest details. Modern high-level languages simplify many common tasks and are designed to make it more difficult to make typical mistakes. This was essential: more powerful languages and more powerful computers yielded vastly more complex software. Today, modern languages enable class designers to make robustness assurances that were simply not possible during the early days of programming languages.

Data protection provides object robustness assurances

Data protection is a collection of robustness techniques where the design of a class offers assurances that the data contained therein are always in a valid state. This alleviates the client's need to doubt the state of an object or to provide validation checks.

Castle Analogy

Perhaps the most instructive data protection metaphor is a castle. All interactions with the assets contained therein occur through carefully controlled checkpoints. Thus, if visitors wish to speak with the king, they must enter through one of the checkpoints. Sometimes they are carefully escorted to the king, other times they relay their message to a courtier or equerry. Visitors and messages from the king also leave through carefully controlled checkpoints. Guards here make sure nothing of value leaves unintentionally (such as a thief with the king's jewels) and that only the king's intended messages are broadcast to the world.

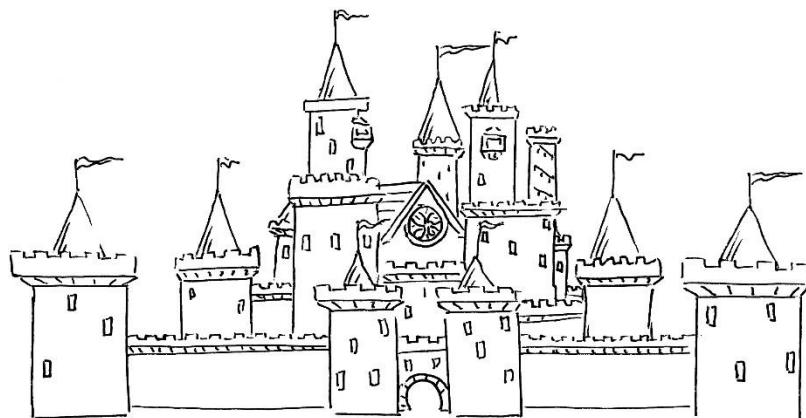


Figure 29.1:
A castle with walls
protecting the assets

In order to ensure that the data within a class are always in a valid state, it is necessary to track all the inflow into the class and all the outflow from the class. A natural tool for this is a DFD. Note that we will add an additional symbol to this graph: a trust boundary. This indicates the boundary of the class. Processors on or within this trust boundary are methods and have direct access to the member variables. In many ways, this trust boundary is similar to the wall in our castle analogy.

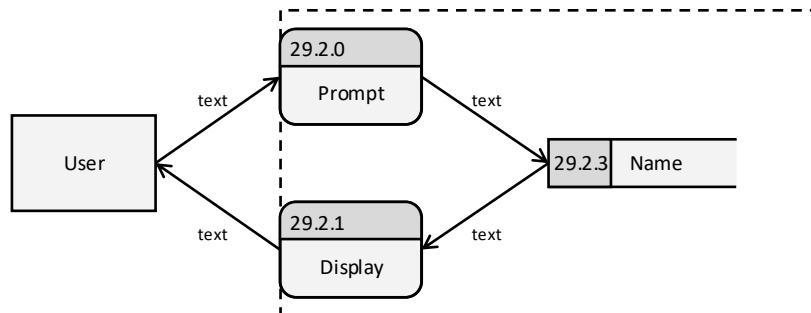


Figure 29.2:
DFD with a trust boundary

In this DFD, we can see that there is only one method which sets the class' data and there is only one method which retrieves it. Thus, there are only two checkpoints in this castle: an input checkpoint called `prompt()` and an output checkpoint called `display()`.

In a more general sense, there are several components in a data protection scenario.

Component	Description
Object	The class whose data we are striving to protect
Client	Anything outside the class interacting with the class
Accessors	Any method facilitating retrieval of the class' data
Mutators	Any method facilitating modification of the class' data
Publics	Any method directly accessible to the client
Privates	Any method with no direct access to the client

To see how these various components interact, consider the following DFD.

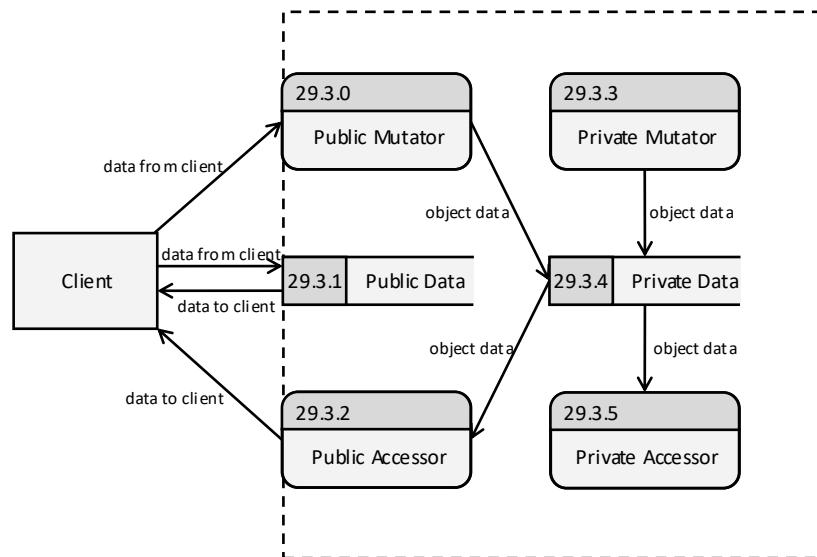


Figure 29.3:
DFD of public and private
accessors and mutators

In the above DFD, notice that the client has direct access to the public accessor, the public mutator, and the public data. In other words, there are exactly three elements on the wall of the castle: two checkpoints and one asset.

Types of Public Interfaces

Just as a castle is only as strong as its weakest point, each public method needs to do its part to help provide data protection assurances. There are several types of public methods which have a special role in this process:

Type	Use
Default Constructor	A method called on object creation when no parameters are provided
Copy Constructor	A method called on object creation when the client input is of the same type as the object
Move Constructor	A form of copy constructor that modifies the client's input by stealing its member variables
Conversion Constructor	Any constructor that is not of the default, copy, or move variety
Destructor	A method called when an object falls out of scope or is otherwise destroyed
Public Mutators	Any public method directly or indirectly allowing a member variable to change
Public Accessors	Any public method eliciting data from the class that the client needs to be in a valid state

These seven types of interfaces can be found in a class diagram, though the various flavors of the constructors can be difficult to detect.

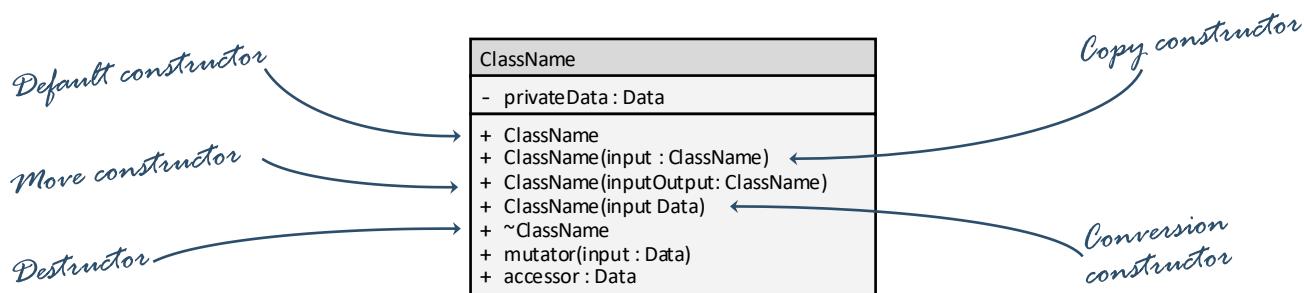


Figure 29.4:
Class diagrams of
public interfaces

In order to provide data protection assurances, each constructor needs to properly initialize the member variables, each mutator and non-default constructor needs to properly validate the data, each accessor needs to not change state and to correctly compute client data, and each member variable needs to be private.

Default Constructor

Initialization is the process of creating an object. In low-level or primitive languages, initialization simply results in allocation of memory to which the new variable is assigned. This amounts to a future homeowner purchasing a parcel of land. While the land must be cleared and leveled before use, it is probably not in this state when the property is created. The first order of business is to haul away that which was left by the previous occupant. The same is true when declaring a variable in C, C++, Fortran, and Cobol. The programmer must explicitly set each variable to a value or the default value will be whatever collection of 1s and 0s resided in memory before the variable was declared.

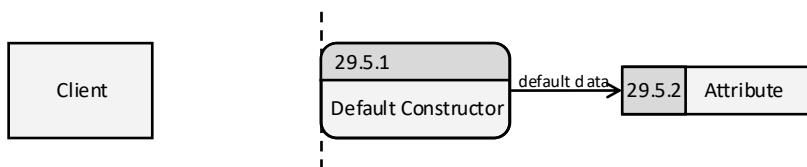
Most modern programming languages automatically set all variables to a default value upon instantiation. Integers are typically set to 0, Booleans are set to false, floats are set to 0.0, and strings are set to the empty string. When creating our own data types (otherwise known as defining a class), it is necessary to figure out an appropriate value for all the member variables. This is typically done through a constructor.

A constructor is a method that is guaranteed to be called when an object is instantiated. In several languages, this special method is the name of the class itself (such as C++, Java, and C#), the name “constructor” (JavaScript, Kotlin, PHP, and Object Pascal), the keyword “new” (Python, VB, and F#), or some variant of “initialize” (Swift, Ruby, Kotlin, Python, and OCaml).

Best Practice 29.1 Always provide a constructor for a class

If your programming language has a constructor (as virtually all do), then every single class definition should include one. This is true regardless of whether the compiler will create one for you if it is not done explicitly. In those cases where the default values for the member variables are uninitialized or not what is needed, then a constructor is absolutely required. In the other cases, it adds clarity and makes the programmer’s intent clearer. A default constructor is a special constructor that takes no parameters.

Figure 29.5:
DFD of a
default constructor



Notice that the default constructor takes no parameters but is responsible for setting the class’ member variables. In other words, the client can only call a default constructor but cannot influence it in any other way. The dangerous thing about default constructors is that the compiler will generate one for you if it is not explicitly declared.

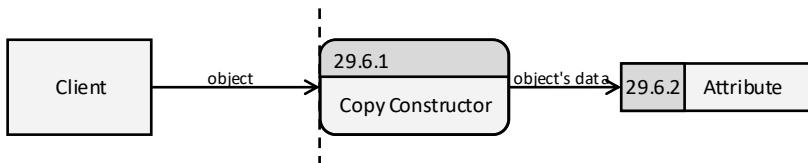
Best Practice 29.2 Always create a default constructor

Even if the compiler-generated default constructor does what you want it to do, it is a good practice to write one anyway. This makes the code easier to understand and makes future enhancements easier to implement. Note that there are some very rare circumstances when a default constructor is not desirable. This happens when only conversion constructors are an acceptable way to instantiate a class. The important thing here is that all object creations occur through a constructor.

Copy Constructor

A copy constructor is a special constructor designed to make a copy of an object. This is commonly used in pass-by-value and return-by-value function call scenarios, though they can be called explicitly. Copy constructors create objects of a class by taking as a parameter an object of the same class. Here, a copy of the input object is made.

Figure 29.6:
DFD of a
copy constructor



If the principle of data protection is completely followed, then there should be no way possible for an object to contain invalid data. This, of course, assumes that there are no bugs in the system.

Best Practice 29.3 Assert on the validity of input parameters for copy constructors

A wise programmer would validate that every input object to a copy constructor is in a valid state. Otherwise, bugs introduced in the input object will be transferred to the newly created object where they will be much more difficult to find. The best way to perform this check is with an assert.

Note that if a copy constructor is not provided, then many languages will create one automatically. This may have unintended consequences. If, for example, a class has a pointer as a member variable, then just the address of the data will be copied rather than the data to which the pointer refers. This is called a “shallow copy” rather than a “deep copy.” If the client has two objects that should have distinct values but actually do not, then the program can be in an undefined or invalid state. To see how this works, consider a simple class representing an array. It will have two member variables: the number of elements in the array and a pointer to the array buffer itself. We now will have an object created from this class with five elements in the buffer.

Figure 29.8:
The state of
an array object

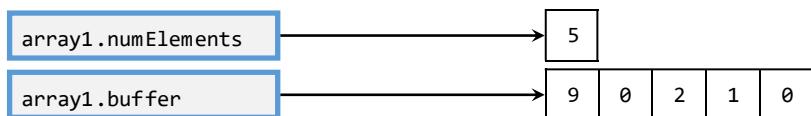
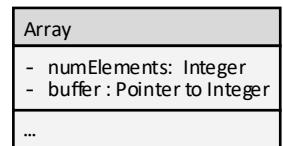
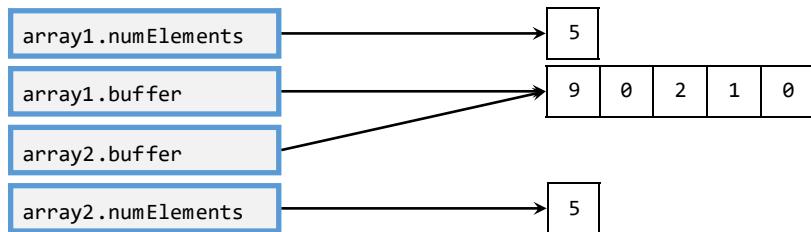


Figure 29.7:
Class diagram of an
array class



If we perform a shallow copy of this object, we will have two objects pointing to the same location in memory. The result is that changes to `array1` will affect `array2` in an unpredictable way.

Figure 29.9:
The state of
an array object after a
shallow copy is complete



Best Practice 29.4 Always create a copy constructor

Generally, there are few good reasons for not creating a copy constructor. They reduce ambiguity and make it more difficult to introduce bugs.

Move Constructor

A move constructor is like a copy constructor except the input object's member variables are stolen. In other words, the member variables of the client's input object are transferred into that of the newly created object and the input object is reverted to the default state.

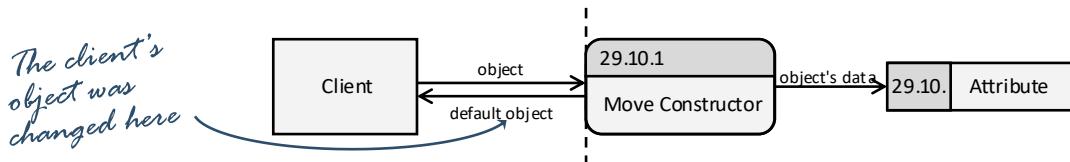


Figure 29.10:
DFD of a
move constructor

The advantage of the move constructor is that it eliminates the need to make a potentially expensive copy. If the member variables of a class are simple (such as integers, floats, etc.), then there is no advantage of creating a move constructor. When the member variables are large (such as arrays or other data structures), the cost of making a copy can be large. The move constructor avoids this copy.

Best Practice 29.5 Use a move constructor to avoid unnecessary copies

If your programming language has a move constructor, then significant performance gains can be made when the data contained within a class are large. Note that the result of a move is that the input class is returned to the default empty state.

Best Practice 29.6 The input object to a move constructor should be returned a valid state

After the data from a move constructor have been moved to the new class, the original class must remain in a good state. Usually this entails reverting the host class to the same state that results from the default constructor.

```
Pseudocode
Array.moveConstructor(input)
    numElements ← input.numElements
    buffer ← input.buffer

    input.numElements ← 0
    Input.buffer ← NULL
```

Figure 29.11:
Pseudocode of a
move constructor

In the above pseudocode, notice that the contents of the input buffer are not copied into the new object, but rather the pointer itself. Copying a pointer is $O(1)$, independent of the size of the array. Copying the contents is $O(n)$, which can be quite expensive for large arrays. This implementation is $O(1)$.

In the second part of the function, the input buffer is reverted to the default state: an array with zero elements and a `NULL` buffer pointer.

Conversion Constructors

Conversion constructors, also known as non-default constructors, are any constructors not otherwise classified as default, copy, or move. They are called “conversion” because they take one data type (the input parameter or parameters) and convert it to another (the class itself). Note that most programming languages can accept any number of conversion constructors if the parameter lists are different from each other.

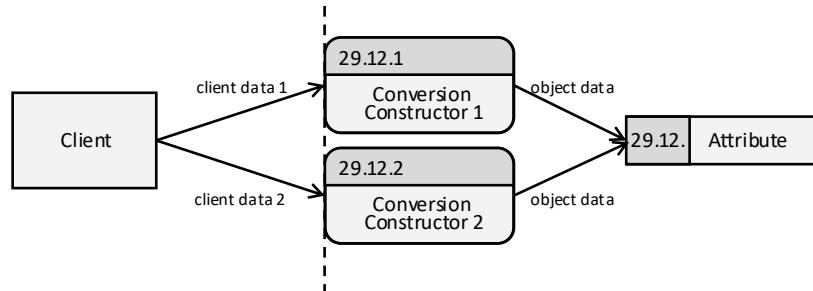


Figure 29.12:
DFD of two
conversion constructors

Usually the data entering a conversion constructor is different than that entering the member variables. Some sort of transformation is usually made to convert the data from a client-centric format to the internal representation.

Best Practice 29.7 Conversion constructors must validate their input

Conversion constructors must validate each input parameter to make sure that no invalid data makes it into the member variables. For example, a class representing temperature in Celsius may take a floating-point number as an input parameter. However, the lowest possible temperature is -273.15°. Calling the conversion constructor with a number less than this must be rejected or the class cannot offer robustness guarantees.

Best Practice 29.8 Call public mutators whenever possible

In most cases, the data passed to a conversion constructor can also be passed to a mutator. If this is the case, do not duplicate the code – just call the mutator directly from the conversion constructor.

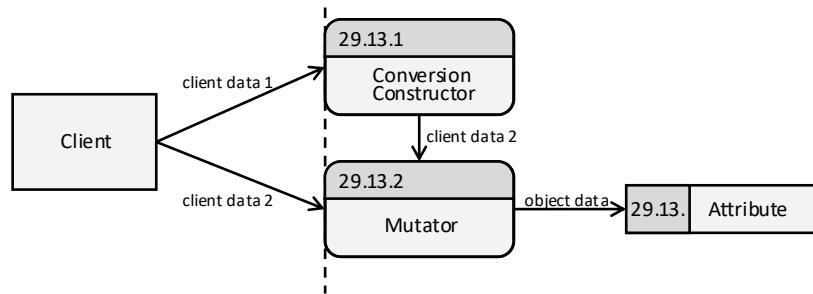


Figure 29.13:
DFD of a conversion
constructor calling
a mutator

Best Practice 29.9 Fail gracefully when invalid data are sent

Each constructor has the obligation to leave the newly created object in a valid state regardless of the provided input. In the case of a conversion constructor, invalid data should result in an object with the default value. In most cases, this means the same state as what results from the default constructor.

Mutator

A mutator is any class that alters the object's state. This could be through setting a member variable based on input from the client, or it could be resetting the member variables to some default state. Setters are mutators.

A key concept behind data protection that allows a class author to make robustness assurances is the notion that only trusted functions can change a member variable. Back to our fortress analogy, the visitors to a castle cannot directly access the king. Only the trusted guards can do so. If one wishes to speak with the king, they must speak through a trusted guard.

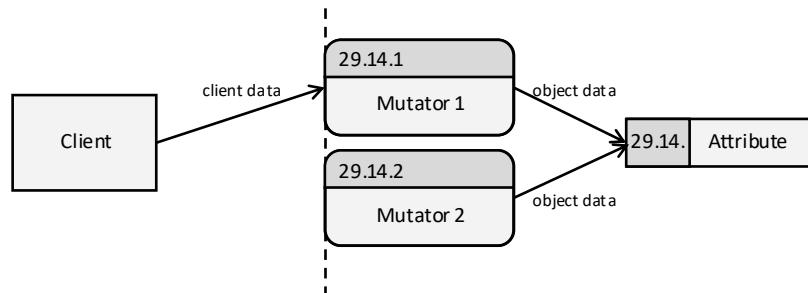


Figure 29.14:
DFD of two mutators

Best Practice 29.10 Avoid public member variables; use accessors and mutators instead

As a rule, it is dangerous to allow public access to a class' attributes. When doing so, we are relying on the client to perform data validation. This makes it hard to be sure the member variables are in a good state. There are a few exceptions to this rule. If a member variable is a Boolean, it might be very difficult to make it invalid!

Best Practice 29.11 All client-provided input should be validated

Treat all client-provided input with suspicion. We do not do this because all clients are malicious (though some might be) or ignorant (which some are). Instead, we do this to alleviate the client from the burden of performing data validation. We are moving complexity from the client's code into the class' code. This has the dual benefit of making the class more convenient and giving the methods looser coupling.

When an error is detected, then several options present themselves:

Option	Description
Do nothing	Leave the member variables unchanged and return
Display a message	Present to the user an error message
Throw an exception	Force the client to catch the exception and deal with it
Assert	In debug builds, indicate the error is catastrophic

There are certainly scenarios when each of these options is appropriate. The most common by far is to throw an exception. Since it is the client who provided the erroneous data, the client is probably in the best position to deal with any error that comes from it.

Best Practice 29.12 Accept client data in the most convenient format for the client

When the client's view of the data is different from the object's internal representation, always make public interfaces work in the client's format. This increases the level of convenience of the interface.

Accessor

An accessor is any that accesses a member variable. This could be a method that serves to return data to the client (often called a getter), it could be a method which displays data in some format (such as a display method), or it could be a method which makes a decision based on the data (such as a wait method).

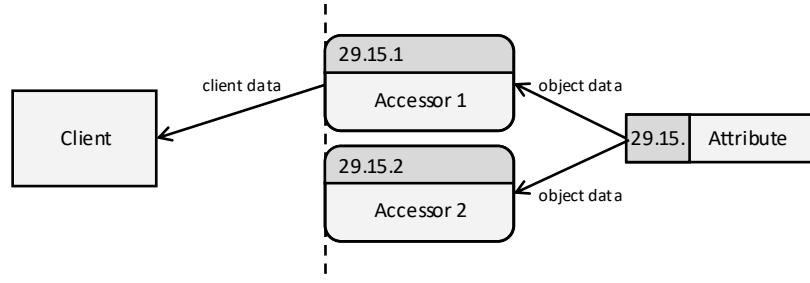


Figure 29.15:
DFD of two accessors

For a class to make robustness assurances, it must be the case that data are always returned in a valid state. The class relies on the constructors and the mutators to ensure that the member variables are set correctly. However, often the accessor needs to transform data into a client-ready format. This transformation also needs to be carefully verified.

Best Practice 29.13 Accessors should not change class state

Recall that mutators change class state whereas accessors observe class state. It is possible to have a method which does both, of course. However, if a method is specifically designed to be an accessor, then it should not change state.

Many programming languages allow the programmer to designate a method as constant. This means that it does not have the ability to modify a member variable. If your language has this capability, then all non-mutators should be constant. The benefit to this is that the compiler will notice if an inadvertent modification was made to a member variable. If you attempt to modify a member variable in a constant method, a compiler error will result.

Best Practice 29.14 Use asserts in accessors to validate class state

Remember that it is the constructor's and the mutator's responsibility to ensure that the member variables of a class are always in a valid state. This means two things. First, if the class is in an invalid state, then it is not the accessor's fault. Since such bugs do not reside in the method, it is appropriate to put state validation asserts at the beginning of most accessors.

Second, if the class is in an invalid state, then any data returned by the accessor is likely to be erroneous. Since bugs need to be fixed at their origin, putting asserts in accessors brings the developer one step closer to finding bugs.

Best Practice 29.15 Give the client data that are well-formed and in the most convenient format

Though it is most convenient for the object to return the class state in the form of the internal representation, a class author should be more concerned about what is most convenient for the client. We are constantly trying to offload work from the client into the class. This is the very heart of the convenience metric. The result of this principle is that accessors typically transform data from the internal representation stored in the member variables to some client-centric format. When this transformation is completed, always ensure that the data are well-formed.

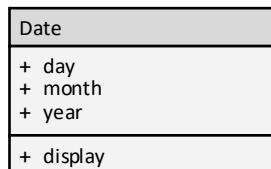
Examples

Example 29.1: Date Class

This example will demonstrate how to increase the level of robustness in a class.

Problem

Consider the following class designed to represent a date (day, month, and year). Improve the design so it will offer more data protection assurances.



Solution

Several things need to be done:

- **Make member variables private.** This is Best Practice 29.10, the first step in avoiding the client setting the month number to -99.
- **Create a default constructor.** This is Best Practice 29.2. In this case, we will default to the 1st of January 2000.
- **Create a copy constructor.** Strictly speaking, this is not needed because we only have primitive data types for the member variables. From Best Practice 29.3, we can see the benefit of an extra opportunity to validate our input.
- **Create mutators.** These will validate the data (Best Practice 29.11) and throw an exception if an incorrect value is passed.
- **Create a conversion constructor:** This will call the corresponding mutators (Best Practice 29.8) and, if they fail, call the default constructor (Best Practice 29.9).

The final class design is the following:

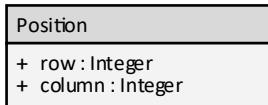


Example 29.2: Position class

This example will demonstrate how to increase the level of robustness in a class.

Problem

Consider the following class designed to represent a position on a chessboard (row and column). Improve the design so it will offer more data protection assurances.

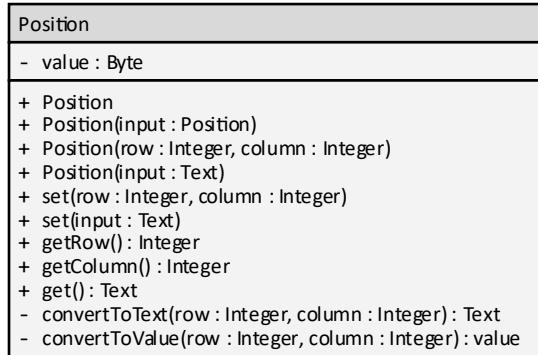


Solution

Several things need to be done:

- **Make member variables private.** This is Best Practice 29.10. While we are at it, we will use a more efficient data representation: the values 0–63 rather than 1–8 and 1–8.
- **Create a default constructor.** This is Best Practice 29.2. We will use position A1 for the default.
- **Create a copy constructor.** From Best Practice 29.3, we will benefit from better data validation.
- **Create mutators.** We will accept both textual input (“D4”) and numeric input (row = 3, column = -4) (Best Practice 29.12). These perform data validation (Best Practice 29.11) and throw an exception when the validation fails (Best Practice 29.15).
- **Create accessors.** Return the data in the most convenient format (row integer, column integer, and position text) that is valid.
- **Create a conversion constructor:** We will create two: one for textual input, and one for coordinates. These will call the corresponding mutators (Best Practice 29.8) and, if they fail, call the default constructor (Best Practice 29.9).

The final class design is the following:



Example 29.3: Stack Move Constructor

This example will demonstrate how to make the move constructor more robust.

Problem

Consider the following class designed to represent the stack data structure (first in, last out). We have a copy constructor for this class:

C++

```
template <class T>
stack<T> :: stack(const stack<T> & rhs) :
    data(NULL), numCapacity(0), numElements(0)
{
    // do nothing if there is nothing to do
    if (rhs.numElements == 0)
        return;

    // allocate buffer
    data = new(std::nothrow) T[rhs.numElements];
    if (NULL == data)
        throw "ERROR: Unable to allocate a buffer";

    // copy elements over. This is where the O(n) comes from
    for (int i = 0; i < rhs.numElements; i++)
        data[i] = rhs.data[i];

    // done!
    numElements = numCapacity = rhs.numElements;
}
```

Create a move constructor for this class.

Solution

The new object will assume the input object's array and restore the input to the default state. In this case, the default state is the empty capacity.

C++

```
template <class T>
stack<T> :: stack(stack<T> && rhs)
{
    // steal the right-hand-side's data
    data = rhs.data;
    numElements = rhs.numElements;
    numCapacity = rhs.numCapacity;

    // set right-hand-side to empty
    rhs.data = NULL;
    rhs.numElements = 0;
    rhs.numCapacity = 0;
}
```

Exercises

Exercise 29.1: Object-Oriented Constructs

From memory, provide a definition for each object-oriented construct:

Construct	Description
Default Constructor	
Copy Constructor	
Move Constructor	
Conversion Constructor	
Destructor	
Constant Method	
Public Method	
Private Method	
Accessor	
Mutator	
Member Variable	

Exercise 29.2: Benefits of Data Protection

Which design metric will the principles of data protection have the greatest benefit?
List each design metric and describe the impact of data protection principles on each.

Exercise 29.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Attributes should be private	
Mutators must validate their input	
Few classes need a constructor	
Conversion constructors should be a thin wrapper for setters	
Use IF statements to validate input parameters of a copy constructor	
Move constructors are more efficient than copy constructors	
It is OK for getters to change class state	

Problems

Problem 29.1: Currency

Consider the following design specification:

A class is designed to represent currency (money). In this case, the currency is always positive and the smallest denominations are cents. This class is used in a financial application that has direct user textual input.

Please do the following:

1. Create a class diagram to describe this class.
2. Create a DFD to illustrate how data move into and out of the member variables.
3. Provide pseudocode for the methods responsible for keeping the member variables in a valid state.

Problem 29.2: Spaceship Fuel

Consider the following design specification:

A class is designed to represent the fuel amount in a spaceship. A variety of interfaces can adjust the fuel level, including refueling stations, bonus fuel loads, and the engines (which consume fuel as they are used).

Please do the following:

1. Create a class diagram to describe this class.
2. Create a DFD to illustrate how data move into and out of the member variables.
3. Provide pseudocode for the methods responsible for keeping the member variables in a valid state.

Problem 29.3: Units of Measure

Consider the following design specification:

A class is designed to represent units of measure. The possible values are teaspoon, tablespoon, cup, pint, quart, gallon, liter, ounce, pound, gram, kilogram, and count. This class is used in a recipe application where data can be read from a file or derived from textual user input.

Please do the following:

1. Create a class diagram to describe this class.
2. Create a DFD to illustrate how data move into and out of the member variables.
3. Provide pseudocode for the methods responsible for keeping the member variables in a valid state.

Challenges

Challenge 29.1: Personal Check

Consider the following problem:

A financial application has the need to represent a personal check. You may need to do some research to see what types of attributes a check can have. The problem will allow the user to hand-enter the various fields, download them from the bank, and save them to a file.

Please do the following:

1. Create a class diagram to describe this class.
2. Create a DFD to illustrate how data move into and out of the member variables.
3. Provide pseudocode for the methods responsible for keeping the member variables in a valid state.

Challenge 29.2: Tic-Tac-Toe

Consider the following problem:

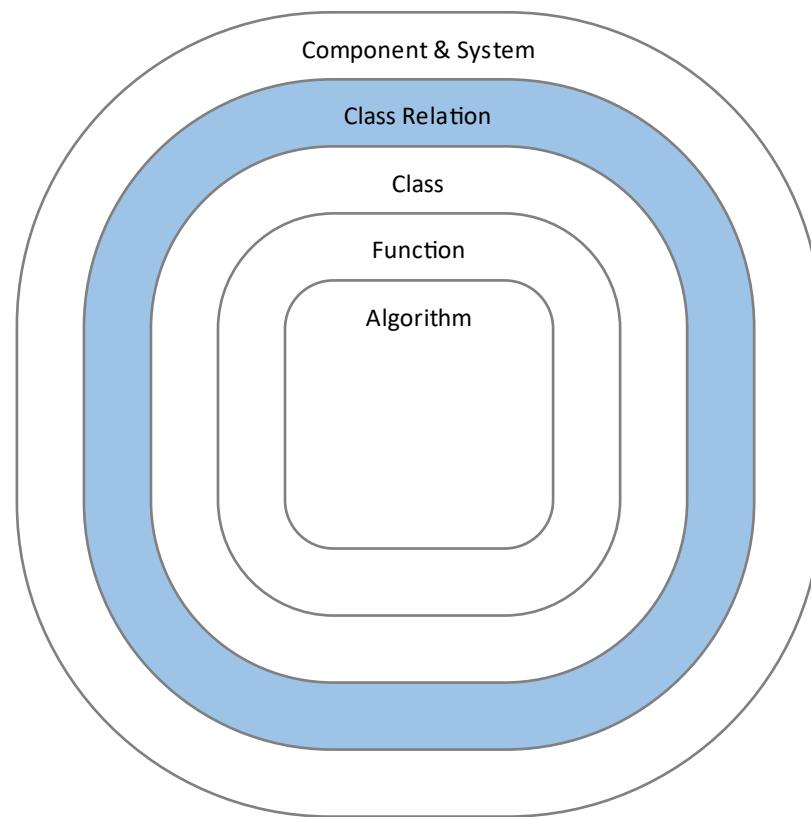
A program plays the game of Tic-Tac-Toe. It allows two users to play, one user to play, or it can even play itself. This game can be played remotely over the internet and can be saved to a file or can be played. We would like to create a class to represent the 3x3 board.

Please do the following:

1. Create a class diagram to describe this class.
2. Create a DFD to illustrate how data move into and out of the member variables.
3. Provide pseudocode for the methods responsible for keeping the member variables in a valid state.

Class Relation Design

Class design often involves a collection of closely related classes, each contributing to a single overall design. We call this class relations. Since class relations are built from individual classes, class relation design builds upon encapsulation, which builds upon modularization and algorithm design.



Class Diagram II

Chapter 30

Class diagrams do more than describe the configuration of a single class; they can also describe how multiple classes relate to each other.

We learned in Chapter 20 how to create class diagrams to describe the configuration of single classes. It turns out that this is only half of what class diagrams can do. They are also useful tools for designing and visualizing how classes are built from other classes.

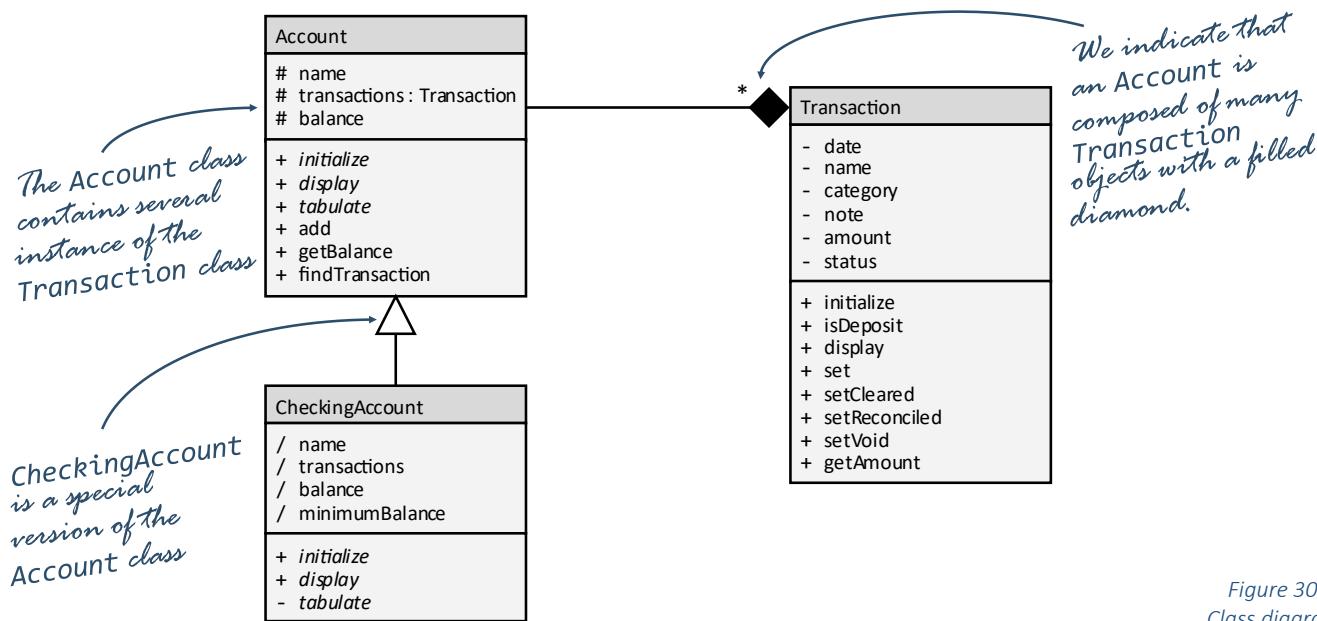


Figure 30.1:
Class diagram

Notice the arrows describing a host of different relations that classes can have with each other. The purpose of this chapter is to learn about how these arrows help us visualize class design ideas.

Class Diagram Elements

A class diagram consists of the three-row table (as described in Chapter 20 Tool: Class Diagram I), as well as the relationships with other classes. A single class diagram table is usually not enough to capture the designer's intent. In almost every case, relation arrows are also needed.

Class relations are represented with lines where the endcaps signify the type of relationship. There are two broad categories of class relations: *is-a*, drawn with vertical lines, and *has-a*, drawn with horizontal lines.

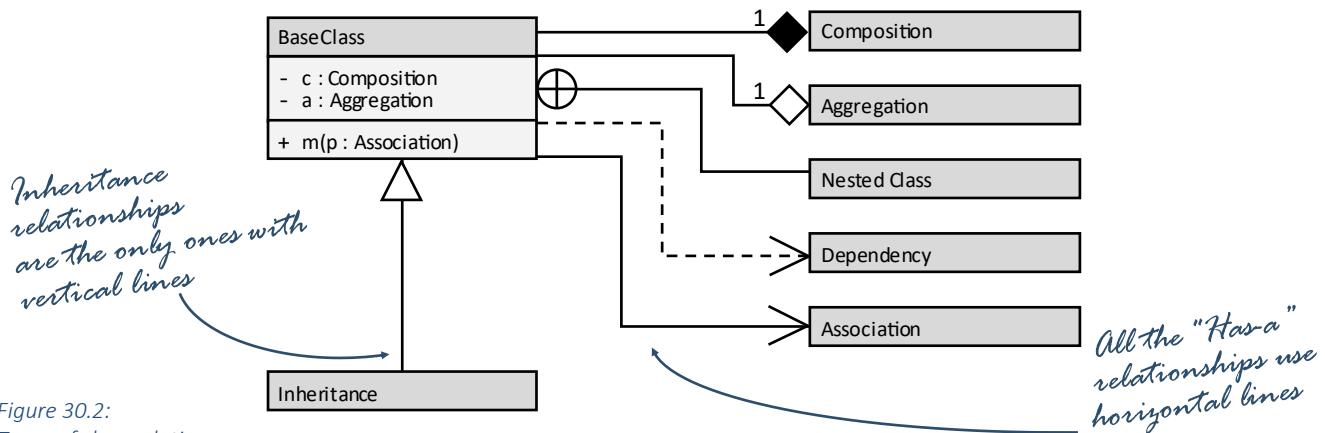


Figure 30.2:
Types of class relations

Is-a relations indicate one class is a derived class from another base class. *Is-a* relations are called “inheritance,” and these lines are drawn vertically. *Has-a* relations, on the other hand, imply that one class contains an instance of another class. There are five flavors of *has-a*: composition, aggregation, nested class, association, and dependency.

Symbol	Meaning
	Inheritance, meaning one class derives from or is inherited from another class. These arrows are vertical.
	Composition, meaning the member variable of one class is an object of another class.
	Aggregation, meaning the member variable of one class is a reference to an object of another class.
	Nested class, meaning one class is defined within another class.
	Association, meaning a local variable or a parameter of one class is an object of another class.
	Dependency, meaning one class depends on another class for some functionality.

Inheritance

Inheritance relations are one type of *is-a* where one class is a type or specialization of another. These are called *is-a* because we can describe the relation between two classes with the phrase "is a."

Inheritance is represented in class diagrams with a solid vertical line and an open (outlined) triangle as the endcap. The derived class points to the base class and, by convention, the base class is on top.

Inheritance is called *is-a* because we use that phrase in a sentence describing the relation between the two classes

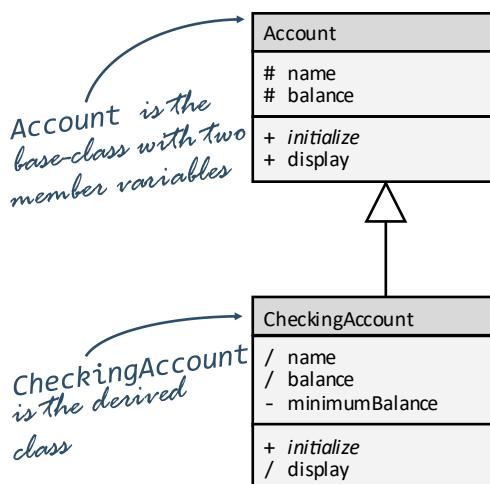


Figure 30.3:
Inheritance class diagram

For example, we might have a class called **Account** which includes such attributes as `name` and `balance`. We might create another class called **CheckingAccount** which takes the properties of **Account** and extends them. In this case, a **CheckingAccount** is an **Account**. Thus, the derived class contains all that the base class has. We can see the member variables `name` and `balance` are defined in **Account** yet are accessible from **CheckingAccount**. The derived class can also add new attributes and operations. In this case, the attribute `minimumBalance` is available in **CheckingAccount** but is not available in **Account**. Finally, the derived class can redefine operations of the base class. We can see that there is a special form of the initialization operation for the **CheckingAccount** even though a method of the same name exists in the base class.

The following is an implementation of **Account** and **CheckingAccount** in Python.

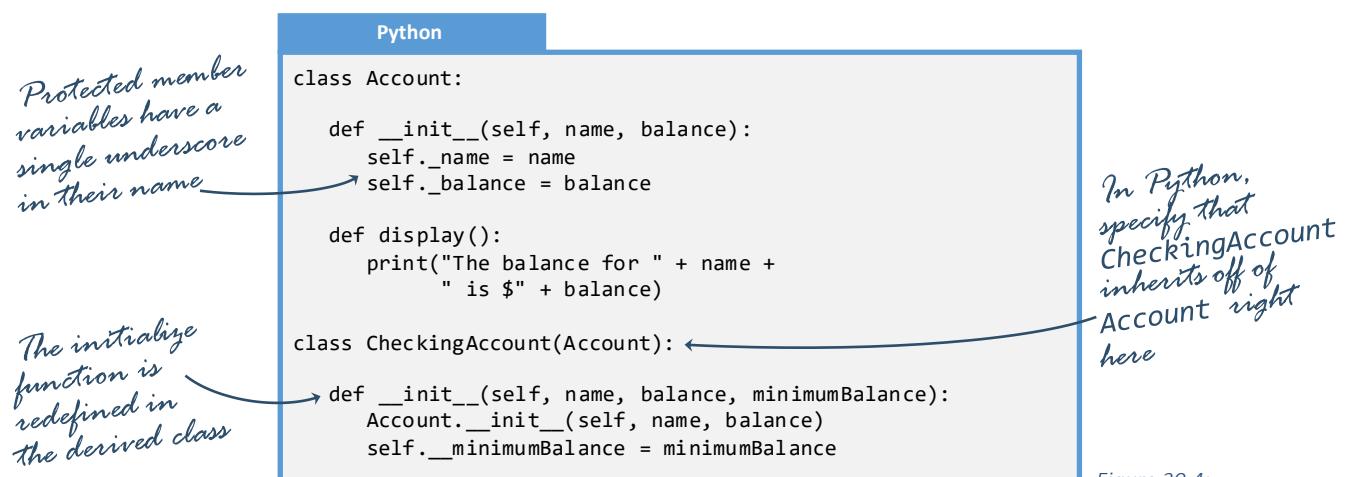


Figure 30.4:
Inheritance in Python

Composition

Composition is a type of *has-a* relation where an object from one class is part of another. When an object of the enclosing class is created, an object of the composition class is also created automatically. Likewise, destroying the enclosing class will free the composed class. In most cases, if a member variable is not of a primitive (built-in) data type but rather an object, then composition is the type of relation you are describing.

Composition is represented in a class diagram with a solid horizontal line and a solid diamond endcap pointing to the class that is being included. The number of instances (called multiplicities) of the included class is presented near the endcap. This can be a single number or a range. We can also indicate an indefinite number of instances with an asterisk.

Multiplicity

5	A fixed number of instances
0 ... 1	Zero or one instance
1 ... 10	Between one and ten instances
*	Zero or more instances

Consider a class representing a spaceship in a 3D game. This spaceship will contain the position, velocity, and orientation of the ship. It will also contain the status of the ship, including fuel load, damage, bullets, and other similar attributes. The spaceship will know how to draw itself, move itself, and respond to a variety of inputs. It will even know if it has run into another game entity. As we create this class diagram, we realize that there are many existing classes which will help us to define our **SpaceShip** class. These include **Position**, **Velocity**, **Orientation**, and **Status**. All of these are described using composition.

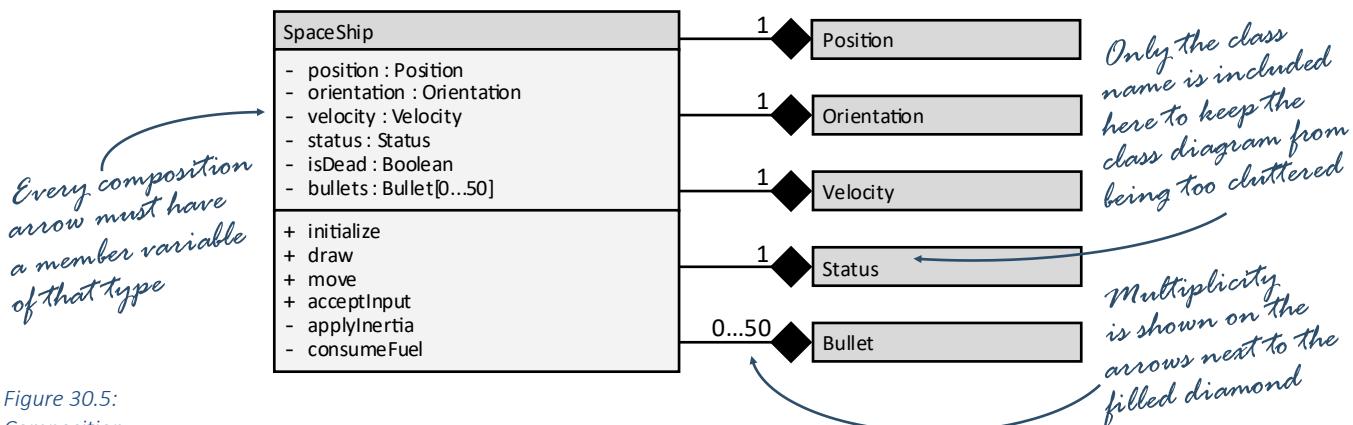


Figure 30.5:
Composition

The code in C# corresponding to the member variable declarations is the following. Note how member variables of a non-built-in type are almost always composition.

```
C#
public class SpaceShip
{
    private Position _position;
    private Orientation _orientation;
    private Velocity _velocity;
    private Status _status; ←
    private bool _isDead;
    private Bullet[] _bullet;
    ...
    ... code removed for brevity ...
}
```

When something other than a built-in data type is used as a member variable, it is probably composition

Figure 30.6:
Composition in C#

Aggregation

Aggregation is like composition, except the enclosing class does not create or destroy the aggregate object

Aggregation class relation means one class owns but may share an object of another class. In other words, aggregation is like composition with one important difference: destruction of the enclosing class does not necessarily result in the destruction of the aggregation class. Aggregation is represented with a solid horizontal line and a diamond outline. We also include multiplicity with aggregation relations.

To understand how this works, consider a program managing many accounts (checking, savings, retirement, etc.). Occasionally we would like to display a report which displays several accounts at the same time. Of course, our **Report** class would need to contain all the information necessary to generate the report, but we do not want to make an unnecessary copy of the relevant accounts. If we were to use composition, then the **Report** class would both create and destroy the pertinent accounts. Instead, we would just like to maintain a reference to the accounts. Thus, aggregation is the natural choice.

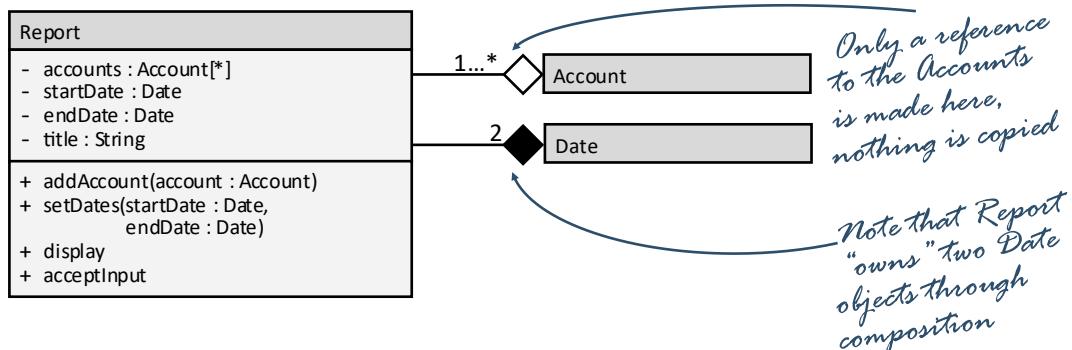
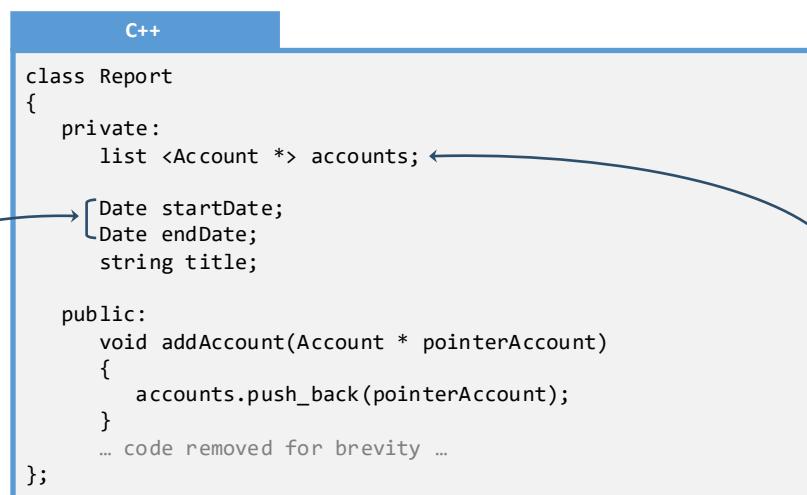


Figure 30.7:
Class diagram featuring
aggregation

Aggregation is easiest to demonstrate in languages that have pointers, though it is also possible to do so in most other languages. Notice in Figure 30.8 below how **Report** contains a collection of **Account** pointers. **Report** holds references to **Accounts** which were created and destroyed elsewhere, but it does not create or destroy an **Account**.



This is a list of pointers to accounts, not a list of accounts. Because they are pointers, this is aggregation

Figure 30.8:
Aggregation in C++

Nested Class

A nested class is a class defined within another class

Most object-oriented languages have the capacity to specify a nested class: that is, a class defined within another class. The most common use for a nested class is as a utility to the enclosing class. A nested class is represented with a solid line and a pinwheel endcap. The interesting thing about the nested class arrow is that the arrow points in a different direction than that of other *has-a* relations.

Consider a grocery list application. One aspect of this application is to represent a single list, which consists of a collection of list items. To make this class easier to use by the application, we will also create an iterator. Since the iterator only makes sense in the context of the list from which it is iterating, it is best to make **Iterator** a nested class.

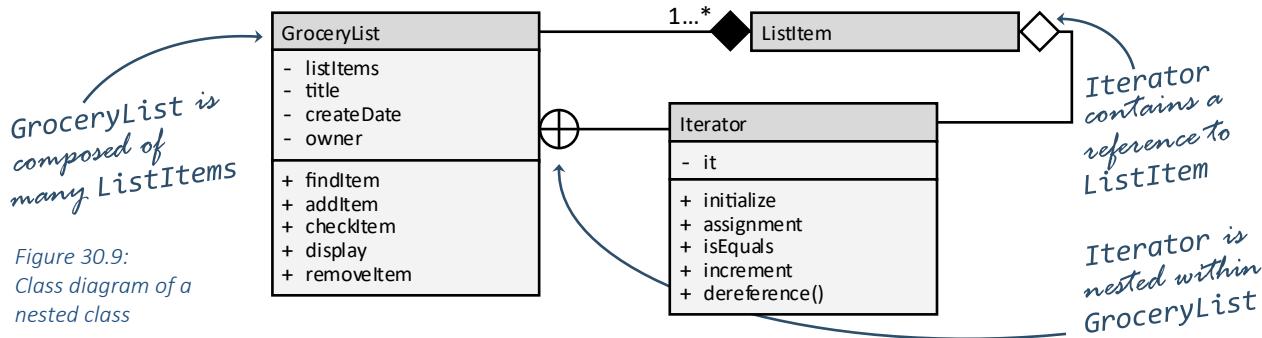


Figure 30.9:
Class diagram of a
nested class

There is a lot going on with this class diagram. The solid diamond (composition) indicates that there are many **ListItems** contained within **GroceryList**. The pinwheel (nested class) indicates that **Iterator** is defined within **GroceryList**. Finally, the diamond outline (aggregation) indicates that **Iterator** contains a reference to a **ListItem**.

The following is Java code implementing a simple nested class. Note that a class can be both a nested class and a composition class at the same time. This would be true if the enclosing class contained an **Iterator** object. In this example, it does not.

```
Java
class GroceryList {
    ... code removed for brevity ...
    public class Iterator
    {
        ... code removed for brevity ...
    }
}
```

Notice how Iterator is defined within the GroceryList class

Figure 30.10:
Nested Class in Java

Association

Association is a form of *has-a* where one class uses an object of another class but does not contain an instance. This means one class cannot contain a member variable that is of the second class. Instead, the one class usually has a local variable in a method that is freed and destroyed when the method executes. Another common usage of association is to pass a parameter to a method where the parameter is an object of another class. Like aggregation, the name “association” does not really capture this relationship well. Perhaps it would be better named “local” or “parameter.”

Association means a method in a class contains a local variable or a parameter of another class

Association relations are represented with a solid horizontal line and an open arrow. It can be helpful to include multiplicity information when specifying association, but it is seldom necessary.

For example, consider **SpaceShip** in a 3D game. This class will need to be able to draw itself on the screen when the game engine needs it. To do so, the graphical context needs to be passed to **SpaceShip::draw()**. Note that it would not be appropriate for **SpaceShip** to make a copy of the graphical context (so composition would be a poor choice) and it is not a good idea for **SpaceShip** to contain a reference to the graphical context (thereby weakening the fidelity of the class). Instead, we just want to pass the graphical context as a parameter. Thus, association is the best class relation choice.

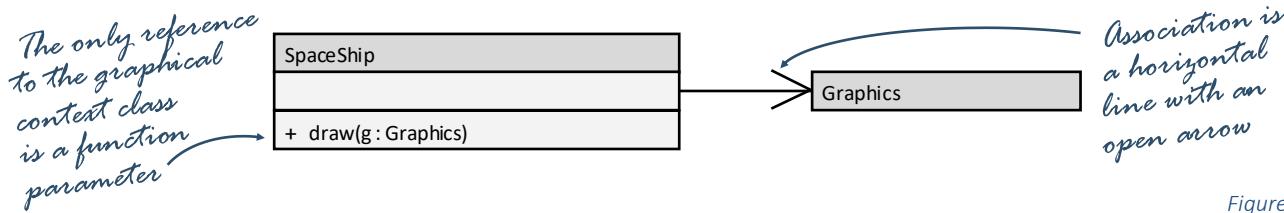


Figure 30.11:
Association

In the following example, a **Graphics** object is passed by-reference to a method in the **SpaceShip** class.

```
VB
Public Class SpaceShip
    Public Sub draw(ByRef g As System.Drawing.Graphics)
        Dim shipPen = New Pen(Drawing.Color.Blue, 5)
        g.DrawEllipse(shipPen, x, y, 25, 12)
        ... code removed for brevity ...
    End Sub
    ... code removed for brevity ...
End Class
```

The **Graphics** class is associated with **SpaceShip** because it is a parameter

Figure 30.12:
Association in VB

Dependency

The term “dependency” implies that one class relates to another in some way. Presumably, changing an interface or implementation of one class will influence the dependent class. This can be said for the inheritance, composition, aggregation, nested class, and association relations, of course. What makes dependency different than the others? In the context of class diagrams, dependency refers to any type of relation not otherwise described. Most commonly, dependency describes the use of a global variable that is an object of another class, a call to a static method without using an object, or the use of an anonymous variable.

Dependency refers to any type of class relation not otherwise described

Dependency relations are represented with a horizontal dashed line and an open arrow. As with other *has-a* relations, the horizontal orientation constraint of the line can be relaxed to accommodate the layout needs of the class diagram.

To illustrate a dependency relation in action, consider a **Position** class for a 3D flying game. Not only does a **Position** need to contain the x, y, and z coordinates, it also needs to know the extent of the arena in which the user will fly. The **Game** class itself does not need to contain a **Position** object, either as a member variable or as a local variable. It does, however, need to set the bounds of the arena. Since no **Position** object is created and since the **Position** class is not defined in terms of any other class, none of the other class relations applies. This leaves us with the dependency relation.

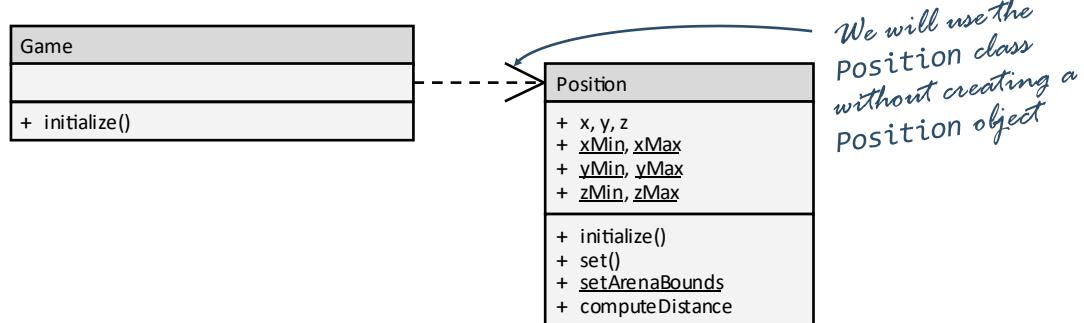


Figure 30.13:
Dependency

In the above example, our **Game::initialize()** method will call the static **Position::setArenaBounds()** method without creating a local variable (which would make this association) or creating a member variable (which would make this composition). The only relation left is dependency.

Swift

```
class Game {  
    func initialize() {  
        Position.setArenaBounds(-100, -100, -100,  
                               100, 100, 100)  
        ... code removed for brevity ...  
    }  
    ... code removed for brevity ...  
}
```

No Position object is created, but a Position method is called

Figure 30.14:
Dependency in Swift

Designing with Class Diagrams

Designing class relations effectively is perhaps one of the biggest challenges of program design. Do it well and your development and maintenance efforts will be greatly simplified. Do it poorly and you will have months or years of frustrating toil. There are two guidelines for using class diagrams to help with this design process: design *has-a* independently of *is-a* and break large class diagrams into more manageable and understandable chunks.

Has-A and *Is-A*

Has-a relations (composition, aggregation, nested class, association, and dependency) are containment relations. They are conceptually and mechanically quite different than *is-a* (inheritance). As such, they should be designed independently. (As an aside, I wish class diagrams represented them more distinctly. Using similar symbols to represent things which are so different is misleading.)

One designs *has-a* relations with a class diagram in much the same way one designs function relations with a structure chart:

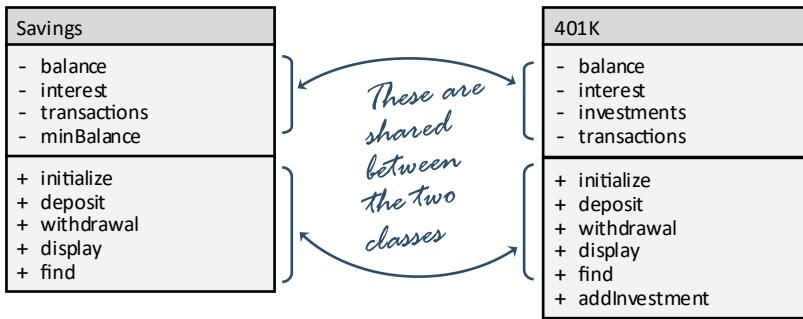
Structure Chart	Class Diagram: <i>Has-A</i>
Structure charts describe verbs, that is, functions	Class diagrams describe nouns, that is, classes
The purpose is to describe the dependency relationship between functions	The purpose is to describe the dependency relationship between classes
Each function performs some work and delegates the rest to subfunctions	Each class represents some concept and delegates the rest to subclasses
You start with the function of interest and work down	You start with the class of interest and work right
Each function performs part of the parent function's work	Each class represents part of the parent class's concept
One function can be called by many functions	One class can be utilized by many classes

We follow much the same design process with class diagrams as we do with structure charts: you start general and work your way to more specifics, always looking for opportunities to leverage that which is already built.

Designing for *is-a* relations is quite different than designing for *has-a* relations. In fact, during the design process, it is usually a good idea to design them completely independently of each other.

Rather than designing top-down as one does with *has-a* class relations and with function relations, *is-a* design usually follows a bottom-up approach. Start with a class describing some aspect of your program. If you find another class that represents a similar concept, then create a base class containing the shared concept. For example, say your personal finance program has two classes representing two different types of accounts: savings accounts and 401(k) retirement accounts.

Figure 30.15:
Two classes before
introducing inheritance

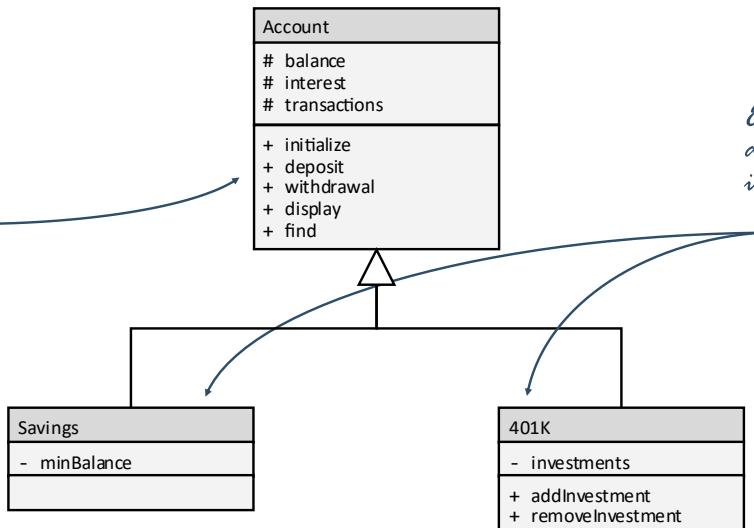


Programming is about finding patterns and exploiting them to make the code more efficient or elegant. In this case we can bring the common components into a single abstract class called **Account**.

Everything common between the classes goes in the base-class

Everything unique about each class goes in the derived class

Figure 30.16:
The same classes after
moving the commonalities
to a base class



As more account types are added to this inheritance tree, we will probably realize that investment accounts are different than loan accounts. From here, there are many subdivisions that might come out as well. In each case, commonalities are represented in base classes and differences are moved to the derived classes.

Hierarchies

A class diagram for a single class can take the better part of a page. When you start adding class relation information, it can become very large and confusing. How does one deal with this complexity? The answer is to use a similar technique to what was employed with flowcharts and data flow diagrams: hierarchies.

Before hierarchies are described, it is important to understand how class diagrams are used. Fundamentally, a programmer has two questions when trying to implement a given design or fix a bug in a given design: “how does this all fit together?” and “what influences the class I am working on?” We will use two different class diagrams to describe each of these.

Question: How does this all fit together?

To get a holistic view of the entire class relation for all the classes in a given program, we need to have everything visible on a single page and we need to remove unnecessary details. We accomplish this by creating a single graph containing all the class names (but none of the attributes nor operations) and all the class relations. This grand map provides a holistic view of the entire system but provides none of the details. We will need to look elsewhere for those details. Back to our financial software problem, our class design may look like the following:

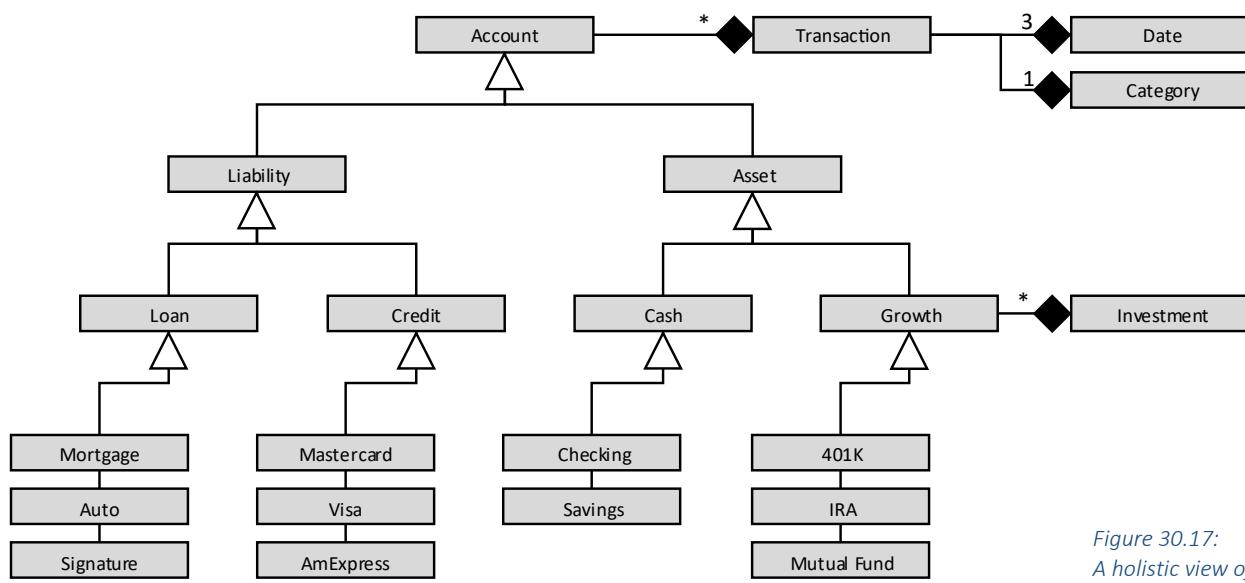


Figure 30.17:
A holistic view of a large
class relationship

From this class diagram, we can immediately see that there are two broad categories of accounts (assets and liabilities) and four subcategories. We can also see that accounts have collections of transactions, each of which has three dates and one category. We can also see that growth accounts have a collection of investments.

In order to implement any of these classes, we need another view.

Question: What influences the class I am working on?

While we typically have only one or two holistic class diagrams for a given system (or subsystem or feature...), we commonly have many class diagrams, each of which focuses on one specific class. Here, we represent the class in all its detail, including all the attributes, operations, and data types. We also represent the connections to related classes so they can be understood.

For example, say I wished to analyze the **Growth** class. From our holistic view, we can see that **Growth** is a child of **Asset** which is a child of **Account**. It also has several children of its own: **401K**, **IRA**, and **Mutual Fund**. We can also see that this class has several **Investments**. Putting this all together, the following class diagram would be developed:

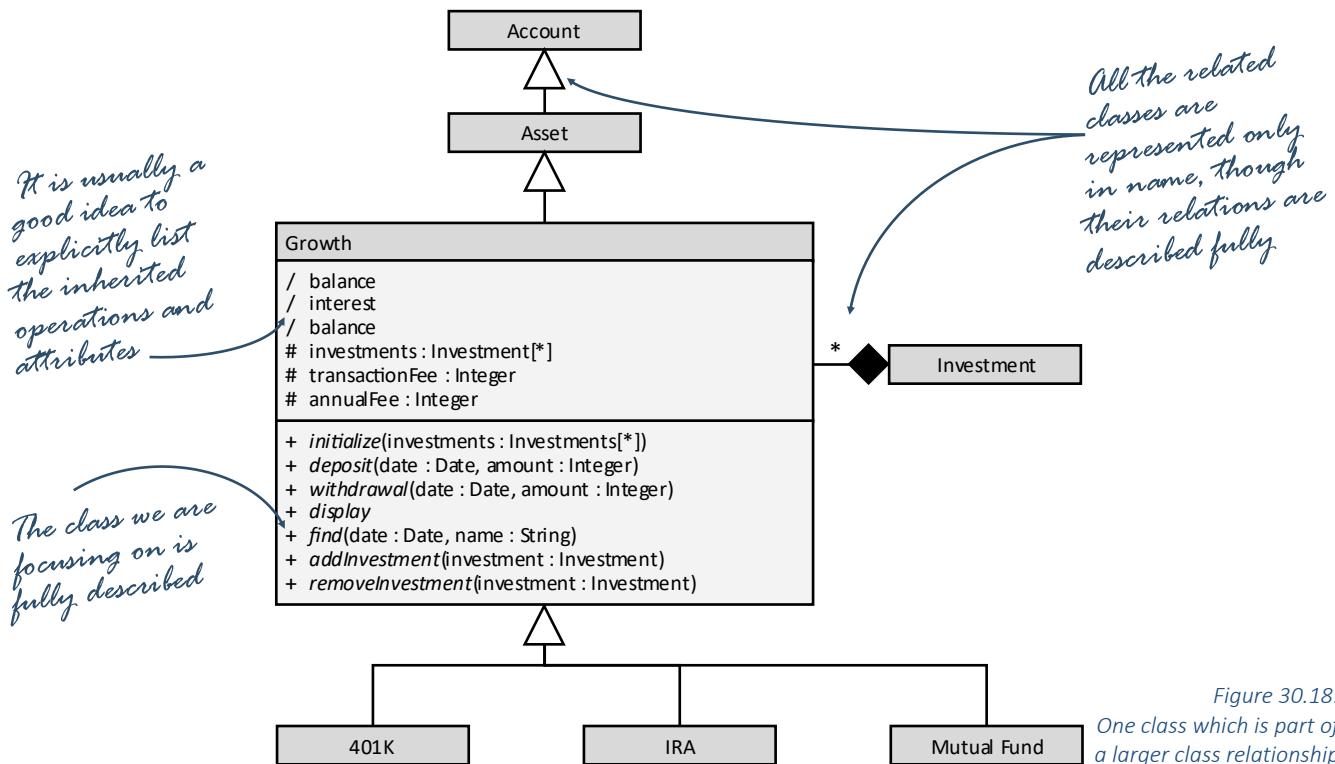


Figure 30.18:
One class which is part of a larger class relationship

There are 22 individual classes described in the holistic class diagram for this application. This means we will need a total of 23 class diagrams to describe everything (1 for the holistic view and 22 for the individual classes). Without the holistic view to tie it all together, it is easy to see how programmers could get lost or confused.

Examples

Example 30.1: Class Diagram from Code

This first example will demonstrate how to create a class diagram to match existing code.

Problem

Create a class diagram corresponding to the following code representing a chess move in a game.

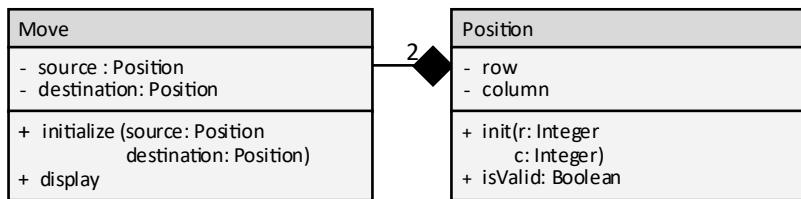
Python

```
class Position
    def __init__(self, row, col):
        self.__row = row
        self.__col = col
    def isValid(self):
        if self.__row >= 8 or self.__row < 0:
            return False
        if self.__col >= 8 or self.__col < 0:
            return False
        return True

class Move
    def __init__(self, source, destination):
        self.__source = source
        self.__destination = destination
    def display(self):
        print "(" + __source + ", " + __destination + ")"
```

Solution

Notice that there are two classes where **Move** contains two member variables of type **Position**.

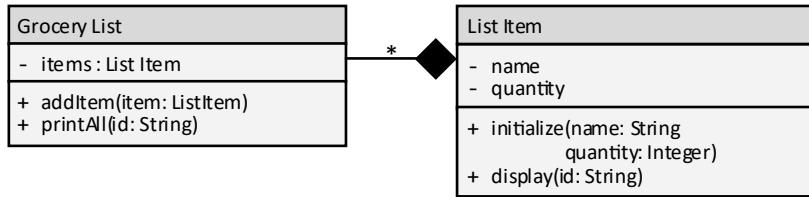


Example 30.2: Code from Class Diagram

This second example will demonstrate how to create code from a class diagram.

Problem

Generate the code corresponding to the following class diagram:



Solution

JavaScript

```
class ListItem {
    #name;
    #quantity;
    constructor(name, quantity) {
        this.#name = name;
        this.#quantity = quantity;
    }
    display(id) {
        Document.getElementById(id).innerHTML = #name;
    }
}

class GroceryList {
    #items;
    addItem(item) {
        #items.push(item);
    }
    printAll(id) {
        for (var i = 0; i < #items.length; i++) {
            #items[i].display(id);
        }
    }
}
```

Example 30.3: Class Diagram for a Chart

This third example demonstrates how to create a class diagram to represent a simple problem definition in several distinct stages.

Problem

Create a class diagram matching the following description:

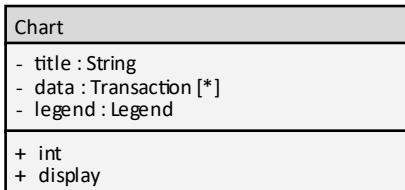
A financial chart is a graph consisting of a title, a chart legend, and the data of the chart. The data are a collection of individual transactions from a given account.

Solution

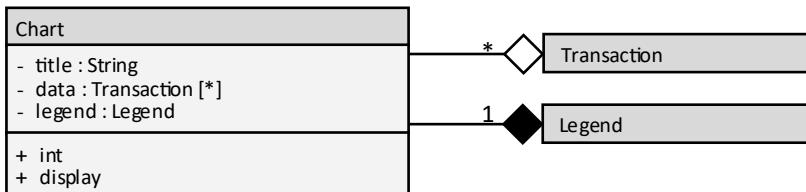
The first step is to work on the top-level class. We will call this a **Chart**.



Next, we will define the attributes and operations associated with **Chart**.



It has become evident that we need two helper classes to finish this: one to contain a legend and one to contain the transactions. It is important to note here that we do not want to make a copy of the transactions; they should stay in the **Account** class. Thus, we will use aggregation rather than composition. However, the **legend** member variable is both created and destroyed with the **Chart** class. Thus, we use composition to represent this relationship.



Example 30.4: Class Diagram for a Missile

This fourth example will demonstrate how to use class diagrams to design with inheritance.

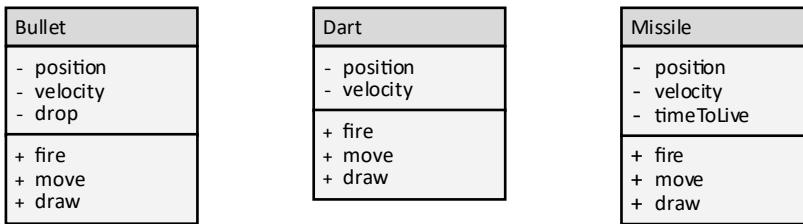
Problem

Create a class diagram matching the following scenario:

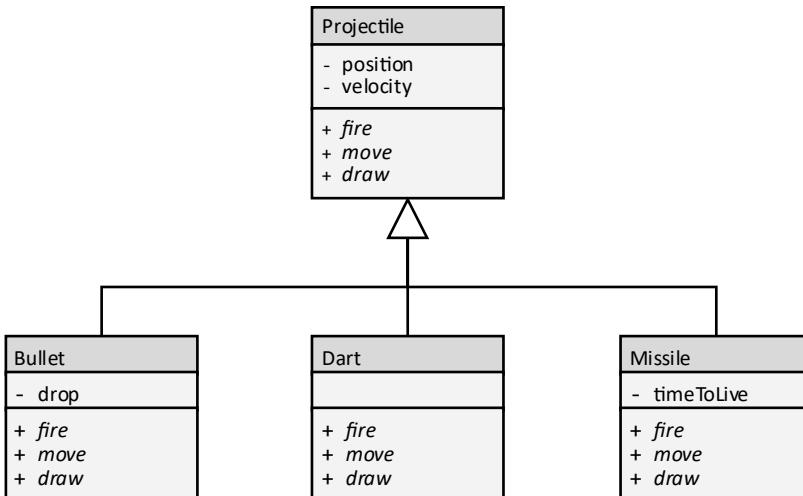
A 3D flying game has three types of projectiles: a bullet, a dart, and a guided missile. Each has a position and velocity. The bullet honors gravity, dropping slightly with every frame. The dart always flies straight. The guided missile always turns towards the target but blows up after 30 frames. Each renders differently, of course.

Solution

The first iteration will be to create a class diagram for the bullet, dart, and missile.



Next, we will realize the commonalities between these: they all have a position and velocity. Each has a shoot, move, and draw function, but the specifics are quite different. We therefore need to make them virtual so they can have separate implementations.



Exercises

Exercise 30.1: Name the Endcap

Name the type of class relation based on the endcap.

Endcap	Class Relation Name

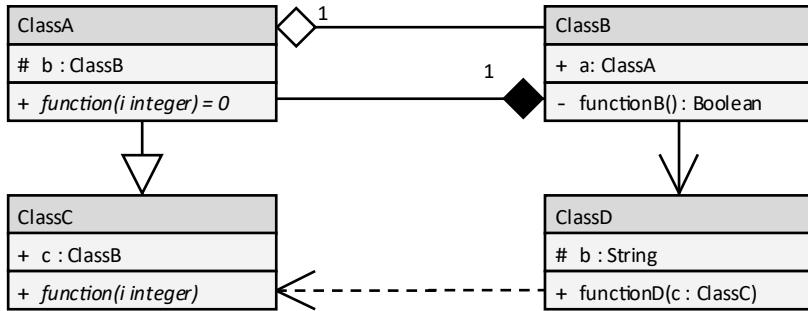
Exercise 30.2: Identify Parts of a Class Diagram

Draw the symbols corresponding to the parts of a class diagram.

Description	Symbol
ClassA uses a global object of type ClassB	
ClassA inherits from ClassB	
ClassB is defined within ClassA	
ClassA has three instances of ClassB as member variables	
ClassA calls a static method from ClassB	
ClassA has a reference to an object of ClassB	
ClassA has a method which has an object of ClassB as a local variable	

Exercises 30.3: Class Diagram Errors

Identify the errors or inconsistencies in the following class diagram:



Exercises 30.4: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Nested relationships should specify multiplicity	
Composition diamonds point to the class that is being included	
Inheritance relations should specify multiplicity	
Nested pinwheels point to the class that is nested within the enclosing class	
If there is a composition relation, then there must be a local variable of that type	
Aggregation relations should specify multiplicity	

Problems

Problem 30.1: Class Diagram from Code

Create a class diagram to match the following C++ code.

C++

```
class Time
{
private:
    int secondsSinceMidnight;
public:
    Time() : secondsSinceMidnight(0) {}

    int getHours() const;
    int getMinutes() const;
    int getSeconds() const;
    virtual void display() const;

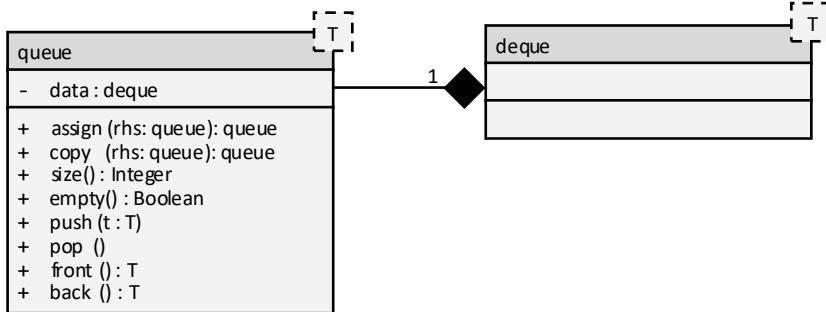
    void set(int hours, int minutes = 0, int seconds = 0);
};

class GermanTime : public Time
{
public:
    void display() const;
};

class FrenchTime : public Time
{
public:
    void display() const;
};
```

Problem 30.2: Code from Class Diagram

In the programming language of your choice, write code to match the following class diagram.



Problem 30.3: Class Diagram for a Grocery List

Create a class diagram to match the following problem definition:

A grocery list application maintains a collection of grocery lists. Each grocery list can be one of three types: alphabetical list, categorical list, or cost-priority list. A list item consists of three components: the name, the quantity, and the category. The category is a class which consists of an enumeration of 10 categories.

Problem 30.4: Class Diagram for a Chart

Create a class diagram to match the following problem definition:

A personal finance application has a chart feature. Each chart has a title and a collection of transactions to be displayed. There are several categories of charts: tables, bar graphs, pie charts, and line graphs. For table charts, there can be single-account transactions, category summaries, and by-month summaries. For bar graphs, there are income/spending histograms and account-balance histograms. For pie charts, there are two flavors: income categories and spending categories. Finally, the line graphs will be two flavors: single-account histogram, net worth histogram.

Challenges

Challenge 30.1: Bank Statement

Create a class diagram to represent your credit card statement.

Look at your credit card statement this month. Create the class necessary to represent that entire statement. List each individual field on the statement and organize it into cohesive groupings.

Hint: This will be a large composition exercise. If there will be few if any inheritance relations.

Challenge 30.2: Asteroids

Create a class diagram to match the following problem:

Consider the 1979 Atari game *Asteroids*. You may need to do some “research” to refresh yourself on how this program works. This game consists of several elements: three types of rocks, bullets, a spaceship, and a flying saucer. The game itself has two scores (the current game and the all-time high score) and a counter related to the number of lives remaining.

Have one holistic class diagram describing the entire system. Then have a collection of detailed class diagrams describing each individual class in the system.

Hint: This will be a large inheritance exercise. There will likely be a few composition relations, but inheritance should dominate your class diagram.

Challenge 30.3: Recipe book

Create a class diagram to represent a recipe book.

Look at a recipe book. Are there different types of recipes (such as baking vs. cooking vs. mixing)? Create a class diagram to describe the entire book. This includes a class to represent the book (title, author, and recipes), a class to represent a recipe with several variations, a class to represent instructions, and one to represent ingredients.

Have one holistic class diagram describing the entire system. Then have a collection of detailed class diagrams describing each individual class in the system.

Adaptability

Chapter 31

Adaptability is the suitability of a base class to leverage existing code to represent previously unknown design concerns through the addition of child classes.

Of the four maintenance costs typically associated with “completed” software (corrective, adaptive, perfective, and enhancements), adding new functionality that was not part of the initial specification often consumes 60% of the overall cost. The inheritance component of object-oriented programming can help in this regard. One could say that inheritance was designed specifically to reduce the cost of enhancements and adaptive changes.

When an inheritance tree is properly designed and built, it should be possible to easily add new functionality by adding derived classes. We call this property adaptability. Adaptability is the suitability of the base classes in an inheritance hierarchy to facilitate the creation of derived classes which can fulfill previously unknown or unanticipated design concerns. There are several parts to this definition. First is the phrase “facilitate the creation of derived classes.” Most of the work done to adapt a program should come through creation of derived classes, not through modification of base classes or altering the structure of the inheritance tree. Second, the phrase “fulfill previously unknown or unanticipated design concerns” refers to the fact that we often have an incomplete view of how the program will be used when it is finished or even what will be needed to build it in the first place. A big part of software engineering is managing uncertainty. A well-designed inheritance tree acknowledges this uncertainty by making it possible to easily make changes when the need inevitably arises.

Adaptability is the suitability of the base classes in an inheritance hierarchy to facilitate the creation of derived classes which can fulfill previously unknown or unanticipated design concerns

Levels of Adaptability

There are five levels of adaptability. Each of these levels focuses directly on how much work is required to make a change to the system, and indirectly on how much the inheritance hierarchy needs to be adjusted to accommodate such a change.

Level	Description
Enabling	Much functionality can be added through small changes.
Straightforward	Changes can be made solely by adding child classes.
Convoluted	Changes require minor alterations to the base class.
Prohibitive	The amount of work is greater than the benefit received.
Closed	Enhancements through inheritance are not possible.

Note that adaptability is easy to measure after the fact: simply look at how much effort was required to make changes to an inheritance tree over the course of the lifetime of a project. It is much more difficult to forecast the level of adaptability when the code is under construction.

Closed Adaptability

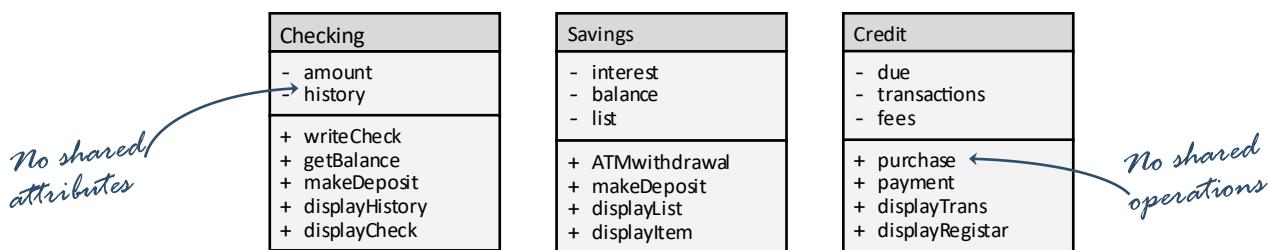
An inheritance hierarchy can be categorized as closed if there has been no allotment made for extensibility. This could be because the designer did not appreciate the potential need for augmentation, or it could be that the types of changes are different than the designer anticipated. There are two indicators of closed adaptability: lack of inheritance in the design, or lack of common properties/attributes.

Closed adaptability is when no allotment has been made for enhancement or modification through inheritance

Lack of Inheritance

Designs lacking inheritance relationships are closed by definition

If a collection of related classes is not linked by inheritance, then they can be considered closed. In cases such as these, no benefit can be achieved by leveraging similar functionality or data representation from related classes; every class is an island unto itself. For example, consider the following class diagram representing a collection of unrelated financial accounts.



There is no allotment made for extensibility in the current class design. If a new account type is to be added, then a new class will have to be created from scratch.

In the above example, the three accounts have different public interfaces. Presumably a new class will also have unique interfaces, forcing the client to do even more work to accept this new account type. This is the hallmark of closed adaptability. To handle a design such as this, the client would need to create a façade. This façade would have to know about the interface details of each of these classes. There has to be a better way!

Figure 31.1:
Closed adaptability

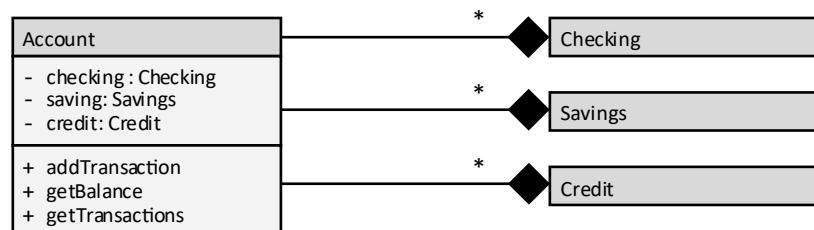


Figure 31.2:
A façade can help mitigate closed adaptability

Illogical Inheritance

Another common source of closed adaptability is when derived classes are defined in terms of a common base class, but the sibling classes have little or nothing in common. It can be difficult to objectively identify illogical inheritance, but there are some symptoms which are easy to spot: the base class lacks attributes, only contains generic methods, or has a generic base class name.

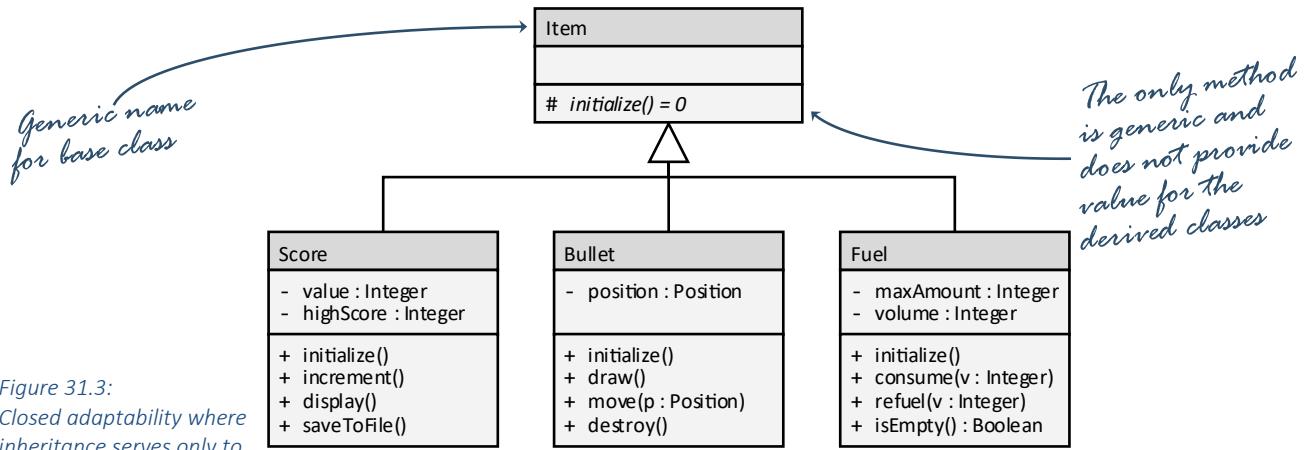


Figure 31.3:
Closed adaptability where
inheritance serves only to
provide a shared initialize
interface

Notice that this design is superior to the design in Figure 31.1 on the previous page because we can use the base class for polymorphism. It is superior, but not by much! It uses inheritance in a very restricted way. Notice that the base class provides no services to the derived classes; each derived class must fully implement each of its own methods. The only benefit of the base class is that it facilitates polymorphism so the client can more easily integrate the diverse set of items. The cost of building a new item, however, is still as high as if there was no inheritance.

Prohibitive Adaptability

An inheritance hierarchy can be categorized as prohibitive if the amount of work required to extend a class is greater than the benefit one could expect from the inheritance. In situations such as these, the design is so convoluted that it would have been easier to just start from scratch or implement the class independently. Usually we do not have this option. Instead we need to refactor a large part of the inheritance tree to accommodate the new change. Once this is done, then we can go about adding the new class. The two most common causes of prohibitive adaptability are incompatible virtual functions and illogical inheritance trees.

Prohibitive adaptability is when the amount of work required to extend a class is greater than the benefit one could expect from the inheritance

Incompatible Virtual Functions

It is often the case that a base class has virtual functions that a derived class is required to implement (be that pure virtual functions or otherwise). When these virtual functions are not aligned with the model of the derived class, then they can be said to be incompatible. Consider the following inheritance hierarchy where we are trying to add a new type of account: stocks.

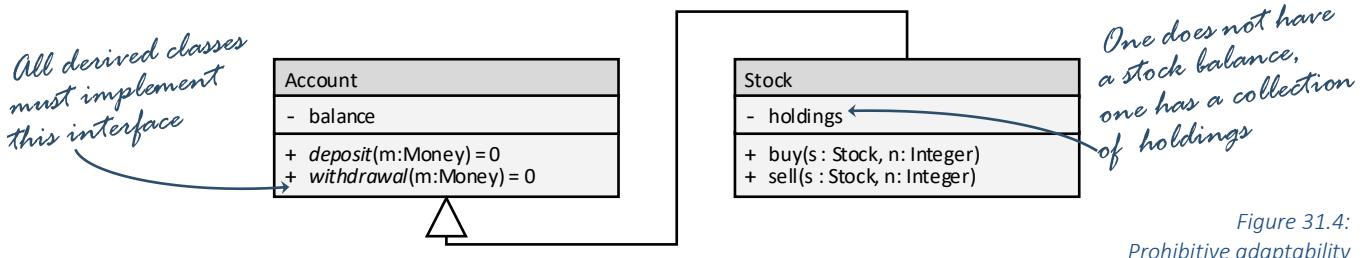


Figure 31.4:
Prohibitive adaptability

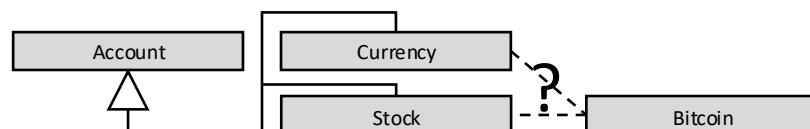
All the other accounts in the system work in terms of deposits and withdrawals. Stocks do not work that way; you buy and sell shares. Making this new sub class work under the **Account** base class will require a great deal of ingenuity; how can one even fulfill the **deposit()** and **withdrawal()** functions? They are fundamentally incompatible!

Incompatible Base Classes

A second cause of prohibitive adaptability is when the inheritance tree is created in such a way that there is no obvious place to put the derived class. Worse, what does one do when there are two or more places to put the new class?

For example, consider an inheritance hierarchy for a collection of different types of financial accounts. One base class handles currency, having the ability to add and remove money as needed. The second handles stocks whose value changes according to the current market price and the number owned. We would like to add Bitcoin to this hierarchy. The problem is that Bitcoin can be regarded as currency and has the volatility of a stock. It needs both!

Figure 31.5:
Another example of
prohibitive adaptability.
Where does Bitcoin go?



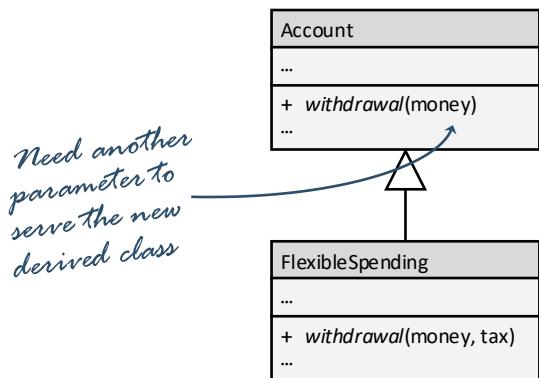
Convolved Adaptability

An inheritance hierarchy can be categorized as convoluted when some changes to the system require minor alterations to the base class and/or sibling classes. This is not a good level of adaptability but is much easier to manage than closed or prohibitive. There are three flavors of convoluted adaptability: interface modification, reimplementations, and expansion.

Convolved adaptability is when some changes to the system require minor alterations to base classes

Interface Modification

Interface modification is the process of changing function signatures in base classes



for the purpose of accommodating the needs of a new derived class. This necessitates modification of all the derived classes which utilized the changed methods. When the changes are simple (such as adding an optional parameter) it might have minimal impact on the other derived classes. When methods are removed or require different parameters, the impact can be severe. Generally, interface modifications of existing classes should be handled cautiously. For example, consider adding a flexible spending account to our financial inheritance tree. Here, all transactions are required to be accompanied by tax information. Since the withdrawal method is inherited, we cannot add the

extra parameter without modifying the base class. This change necessitates an interface modification.

Figure 31.6:
Convolved adaptability

Reimplementation

Reimplementation is the process of altering the logic of one or more methods in a base class without changing the interface. Since the interface is unchanged, one may be tempted to think that it is not necessary to retest all the derived classes. This is false! Every derived class that used the reimplemented method needs to be reverified. Hopefully, the class author has provided unit tests to facilitate this!

Expansion

The most innocuous form of base class modification is interface expansion, involving adding member variables and/or methods to a base class that were previously unutilized by the other derived classes. If done correctly, we can be assured that our change is localized to just the newly added subclass.

For example, imagine a new financial account like a standard savings account except having an annual fee. We could place the fee code in the new class, but it is likely that other accounts will require the same functionality. To handle this, we add a new method and member variable to our **Account** class. The new method and member variable is only utilized by the new derived class, thereby limiting its impact on the rest of the program.

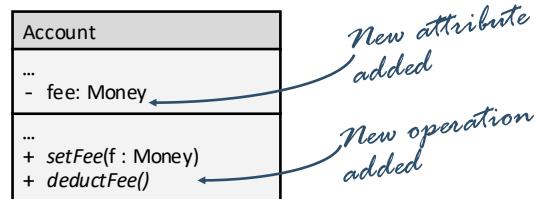


Figure 31.7:
Interface expansion

Straightforward Adaptability

Straightforward adaptability is when foreseeable changes to the system can be made solely through the addition of child classes

prohibitive, and closed adaptability should be avoided.

An inheritance hierarchy can be categorized as straightforward when all foreseeable and realized changes to the system can be made solely through the addition of child classes. In other words, no changes are required for the base class. When designing inheritance trees, we should always strive for straightforward or enabling; convoluted,

One of the most important advantages of straightforward adaptability is that it is generally unnecessary to retest all the sibling derived classes. Since the change is localized to the newly added class, the siblings are unaffected. Depending on the way the derived class uses the base class, we might need to do minimal verification on the base class as well. However, we need to verify that the new use of the base class does not expose previously untested conditions.

Another advantage of straightforward adaptability is that all the development effort can be focused on the new functionality, not on making the old system work with a new design. Thus, the development effort is considerably decreased in this situation.

Back to our financial account example, consider adding a MasterCard to our existing credit card class. There are important differences which need to be considered, such as a feature called price protection. Fortunately, we can completely implement this feature in the derived class and under a single virtual function. Because the base class is unchanged and creating of the derived class is easy, this can be classified as straightforward adaptability.

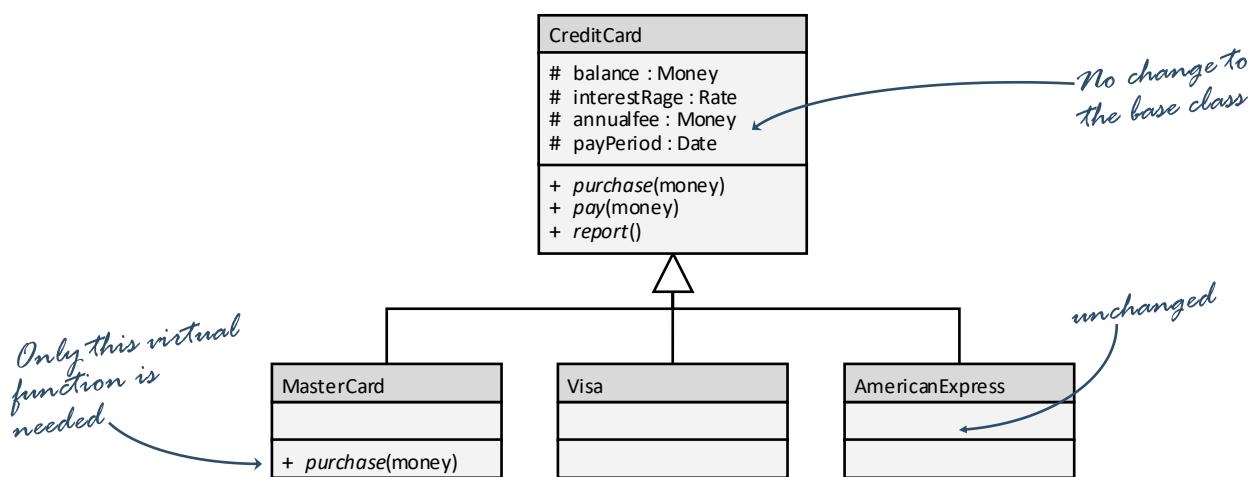


Figure 31.8:
Straightforward adaptability

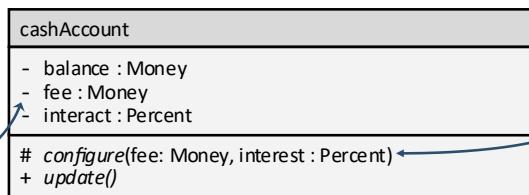
Enabling Adaptability

An inheritance hierarchy can be categorized as “enabling” when a great deal of functionality can be added to the system through simple inheritance changes. Ideally, one would be able to implement new functionality with almost no code change or, if code is required, it is trivial to create. In many ways, this aligns with the configurable and straightforward levels of malleability.

For example, consider our financial accounts problem. To make our accounts inheritance tree as data-driven as possible, the base class will “ask” the derived classes for the most common types of information. This configuration information will then be used by the base class to handle most account types. When adding a new account type, the programmer needs only to provide the base class with the needed information rather than write any code.

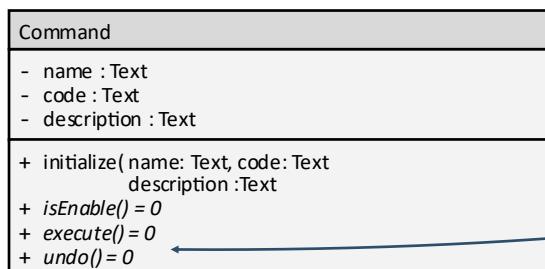
Enabling adaptability when a great deal of functionality can be added to the system through simple changes

*Most cash accounts
only need to adjust
these variables*



*Individual
accounts call
the configure
function*

It is difficult to visualize how enabling adaptability can simplify the work of the designer without a more in-depth example. Imagine a large application designed to perform many user-initiated commands. These commands can vary wildly, from displaying a simple dialog box to saving a file to running a complicated report. Every operation, however, has a few common elements: a name, a code used to identify the command in a log file, a verbose description used for tooltips and help features, a status flag indicating whether the action is ready to execute, an action which actually does work, and an undo which reverses that which was previously done. Now, since there will be potentially hundreds or even thousands of commands that the system can perform, we need to make it as easy as possible to add a new command. To do this, we create the following class:



*Each command
needs only to
implement what is
unique about it*

Figure 31.10:
Enabling adaptability

Now each command needs to implement a derived class. Specifically, it needs to call the base class’ `Initialize()` function to set up the member variables and then implement the three virtual functions (`isEnabled()`, `execute()`, and `undo()`).

Figure 31.9:
Enabling adaptability

Pseudocode

```

class CommandCloseFile  inherited from Command

    CommandCloseFile()
        Command("Close file",
                "CLS",
                "Close and archive a user file")

    isEnabled()
        RETURN true

    execute()
        file.status ← CLOSED

    undo()
        file.status ← OPENED

```

Figure 31.11:
A class implementing the command interface

The important thing to recognize about this design is how easy it is to add a new command to the system. The developer only needs to provide information directly related to the new command she is implementing. Furthermore, the class itself is a template as to what needs to be done. It can almost be called self-documenting code: the base class “asks” questions (by requiring parameters of the derived class), and the derived class answers them.

Enabling adaptability is easy to identify in retrospect (when one is looking back on a change that was recently made). It is difficult to identify looking forward because one needs to anticipate the future needs of the class and the application. How can one anticipate these future needs? One can gather such data by talking with the marketing analyst, the product sponsor, or other members of the team. It is often their job to think several years ahead and envision what future releases will entail.

Best Practice 31.1 Avoid overengineering

A common pitfall programmers fall into when designing for adaptability is called “overengineering.” Overengineering is the process of making a design more complicated than necessary for the task at hand. In other words, we can build too much for tomorrow and suffer compromises today. Common indications of overengineering include overcomplication of simple tasks, performance problems, and excessive member variables. The desire to facilitate enabling adaptability runs contrary to our desire to avoid overengineering. How does one balance these two forces? Unfortunately, there is no easy answer. It takes years of experience to do this right.

Best Practice 31.2 Avoid universalization

Universalization is the opposite of abstraction. This is the process of making a single type that contains all possible attributes. This class can be very complicated and difficult to verify, thereby undoing all the benefit we received by using inheritance. Universalization is usually the result of over-application of the principle of adaptability, when we attempt to parameterize everything. As with overengineering, it can be difficult to detect when we have gone too far down the path of adaptability. As with overengineering, it takes years of experience to do this right. The first step is to recognize universalization and appreciate why it is dangerous.

Related Metrics

The inheritance metric of adaptability is highly related to the algorithmic metric of malleability and the encapsulation metric of convenience. This should not be surprising; often, fundamental design principles such as these manifest themselves on a variety of design levels.

Adaptability and Malleability

Recall that the maintainability metric of malleability is defined as a measure of how easy it is for a programmer to make changes to a system. There are a lot of concepts shared with adaptability. In fact, the only meaningful difference is scope: Malleability is not restricted to any level of programming, whereas adaptability refers specifically to inheritance. In other words, adaptability is a special form of malleability. As a reminder, malleability has four levels of quality:

Level	Definition
Configurable	Big changes can be made without altering any code
Straightforward	Small changes can be made without altering much code
Convoluted	Refactoring and redesigns are required for most tasks
Prohibitive	It is easier to start over than to change the code

Malleability is how easily a programmer can make changes to a system

The first thing to realize is that three levels of malleability align perfectly with three levels of adaptability (straightforward, convoluted, and prohibitive). The names are the same, but the definitions of convoluted and straightforward are a little different. Malleability is a metric of how much work needs to happen in a given algorithm or function in order to accommodate changing needs of the system, whereas adaptability is more focused on the degree to which a base class needs to be changed. In other words, from an inheritance perspective, the hope is that only derived classes need to be added.

The second thing to realize is that the top level of adaptability is different from that of malleability. The most desirable level of malleability is configurable, where the system can be updated without changing code through some data-driven design. Can the essential characteristics of a function be parameterized in such a way that they can be manipulated without altering the underlying algorithm? The most desirable level of adaptability is enabling, where the selection and design of base classes facilitate adding new functionality with little effort. Can the essential characteristics of a class be parameterized in such a way that they can be manipulated through the addition of a small number of virtual functions? The goals are the same in both cases, but the mechanism one uses is different.

Adaptability and Convenience

The encapsulation metric of convenience is defined as how easy it is for a client to use a class in an application. If you consider the client to be the author of the derived class and the provider to be the author of the base class, then there is a great deal of overlap with the inheritance concept of adaptability. As a reminder, the convenience metric of encapsulation has five levels of quality:

Level	Description
Seamless	Perfectly aligned with the needs of the application
Easy	No extra work is required to manipulate
Straightforward	Any data manipulation can be readily performed
Convolved	Much needs to be done to use the class
Prohibitive	More work is required than it is worth

Convenience centers on utilizing an existing class, whereas adaptability is about adjusting an inheritance hierarchy

As with malleability, the bottom three levels of convenience align with the middle three of adaptability. The definitions, however, are quite different. The difference here is that it is understood that occasionally a base class will need to be modified in the inheritance scenario. In the encapsulation scenario, classes are often provided "as-is," with little opportunity to make changes. Classes in a library or framework, for example, are often read-only. This distinction is the heart of the difference between adaptability and convenience: convenience centers on utilizing an existing class, whereas adaptability is about adjusting an inheritance hierarchy.

Adaptability Influences Robustness

The encapsulation metric of robustness is not related directly to adaptability but these two principles need to be considered simultaneously when designing base classes. As a reminder, robustness is defined as the degree of resistance a class has to being placed in an invalid state or sending invalid data to another part of the program.

The further up the inheritance tree a class resides, the more important it becomes for that class to be robust

The more code that depends on a single class or function, the more important it becomes to make that class or function robust. This is true whether that class or function is utilized by many parts of the program or whether that class is the base class to a large inheritance tree. Thus, the further up the inheritance tree a class resides, the more important it becomes for that class to be robust. Unfortunately, this yields an interesting dilemma. The further up the inheritance tree a class resides, the more difficult it becomes to validate it! This is because such classes are very abstract, containing mostly generic functionality and simple attributes. They offer shared interfaces that the derived classes utilize, which can be extremely difficult to test! Often dedicated base classes are developed with the sole purposes of testing what little there is to validate. Recall the concept of doubles from Chapter 25. We often need to create dummies, fakes, spies, and mocks to validate base classes.

Robustness is a measure of how well a class avoids being a source of bugs

The further up the inheritance tree a class resides, the more difficult it becomes to validate it

Best Practice 31.3 Be very careful about changing a base class.

Contemplate changes to a base class the same way you drive on icy roads: look far up the road and avoid drastic changes.

Examples

Example 31.1: Closed Adaptability

This example will demonstrate how to identify closed adaptability.

Problem

Identify the level of adaptability from the following class diagram:

SpaceShip	Alien	Bullet
- x,y,z - dx,dy,dz - direction	- x,y,z - dx,dy,dz - direction	- x,y,z - dx,dy,dz
+ shoot + thrust + turn + move	+ run + attack	+ zoom

In this scenario, we would like to add a new type of ship.

Solution

Notice that these classes are not joined by inheritance. This automatically makes it closed adaptability.

Even if these were joined by a common base class, there are no shared interfaces. Thus, the inclusion of a common base class would probably still result in closed inheritance.

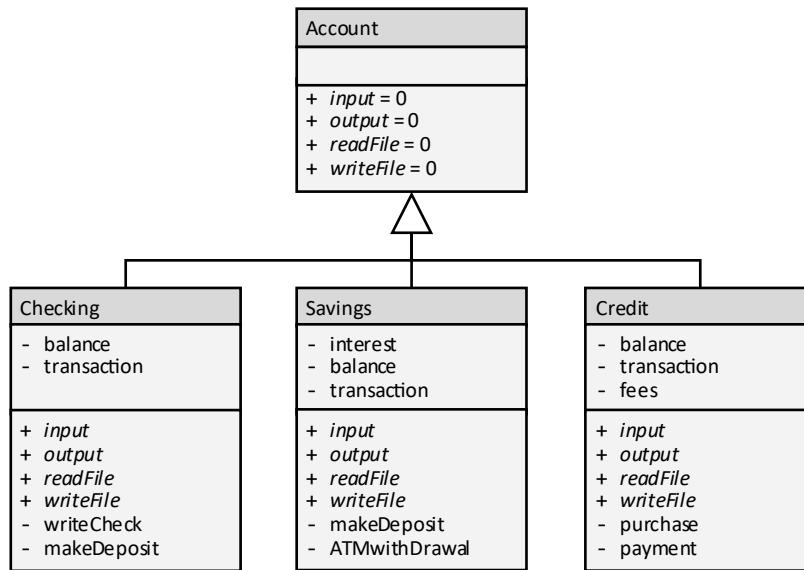
In this scenario, we wish to add a new ship. It would be a mistake to inherit directly from **SpaceShip** because it is not designed to be a base class. Instead, we should inherit off a **Ship** class from which **SpaceShip** and **Alien** are derived. This would make the level of adaptability either straightforward or enabling, depending on how the **Ship** class is designed.

Example 31.2: Prohibitive Adaptability

This example will demonstrate how to identify prohibitive adaptability.

Problem

Identify the level of adaptability from the following class diagram:



Solution

The three derived classes are all siblings of the same base class and there are shared interfaces. This does not meet the “lack of inheritance” or “illogical inheritance” criteria of closed adaptability.

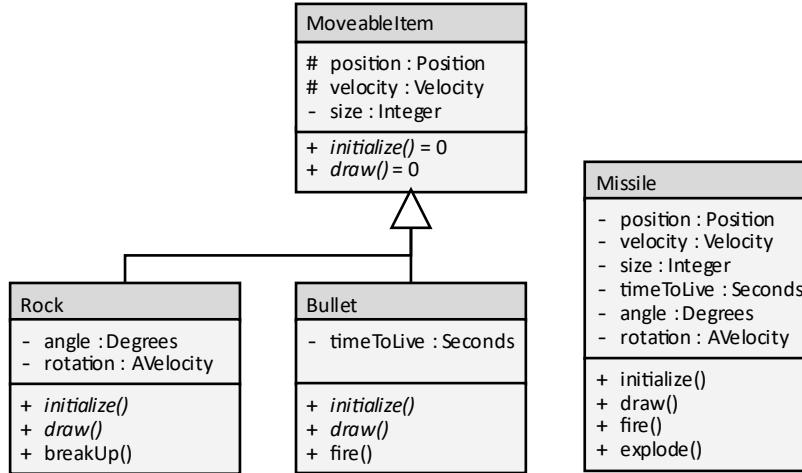
The base class is compatible with the derived classes, but the methods are not. Each derived class has its own way to modify the balance (through writing a check, making an ATM withdrawal, or making a purchasing). These then need to be articulated into the generic `input()` and `output()` interfaces provided by the base class. This requires a great deal of work, more than would be the case if there were no base class at all. For this reason, the class design can be classified as prohibitive.

Example 31.3: Convolved Adaptability

This example will demonstrate how to identify convoluted adaptability.

Problem

Identify the level of adaptability from the following class diagram where we are adding the **Missile** class:



Solution

The inheritance tree is clearly not closed because there is inheritance and there are shared methods/attributes.

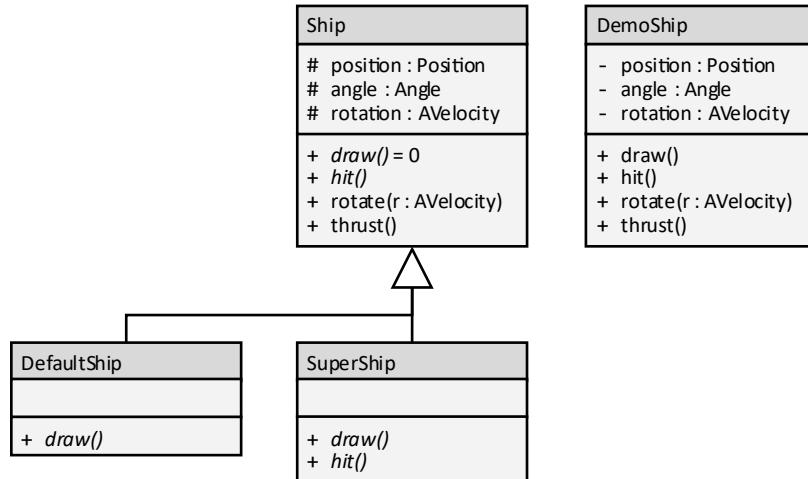
Note how a **Missile** takes part of **Rock** and part of **Bullet**. This leaves us with several bad options: either make **Missile** a child of **Rock** and duplicate **Bullet**'s functionality, make it a child of **Bullet** and duplicate **Rock**'s functionality, or move all the common attributes/methods to **MoveableItem** where **Rock** and **Bullet** will not use everything. In any case, it is convoluted: the base class will need to change.

Example 31.4: Straightforward Adaptability

This example will demonstrate how to identify straightforward adaptability.

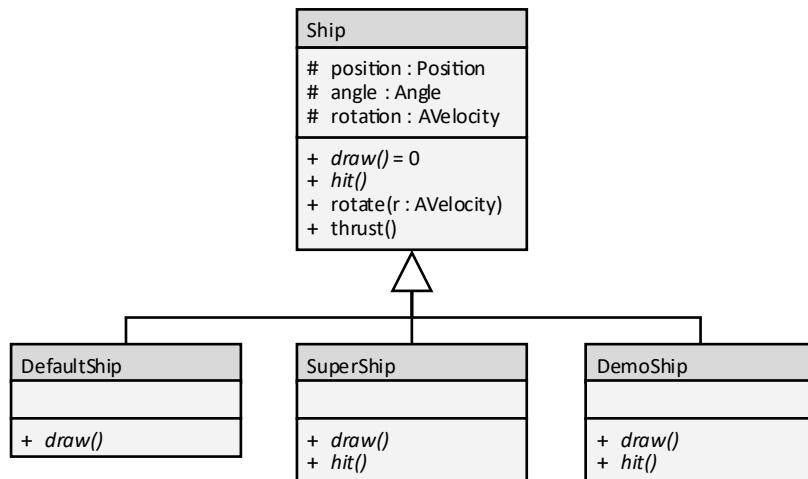
Problem

Identify the level of adaptability from the following class diagram when we wish to add a **DemoShip** class.



Solution

Notice that all the attributes of the new class **DemoShip** are already present in the base class **Ship**. Notice also that all the methods of **DemoShip** already exist in the base class. Assuming that **DemoShip::rotate()** is the same as **Ship::rotate()** and that **DemoShip::thrust()** is the same as **Ship::thrust()**, then we can easily add **DemoShip** as a derived class of **Ship** without making any modifications of **Ship**. For this reason, the adaptability is straightforward.



Exercises

Exercise 31.1: Adaptability Definitions

From memory, define each of the adaptability levels:

Level	Definition
Enabling	
Straightforward	
Convoluted	
Prohibitive	
Closed	

Exercise 31.2: Adaptability Characteristics

Characterize the level of adaptability based on the scenario:

Scenario	Level of Adaptability
The virtual functions in the base class have little to do with the needs of the derived classes.	
In most cases, adding a new derived class involves just writing new methods for the new derived class.	
Five classes are standalone without an inheritance tree to connect them.	
Adding new derived classes can be accomplished by writing trivial code.	
Adding a derived class requires modifying the interface of the base class.	
Two classes are related by inheritance in name only; they share no common properties or attributes.	
Adding a derived class requires adding an attribute to the base class.	

Exercises 31.3: Fact or Fiction

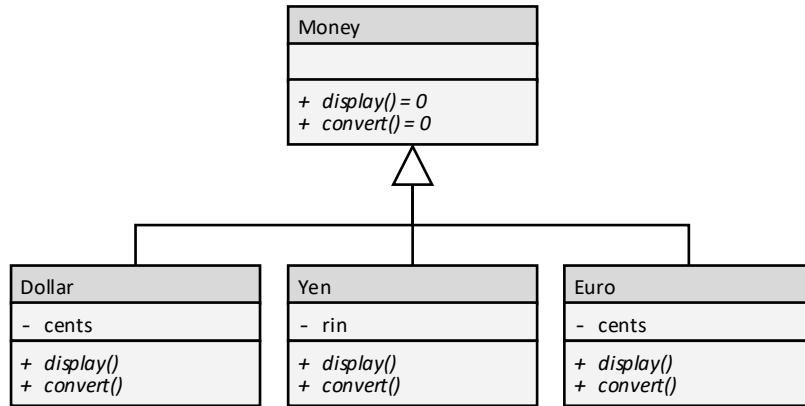
For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
The further up the inheritance tree a class resides, the more difficult it becomes to validate it.	
Overengineering is the process of going the extra mile to make things perfect.	
Changing methods in base classes is as easy as changing methods in derived classes.	
Universalization is superior to abstraction.	
The further up the inheritance tree a class resides, the more important it becomes for that class to be robust.	
Enabling adaptability is similar to configurable malleability.	
Generally, well-designed inheritance trees facilitate enhancing code.	

Problems

Problem 31.1: Identify Level of Adaptability

Consider the following class diagram. The base class has no member variables and two pure virtual functions. The derived classes honor the base class' interface but provide all their own functionality. The `convert()` method allows one currency type to convert to another.

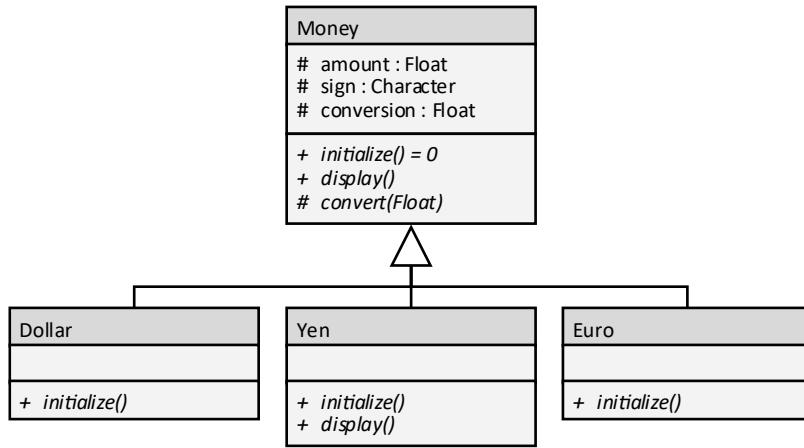


Characterize the level of adaptability based on the following scenario:

Augment the money class to include an English pound.

Problem 31.2: Identify Level of Adaptability

Consider the following class diagram. The `initialize()` function sets the sign (used in the `display()` method) and conversion value (used in the `convert()` method) in the base class.

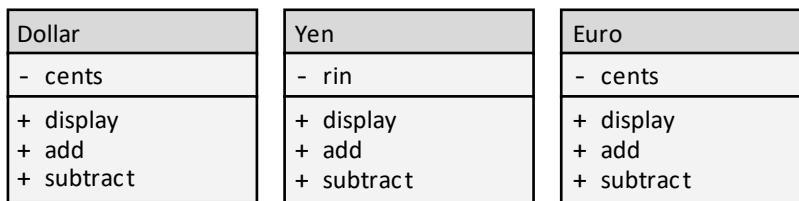


Characterize the level of adaptability based on the following scenario:

Augment the money class to include an English pound, which behaves much like the American dollar and the euro.

Problem 31.3: Identify Level of Adaptability

Consider the following class diagram:



Characterize the level of adaptability based on the following scenario:

Augment the money class to include an English pound.

Problem 31.4: Design a Class

Create a class diagram for the following problem:

A personal finance application can display several types of reports: a budget report, a histogram showing account balances over time, and a cash flow graph depicting income vs. outgo.

Based on this class diagram, characterize the level of adaptability.

After your initial characterization of adaptability, make the following change:

Create a new report type: a pie chart. This will display the percentage of spending associated with each budget category.

After making this change, characterize again the level of adaptability. Were your predicted and realized levels of adaptability comparable?

Problem 31.5: Design a Class

Create a class diagram for the following problem:

A personal finance system can have three types of users: an administrator who has complete access to the entire system, an auditor who has read-only access to the entire system, and a standard user who can only read/write those accounts specifically assigned to her.

Based on this class diagram, characterize the level of adaptability.

After your initial characterization of adaptability, make the following change:

Add a new type of user: restricted. This user can have read-only access to only her specifically assigned accounts and no other.

After making this change, characterize again the level of adaptability. Were your predicted and realized levels of adaptability comparable?

Challenges

Challenge 31.1: Financial Software

Create a class diagram for the following problem:

A personal finance application is designed to track the spending and budget of a college student. This system stores the user's data on a server, having both a web and a mobile user interface. The system can handle a variety of different types of accounts and produce a variety of reports.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of adaptability.

After your initial characterization of adaptability, make the following change:

Create a new account type: Bitcoin. Create new report types: histogram of Bitcoin value and Bitcoin account balance. You may need to do a little research on Bitcoins to complete this assignment.

After making this change, characterize again the level of adaptability. Was your predicted and realized levels of adaptability comparable?

Challenge 31.2: List Application

Create a class diagram for the following problem:

A personal list mobile application allows users to keep track of a variety of types of lists: to-do lists, grocery lists, life goals, and assignment due dates. The application allows users to add and remove lists, and even create reports created from data collected from many lists. Some reports include recently added items, recently completed items, and urgent.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of adaptability.

After your initial characterization of adaptability, make the following change:

Create a new type of list: a recipe. Create a new type of report: money spent in the last 24 hours.

After making this change, characterize again the level of adaptability. Were your predicted and realized levels of adaptability comparable?

Challenge 31.3: Video Game

Create a class diagram for the following problem:

A video game allows users to fly a ship through a 3D world filled with obstacles to be avoided. Points are added as levels are cleared and prizes are discovered. Points are deducted as walls and obstacles are hit, and as time elapses.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of adaptability.

After your initial characterization of adaptability, make the following change:

Create a new type of obstacle: a homing land-mine that is attracted to your current position. Create a new type of prize: if you fly through a glowing gold sphere, you get a 20-second time bonus.

After making this change, characterize again the level of adaptability. Were your predicted and realized levels of adaptability comparable?

Challenge 31.4: Personal Project

Create a class diagram for a personal project you have already completed or are thinking of completing in the future. Characterize the level of adaptability of this design. What design alterations can you make that will simplify future enhancements?

Chapter 32 Alignment

Alignment is the degree to which the inheritance tree models relationships in the problem domain.

Fundamentally, software is about modeling things in the real world. This could be something as tangible as a checkbook or as abstract as a video game. In each case, those 1s and 0s need to mean something. While this may seem obvious, often programmers get fixated on the tools of programming (IF statements, functions, and classes) and forget what they represent. You know you are in this predicament when you have difficulty naming things: when your variable names become single letters, when a function name has the word “stuff” in it, or when a class is called “holder.” This problem is so common in object-oriented design that we have a special label for it: alignment.

Alignment is the degree in which the inheritance tree models relationships in the problem domain. The phrase “problem domain” means the thing in the real world that the software is meant to do. Perhaps this is best explained by example. If one

were to implement a tax feature on a personal finance package, then it would probably be necessary to hire an accountant or a tax attorney. This individual, commonly known as a subject-matter expert (SME), would most likely know nothing about programming but would be the master of the problem domain. If this SME were to walk into the office of the programmer and point to a random piece of code, theoretically the programmer should be able to explain to this infinitely patient SME how that code relates to the problem domain. When this can be said, then the code has achieved alignment.

Alignment is the degree in which the inheritance tree models relationships in the problem domain

Levels of Alignment

There are four levels of alignment. You may notice that there is a close correlation to the levels of alignment with those of the cohesion and fidelity. The level names are the same, but the descriptions of the levels are different.

Level	Description
Complete	The problem domain and the inheritance tree match.
Extraneous	Some classes in the inheritance tree do not match the problem.
Partial	There are missing constructs but nothing extra.
Poor	The inheritance tree poorly represents the design concern.

Complete Alignment

An inheritance hierarchy can be categorized as complete if there is a direct mapping between the classes in the inheritance tree and the constructs in the problem domain. Thus, alignment is not a characteristic of a single class in the hierarchy, but rather of the entire inheritance tree.

It is often impossible or, at best, problematic for a single developer to accurately categorize an inheritance tree as complete. Usually, a SME needs to be involved. Here, the developer would create a simplified class diagram including the entirety of the inheritance hierarchy. It would be simplified to only include class names (omitting attributes and operations) and arrows representing relations. The developer would then explain each class to the SME. From this, the SME should recognize the name and purpose of every single class in the hierarchy, mapping each to elements in the problem domain. If this can be done, then it can be said to have complete alignment.

Complete alignment is when there is a direct mapping between the classes in the inheritance tree and the constructs in the problem domain

To classify something as having complete alignment, one needs to start with the mental model of the SME. For example, if we were to ask the SME what the different types of lists that would be needed for our list management application, he might produce a concept map like the following:

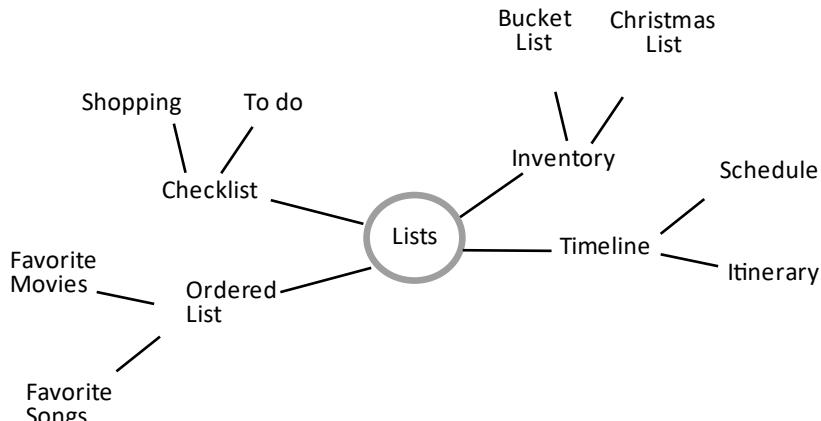


Figure 32.1:
Concept map

From this, the following inheritance tree would have complete alignment: there is a one-to-one mapping between the classes and superclasses of the inheritance tree to the concept map provided by the SME.

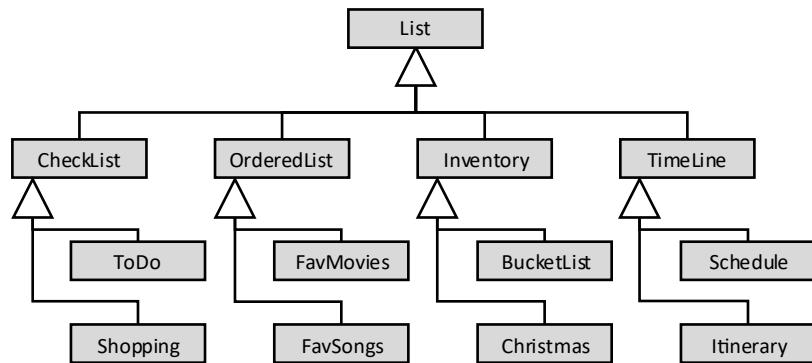


Figure 32.2:
Class diagram with
complete alignment

Extraneous Alignment

An inheritance hierarchy can be categorized as extraneous when the problem domain is completely modeled but there are also extra classes. If any extra class can be found in the inheritance hierarchy that the SME cannot recognize, then the inheritance tree can be classified as extraneous. In almost every case, this is the result of an extra base class that has been added for programming convenience as opposed to a need specified by the client.

Extraneous alignment is when there are classes in the inheritance tree that have no corresponding construct in the problem domain

There are several reasons why a designer may be tempted to introduce an extraneous class into the inheritance tree. One is that the overall system design may require a common base class. This is commonly the case for polymorphism and abstraction reasons. An example is the `java.lang.Object` generic base class from which all Java classes inherit. This `Object` class does not map to anything in the problem domain because it does not mean anything; it is there so all classes can be said to inherit off the same base class. In other words, there is a programmatic reason for the class, but not a problem domain reason for it.

Another common reason to introduce an extraneous class is to share attributes. To reduce or eliminate redundancy or duplicate code, a base class can be utilized for the sole purpose of containing member variables common to multiple derived classes. It is often difficult to name such a class because it doesn't represent anything. However, classes such as this reduce redundancy and simplify maintenance.

Back to our list application example, the designer may realize that the `Inventory` and the `Checklists` base classes have shared functionality. In effort to eliminate duplicate code, a new base class called `SemiOrdered` is added.

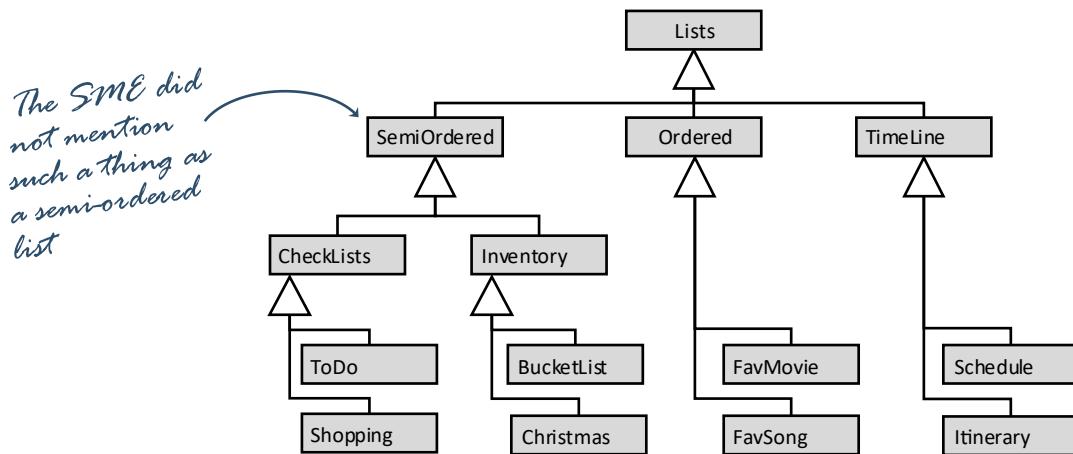


Figure 32.3:
Class diagram with
extraneous alignment

Extraneous alignment is not an indication of bad design; the problem domain is completely modeled, after all! Often a programmer will settle on an extraneous design because problems resulting from a nonaligned class are offset by advantages gained elsewhere. In other words, complete alignment is better and worth striving for, but extraneous is good enough.

Partial Alignment

Partial alignment is when there are missing constructs from the problem domain, but none are extra

requirements might be difficult to fulfill.

A partial inheritance tree is often the result of a designer not spending enough time understanding the problem domain before beginning the development process. It is easy to think you have “enough to go on” after a brief meeting with the SME. This tendency to “get things started” can often result in more work in the future.

It can also be very difficult to see the problem from the SME’s perspective. Their domain-specific knowledge is expressed in a foreign lingo. They often go into excruciating detail on unimportant things and then gloss over the things you really need to understand. The entire process can be a frustrating, humiliating, and time-intensive process. While all these things are true, it is time well spent. More time spent with the SME leads to better designs and avoids costly mistakes.

In rare cases, partial alignment can be a desirable property. Distinctions that are important to the SME might not be important to the design. For example, there is a long historical rift between the Hebrew and Arabic peoples. Their languages and writing systems, however, are more alike than they are different. The SME may swear that there is no relation and insist that they reside on opposite sides of the inheritance hierarchy. The astute (and brave) designer, however, may choose to group them together for a variety of sound technical reasons. Because the resulting inheritance tree is different than that of the SME, it has partial alignment.

Aside from extreme cases such as that mentioned above (with Hebrew and Arabic languages), partial alignment is usually an indication of poor design. Consider, for example, the following inheritance tree representing the various types of lists for our list class. Here, the designer made no attempt to capture the groupings that the SME provided. Because important design elements are not represented in the class diagram, we have partial alignment.

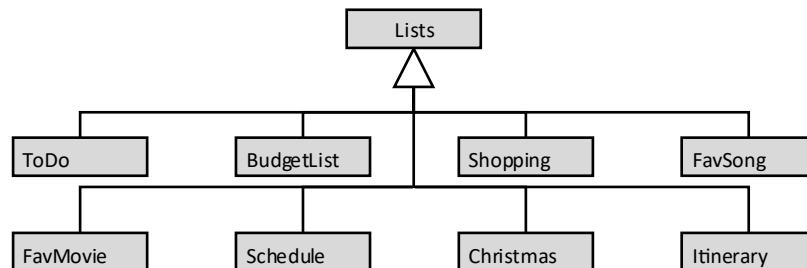


Figure 32.4:
Class diagram with
partial alignment

An inheritance hierarchy can be categorized as partial if constructs are missing from the problem domain, but none are extra. In almost all cases, partial alignment is a dangerous thing. It indicates there is missing functionality or that future

It is worthwhile to spend the time necessary to fully understand the problem domain

Poor Alignment

Poor alignment is when the inheritance tree poorly represents the problem domain

An inheritance hierarchy can be categorized as poor if the structure and members of the inheritance tree poorly represent the problem domain. This usually happens because there are extraneous classes in the inheritance tree as well as elements from the problem domain are missing. In almost all cases, poor alignment leads to a host of other problems, including lack of adaptability, redundancy, maintenance difficulty, and incomprehensibility.

Many years ago, when classes were first appearing in mainstream programming languages and developers started thinking in terms of objects, inheritance was considered the next “big thing.” Accounts of those early days have a common theme: initial excitement, a period of disillusionment, following by frustration working with impenetrable codebases. Inheritance trees were vast and complex. Classes were abstract and didn’t seem to do anything. Bugs were subtle, often spanning many classes and files. In short, it was incomprehensible. The underlying problem in most of these cases was poor alignment. Base classes were created for convenience, not because they represent something meaningful in the problem domain. This was widespread and gave object-oriented programming a bad name. Now, decades later, there are still programmers wary of inheritance!

One of the best tools for detecting poor alignment is to create a concept map. A concept map is simply a collection of ideas drawn on a sheet of paper or a white board with lines connecting them. A concept map does not need to be highly structured; it just shows which ideas are related. Sometimes the process of simply building a concept map with a collection of SMEs helps solidify murky ideas. Comparing the concept map with the class diagram can help detect alignment problems. In the figure below (32.5), we have a concept map on the left given to us by the SME. The class diagram on the right does not resemble this. There are extraneous classes (**Moveable** and **Stationary** do not appear in the concept map) and there are missing classes (**Rocks** and **Ships**). As a result, the design exhibits poor alignment.

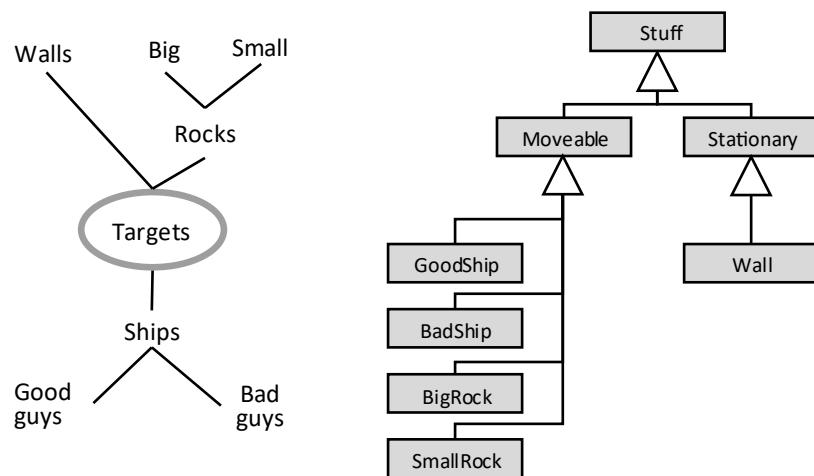


Figure 32.5:
Class diagram with
poor alignment

Related Metrics

The inheritance metric of alignment is highly related to the modularization metric of cohesion and the encapsulation metric of fidelity. As with adaptability, the similarities are expected as they are meant to capture the same type of thing, but the differences are necessary due to the different abstraction level in which they work.

Alignment and Cohesion

The modularization metric of cohesion is defined as how well a unit of software represents one concept or performs one task. This metric is focused on verbs, what a function does. Alignment focuses on nouns, what a class represents. As a reminder, cohesion has four levels of quality:

Cohesion Level	Description
Strong	The function performs one task completely
Extraneous	There exists extra functionality
Partial	The function does not complete a single task
Weak	Both extraneous and partial

The four levels of cohesion use similar labels as the four levels of alignment. However, their definitions are different. Cohesion level definitions are about tasks whereas alignment level definitions focus on the problem domain.

To illustrate why we cannot simply use cohesion to describe alignment, we will adapt the strong cohesion level to inheritance with “The class represents one thing completely.” Consider an abstract class **Money** with several derived classes (**Dollars**, **Euros**, etc.). Because **Money** is abstract (containing at least one pure virtual function), it does not “represent one thing completely.” In fact, no abstract class can ever be strongly cohesive; they all rely on derived classes to finish their job! However, **Money** does have an obvious and direct correlation in the problem domain so it can be said to have a strong alignment.

Cohesion focuses on individual functions or methods whereas alignment focuses on collections of classes

For this reason, cohesion and alignment are highly related metrics though adjustment to the definition and levels of alignment are necessary so they can be used to effectively measure inheritance design quality.

Alignment and Fidelity

The encapsulation metric of fidelity is defined as the suitability of a class in representing a design concern. This is highly related to the concept of alignment with an important distinction. Fidelity is about a single class wholly representing a design concern whereas alignment is about a base class representing a component of the problem domain. As a reminder, fidelity has four levels of quality:

Fidelity Level	Description
Complete	States of the class completely match the design concern
Extraneous	There are extra states but none missing
Partial	There are missing states but nothing extra
Poor	The class poorly represents the design concern

As with cohesion, the level labels of fidelity are the same as those of alignment, but their definitions are different. For a class to exhibit complete fidelity, it needs to fully represent a design concern. Note that base classes seldom if ever fully represent a design concern. Instead, they help derived classes do this. Their purpose in a program is completely different, requiring a different metric.

Fidelity focuses on individual classes (nouns) whereas alignment focuses on collections of classes (inheritance trees)

The important thing about alignment is that every element in the inheritance hierarchy must map to an aspect of the problem domain. It is a non-goal that the aspect be completely represented by a base class. In fact, it is undesirable in most cases.

Examples

Example 32.1: Poor Alignment

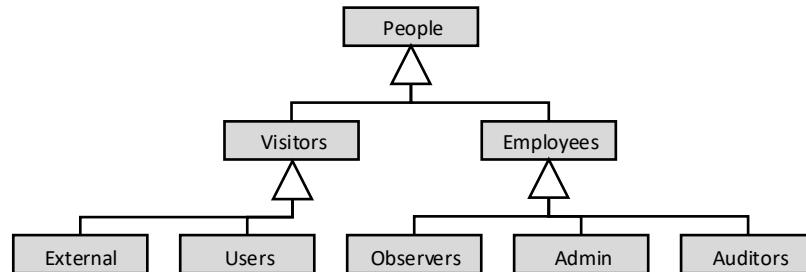
This example will demonstrate how to identify poor alignment.

Problem

Consider the following problem domain:

A financial program needs to model several types of people: those who have access to the system and those who are simply external contacts. Those who have access are subdivided between observers, users, administrators, and auditors.

Identify the level of alignment from the following class diagram:

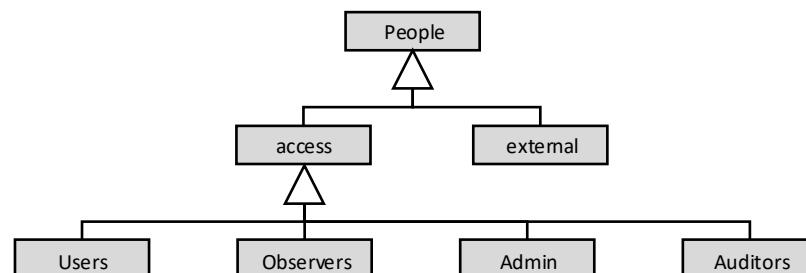


Solution

The problem description makes no distinction between visitors and employees. This might make sense in other contexts but does not align to the problem domain. This makes the alignment extraneous or poor.

The problem definition makes a distinction between internal and external contacts. This distinction is not present in the class diagram, making the design partial or poor. Thus, the overall level of alignment for this design is poor.

To improve the design, a hierarchy can be introduced to better match the problem definition. The following would be complete:



Example 31.2: Partial Alignment

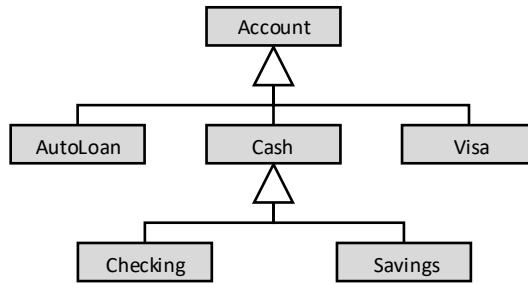
This example will demonstrate how to identify partial alignment.

Problem

Consider the following problem domain:

A bank offers several types of accounts: loans, cash accounts, and credit cards. There is currently one type of loan (auto) but the bank will offer more in the future. There are two types of cash accounts (checking and savings), and currently one credit card (Visa).

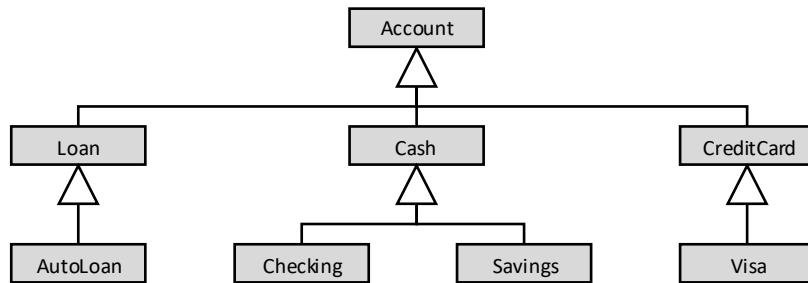
Identify the level of alignment from the following class diagram:



Solution

Every single class in this design is represented in the problem domain. This means that the level of alignment is partial or complete. Note that several items in the problem domain are mentioned but not in the class diagram: a loan and a credit card. This makes the design partial or poor. Thus, the overall level of alignment for this design is partial.

On the surface, the class diagram represents a good design for the application. Why make a base class **Loan** when there is only one child class (**AutoLoan**)? The problem is that the SME mentioned that there is currently only one type of loan and currently one type of credit card. This means that more will probably follow. A design more aligned with the client's needs would be the following:



Example 31.3: Extraneous Alignment

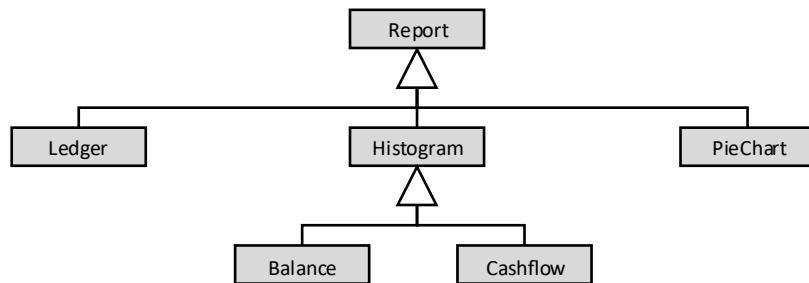
This example will demonstrate how to identify extraneous alignment.

Problem

Consider the following problem domain:

A financial program needs to display several types of reports representing account activity: a histogram of the account balance over time, a histogram of the cashflow over time, a ledger of the balance, and a pie chart displaying how money is spent. Interviews with the SME suggest that each of these reports are distinct.

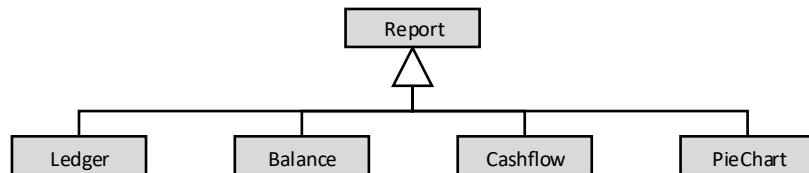
Identify the level of alignment from the following class diagram:



Solution

Every element mentioned in the problem description is present in the class diagram design. This makes the alignment either complete or extraneous. Note that the SME did not mention the notion of a histogram. Even though “a histogram of the account balance over time” and “a histogram of the cashflow over time” are separately mentioned, the SME went so far as to say, “each of these reports are distinct.” This means that the alignment of this design must be extraneous or poor. Therefore, the overall alignment level is extraneous.

A design that exhibits better alignment would be the following:



The question is: which design is superior? If we only look at the alignment metric, then the design without the `Histogram` base class would be better. However, if one considers redundancy (see Chapter 33) or adaptability (see Chapter 31), then the original design might be a better choice. Based on the provided information, we cannot tell at this point. More interactions with the SME will be required before making a final decision.

Example 31.4: Complete Alignment

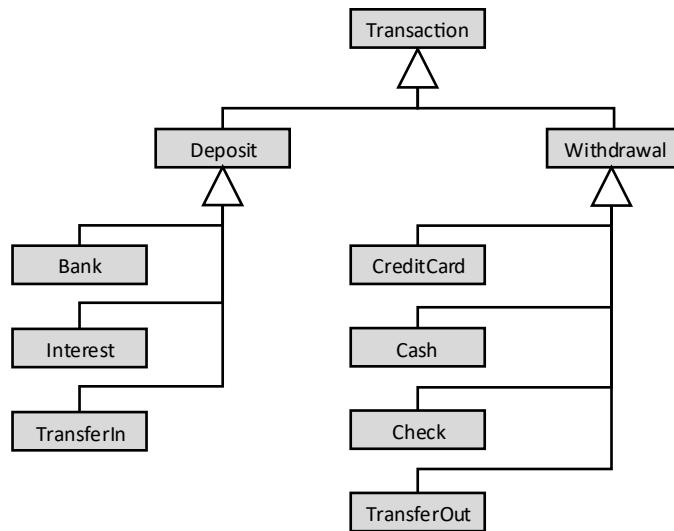
This example will demonstrate how to identify complete alignment.

Problem

Consider the following problem domain:

A financial program has a notion of a transaction. This can be either a withdrawal or a deposit. A withdrawal could be a cash withdrawal, a check, a transfer to another account, or a credit card transaction. A deposit can be a bank deposit, interest earned, or a transfer from another account.

Identify the level of alignment from the following class diagram:



Solution

Both the overall name of the various types of transactions (the **Transaction** base class) and the two main classifications of transaction (the **Deposit** and **Withdrawal** classes) are represented. Every flavor of transaction, as well as the classification each flavor belongs to, is also represented. Therefore, this alignment of this design is either complete or extraneous.

Every class in the class diagram has a counterpart in the problem definition. This means that the alignment of this design must be complete or partial.

Therefore, the only possible degree of alignment for this design is complete.

Exercises

Exercise 32.1: Alignment Definitions

From memory, define each of the adaptability levels:

Level	Name
Complete	
Extraneous	
Partial	
Poor	

Exercise 32.2: Alignment Characteristics

Characterize the level of alignment based on the scenario:

Scenario	Level of Alignment
There is an item in the class diagram that the SME cannot recognize.	
The design is neither sufficient nor free of redundancy.	
The developer can point to any item in the inheritance tree and the SME would be able to understand it. Also, nothing is missing.	
The developer thinks she has “enough to go on” but actually has an incomplete understanding of the problem.	
Something was added to the inheritance tree for the purpose of making the code easier to write, but it is not in the problem domain.	
The concept map and the class diagram look similar. If an item is on one, then it is also on the other.	

Exercises 32.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
It is never a good idea to use extraneous alignment.	
Alignment, fidelity, and coupling are similar concepts.	
Partial alignment almost always means poor design.	
Alignment is a characteristic of a single class in an inheritance tree.	
Software constructs should represent things in the problem domain.	
Partial alignment happens when one writes code before fully understanding the problem.	
The names of the cohesion and fidelity levels are the same.	

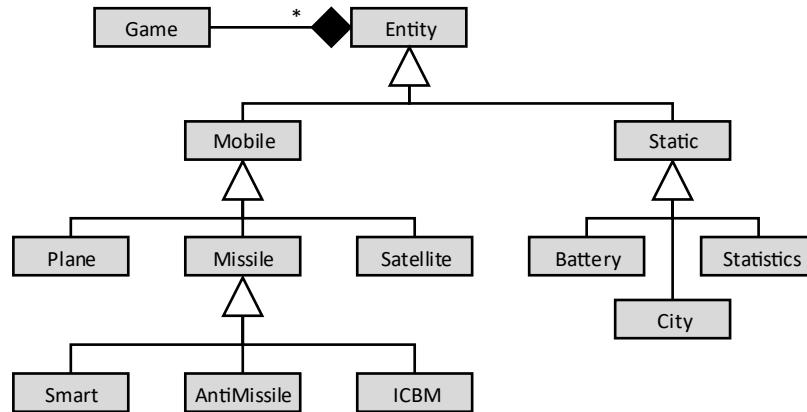
Problems

Problem 32.1: Alignment of Missile Command

Consider the following problem domain:

The 1980 game *Missile Command* consists of three types of entities: targets, assets, and inert objects. The targets are things that the player can shoot at, the assets are things the player attempts to protect, and inert objects are non-interactive elements designed to contribute to the look and feel of the game. Among the targets are incoming ICBM missiles, smart bombs, bomber planes, and satellites. Each can be destroyed by interceptor missiles controlled by the player. Assets include cities and the anti-missile batteries from which the interceptor missiles are fired. All assets can be destroyed by missiles and smart bombs. The inert objects include the score, number of missiles left, high score, and other status messages.

Identify the level of alignment of the following class diagram:



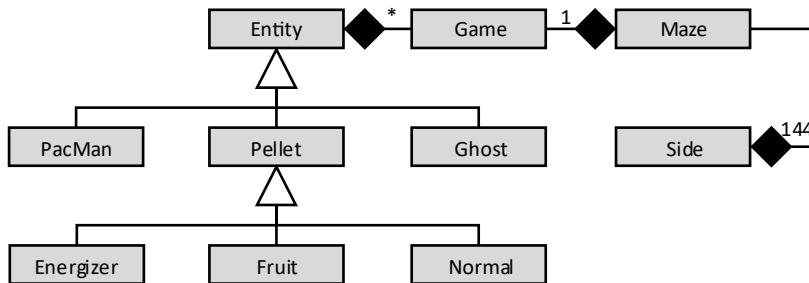
If the above design is not complete alignment, suggest a design that is. Finally, provide a justification as to why the alignment level of your design is complete.

Problem 32.2: Alignment of Pac-Man

Consider the following problem domain:

The 1980 game *Pac-Man* consists of a maze and several game entities: ghosts, pellets, and Pac-Man himself. The maze is a matrix of sides, each of which can be opened or closed. The ghosts have four flavors (Blinky, Pinky, Inky, and Clyde). These ghosts are in one of three modes: edible (blue), slow (when a power pellet is eaten), normal, and dead (when the ghost returns to the base). The pellets have three flavors: normal, energizer, and fruit.

Identify the level of alignment from the following class diagram:



If the above design is not complete alignment, suggest a design that is.

Problem 32.3: Design Dig Dug

Consider the following problem domain:

The 1982 game *Dig Dug* consists of enemies, Dig Dug, and a playing field. There are two types of enemies: Pookas and Fygars. Both can move through tunnels, ghost through solid ground, or be in various stages of inflation. The ground can be dirt, a tunnel, or a rock.

Create a class diagram exhibiting complete alignment. Justify why you think your design's alignment is complete.

Problem 32.4: Design Centipede

Consider the 1981 arcade game *Centipede*. Research how the game was played and provide a class diagram exhibiting complete alignment. Justify why you think your design's alignment is complete.

Challenges

Challenge 32.1: Financial Software

Create a class diagram for the following problem:

A personal finance system is designed to track the spending and budget of a college student. This system stores the user's data on a server, having both a web and a mobile user interface. The system can handle a variety of different types of accounts and produce a variety of reports.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of alignment.

Challenge 32.2: List Application

Create a class diagram for the following problem:

A personal list mobile application allows users to keep track of a variety of types of lists: to-do lists, grocery lists, life goals, and assignment due dates. The application allows users to add and remove lists, and even create reports created from data collected from many lists. Some reports include recently added items, recently completed items, and urgent.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of alignment.

Challenge 32.3: Video Game

Create a class diagram for the following problem:

A video game allows users to fly a ship through a 3D world filled with obstacles to be avoided. Points are added as levels are cleared and prizes are discovered. Points are deducted as walls and obstacles are hit, and as time elapses.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of alignment.

Redundancy exists when common elements exist in an inheritance tree. We can eliminate redundancy when these common elements are moved to superclasses.

Inheritance redundancy is when more than one instance of an attribute or operation exists in an inheritance tree

Imagine an application designed to help manage an individual's personal finances. A part of this program is the need to compute one's income tax burden. Rather than create a single function for this purpose, the programmer decides to copy and paste the tax code into many places in the application. Soon there are a dozen copies of this code in various parts of the program. A few weeks later, a bug is discovered in one of these copies. Being a diligent programmer,

she finds most of the other copies and fixes the bugs. Then the tax law changes, necessitating an update to the algorithm. Again, most of the copies are found and the changes are made. This continues for many rounds. In each case, most of the copies are found but a few are missed. After a while, each copy is slightly different. To make matters worse, our programmer just left the team, and with her all knowledge of where the various copies are located. We are left with a maintenance problem.

Redundancy is the source of many software maintenance woes. Testing is difficult, localizing bugs is tedious, adding functionality is expensive, and software is complex and difficult to understand. While functions are a great tool to minimize software complexity, inheritance gives the programmer perhaps the most powerful tool to combat redundancy. Note that redundancy is the only quality metric defined upside-down. All other software design metrics are expressed in terms of quality, whereas redundancy is expressed in terms of a problem; we seek to minimize redundancy rather than increase it!

Inheritance redundancy is the extent in which the common elements exist in an inheritance tree. If an attribute or operation exists in two related classes, then it should probably be moved into a common base class. When the inheritance tree fails to do this, we have redundancy.

Inheritance redundancy is when more than one instance of an attribute or operation exists in an inheritance tree

Levels of Redundancy

There are four levels of redundancy, each describing the existence and locality of redundant properties or operations in the inheritance tree.

Level	Description
Distinct	There are no duplicated attributes or operations.
Minor	Any duplication exists in non-adjacent classes.
Critical	Siblings have duplicate attributes or operations.
Redundant	Multiple instances of redundancy exist in the tree.

Distinct Redundancy

An inheritance tree can be classified as distinct when there are no instances of duplicate attributes or operations in the inheritance tree. There are two components of this definition: the attributes need to be distinct, and the operations need to be distinct.

Distinct means there are no instances of duplicate attributes or operations in the inheritance tree

We can say that there is no duplication of attributes when no member variables are defined more than once in the entire inheritance tree. Note that a single attribute could manifest itself in different ways, making it a little more difficult to validate the distinct attribute requirement than just looking for duplicate member variables. For example, consider a video game representing many obstacles for the user to fly around. There could be one set of obstacles which are destructible and some which are not. Within the destructible set, one collection can move (with dx , dy , and dz attributes representing movement along various axes). On the indestructible side, a collection can move here as well. This is represented differently (with speed, ϑ , ϕ representing linear speed and direction). There are no duplicate member variables in this tree, but there are duplicate attributes since you can compute (dx, dy, dz) from $(\text{speed}, \vartheta, \phi)$ using simple trigonometry.

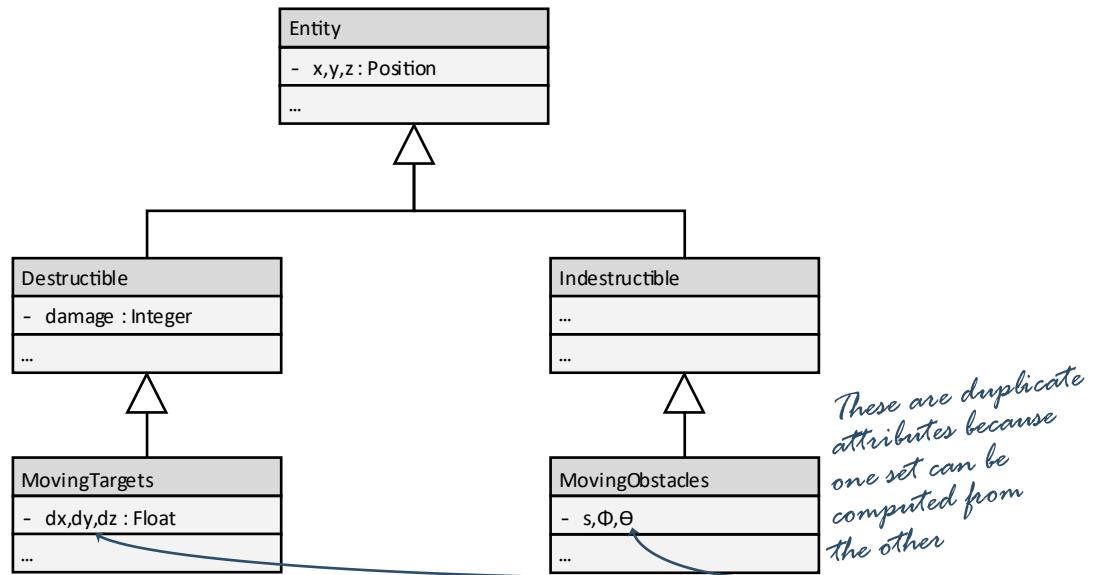


Figure 33.1:
Not distinct redundancy
due to duplicate attributes

As with duplicate attributes, duplicate operations can be difficult to detect. The most obvious case is when there are two instances of the same method in different parts of the inheritance tree. It is more subtle when a task appears in two different locations. This could be something as simple as a few lines of code to translate coordinates from one format to another. To find such duplications, it is necessary to carefully review every method in the entire inheritance tree. This can be quite an undertaking if the tree is large.

Minor Redundancy

Minor redundancy follows when any duplication of attributes or operations in the inheritance tree is in non-adjacent branches or at different levels

method may be duplicated. It is up to the designer to determine what exactly "minor" means.

An inheritance tree can be classified as having minor redundancy if there exists duplication of attributes or operations only in non-adjacent branches or at different levels. Note the subjective label "minor redundancy." This could mean a single member variable is duplicated or perhaps a small part of a

Minor redundancy can often be justified when the process of removing the redundancy degrades the design. An example of minor redundancy would be a method appearing in two different classes in far-separated corners of a very large inheritance tree. We can rectify the duplicate operations problem by creating a protected method in the base class. This design option has the advantage of eliminating the redundancy and all the maintainability issues that go along with it. There is a downside, however. There is now a function that is in scope to every single class in the inheritance tree. Most of the classes do not need this functionality. Furthermore, the base class is now more complicated than necessary. This base class now must be aware of implementation details of a far-removed derived class.

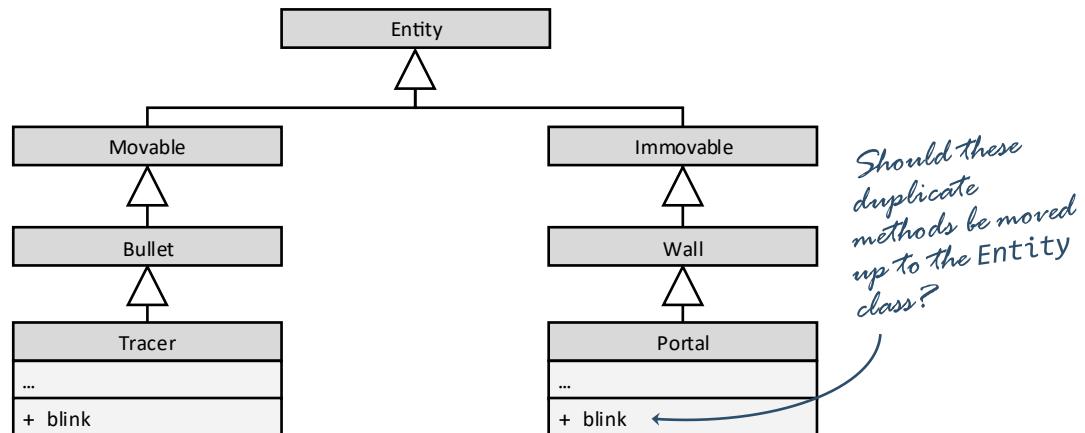


Figure 33.2:
Minor redundancy

Another example of when minor redundancy might be justified: consider the same large inheritance tree containing two instances of the same member variable in mostly unrelated classes. Of course, we can eliminate this redundancy by moving the member variables into the uppermost base class. There are some pretty serious consequences to this design alternative. First, the fidelity of the base class and all the derived classes is reduced. Every class that does not need the state represented by this new member variable will now show extraneous fidelity. Second, the scope of this variable has increased many fold. Instead of being visible only to the classes in which it is relevant, many classes in the inheritance tree will have access to an irrelevant variable. Finally, the base class will now be aware of implementation details of a derived class. All things considered, removing this redundancy is a bad design decision.

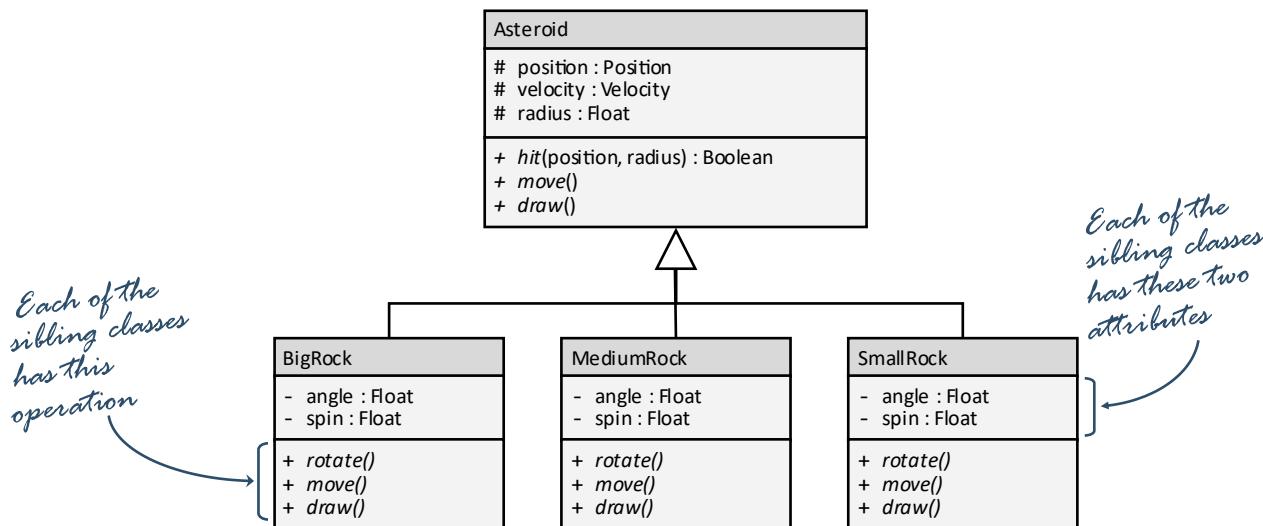
Critical Redundancy

An inheritance tree can be classified as critically redundant if sibling classes share duplicate attributes or operations. In situations such as these, the rest of the inheritance tree is not impacted by removing the redundancy.

Critical redundancy is when sibling classes share duplicate attributes or operations

Critical redundancy is usually easy to spot. When reviewing a class diagram, siblings in the inheritance tree should have distinct attributes. Similarly, when reviewing a class implementation, duplicate code in methods is a big tipoff that we have critical redundancy.

Consider the following code representing Asteroids in a 3D game. Each **Asteroid** has a position and a velocity. It also has a method to perform hit detection, code to move the asteroid according to the law of inertia, and code to draw the asteroid on the screen. Of course, all these common attributes and operations are in the base class. Now the author of a **BigRock** class decided to make this flavor of asteroid spin. This required two member variables and a private method. Another programmer came along and duplicated that code for the **MediumRock** and **SmallRock** classes. With the duplication of this operation and these attributes in sibling classes, we have critical redundancy.



In almost every case, the quality of the inheritance tree can be increased by percolating the duplicate code into the shared base class. Complications arise when only a subset of the siblings share the need for this functionality. In cases like this, is worthwhile to reduce or eliminate the redundancy at the expense of increasing the scope of the attribute or operation? Would it be better to create a new base class to capture this shared functionality, perhaps at the expense of alignment? It is the job of the designer to take all these things into account when making design trade-offs.

Figure 33.3:
Critical redundancy

Redundant

Redundant is when there are multiple instances of duplicate attributes and/or operations in the inheritance tree

An inheritance tree can be classified as redundant if no apparent attempt was made to reduce or eliminate redundancy in the inheritance tree. Usually this is the result of multiple instances of redundant attributes or operations, though a large

inheritance tree can be classified as having minor redundancy if there are only isolated and non-systematic examples of redundant attributes or operations.

A strong indication of redundancy is when the inheritance tree is extremely broad and shallow. In most cases, this is a bad design decision. Studies have shown that people have an easier time comprehending deep and narrow inheritance hierarchies rather than shallow and broad ones. For example, the author of our 3D game decided to make all game entities rely on the same base class. When this game is completed, there could be many dozens of game entities in this inheritance hierarchy. It is unlikely that there are no shared attributes or operations in this tree.

When the hierarchy is broad and shallow, there is probably redundancy

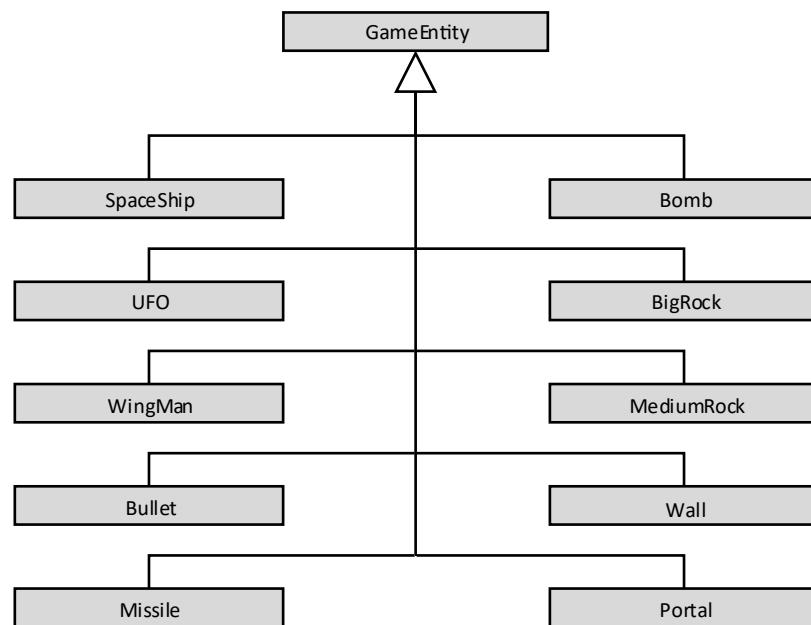


Figure 33.4:
Redundancy

Why would a programmer allow for such a poor inheritance design? Though there are many answers, the most common is this: classes were grouped under a common base class for the purpose of polymorphism. By grouping every member under a shared interface, the client can utilize every class without being aware of any of the specifics. On the surface, this appears to be a good design decision: polymorphism shields the client from a great deal of unnecessary complexity and implementation details. The problem is that there is unrealized potential. A large part of the program can be more adaptable, understandable, and maintainable with a more thoughtful inheritance tree design.

Related metrics

Unlike the adaptability and alignment metrics of inheritance design quality, there are no direct algorithmic, modularization, or encapsulation metrics which are analogous to redundancy. That being said, redundancy can have a positive or negative impact on several related metrics.

Redundancy Influences Maintainability

Redundancy is not a component of maintainability; however, redundancy can be a cause of maintainability problems. For example, consider a redundant operation in a large inheritance tree. A programmer makes a simple code change in this redundant operation not realizing that another instance exists. Here, the understandability is deducible at best; it is possible but certainly not straightforward to realize that another code change is required. Similarly, the malleability would be considered convoluted. More work will be required than should be necessary. In fact, the best fix would require some refactoring to remove the redundancy.

Maintainability is measure of how much time or effort is required to fix defects or make enhancements

Redundancy also has a negative impact on the testability of the software. While the developer certainly should be aware of the duplicate code, the tester should as well. All redundant code paths would have to be validated every time a single piece is changed. This would make it impossible for the tester to reduce the size of the test matrices by using equivalence classes.

Reducing Redundancy Should Not Impact Cohesion

If a method does one thing (and is thus cohesive), what happens when part of that functionality is moved to a method in a base class? Is that method more or less cohesive? Similarly, if part of a class is cored out into a base class, how does this influence the cohesion of the class?

Cohesion is a measurement of how well a unit of software represents one concept or performs one task

A function needs to accomplish one complete task to be cohesive. This is true whether the function does all the work itself or if it delegates some of the work to a subroutine. Thus, a method remains cohesive if it calls the corresponding method in a base class to do work which was common to sibling classes. Inheritance done right does not decrease cohesion.

Redundancy Can Influence Coupling

If a member variable is moved to a base class, then does this not result in larger scope and thus extraneous coupling? Isn't the whole point of modularization to reduce the complexity of information interchange between functions rather than add dependencies?

Coupling is a measurement of the complexity of the interface between units of software

Moving a common attribute into a base class does not necessitate an increase in scope for that variable. Scope can remain tight if the base class makes the member variable private, providing access only through protected methods. Generally, protected attributes should be managed carefully or avoided for coupling reasons. Thus, making code less redundant does not increase coupling if done right.

Designing Out Redundancy

Designing out redundancy can be summed up with the DRY principle: Don't Repeat Yourself. The best way to avoid redundancy in an inheritance tree is to start early in the design process. As the requirements are gathered and the initial drafts of the class diagrams are produced, take special note of the variants and invariants.

Variants

A variant is an attribute or operation that is unique to a given class. You could say that the variants are the defining characteristics of a given class. It is what makes the class distinctive, its reason for being. There should be a variant in every single class, be that a unique member variable or a different implementation of an operation. If there is no variant, then why does this class even exist? Why not use an existing class with the same functionality?

Best Practice 33.1 Percolate variants down in the inheritance hierarchy

If a base class contains elements unique to a subset of the classes, then move that uniqueness down the inheritance tree. This will reduce irrelevant attributes and operations, increasing the overall fidelity of the classes.

Invariants

An invariant is an attribute or operation that is shared between classes. Sometimes invariants are nothing more than a shared interface where each individual has a unique implementation. Invariants are the bases of collections and other organizational strategies. There should be invariants in every inheritance relationship. If two classes have completely disjoint attributes, operations, and interfaces, when why would anyone want to group them together under a single base class? What would that base class even look like?

Best Practice 33.2 Percolate invariants up the inheritance hierarchy

As common elements are found in classes, put the commonalities in a base class and make the rest child classes. This will minimize redundancy.

A Warning...

Imagine a designer working on a program with a hundred individual classes. With each class, she creates a class diagram and places them on an index card. Now, with all these index cards laid out on a table, she forms collections based on invariants. These collections become base classes, soon organized into an inheritance hierarchy that minimizes redundancy.

The problem with this approach is that the groupings and thus the base classes are unlikely to map to the problem domain. In other words, we probably have an alignment problem.

Be cautious about over-optimizing one design metric at the exclusion of the others. It is not uncommon for multiple metrics to be at odds with each other, pulling the design in different directions. There is no clear-cut answer as to what should be done in situations such as these. Take heart in this: mistakes are most commonly made because of ignorance and tunnel vision. In other words, engineers taking all factors into account will most often make the best trade-offs.

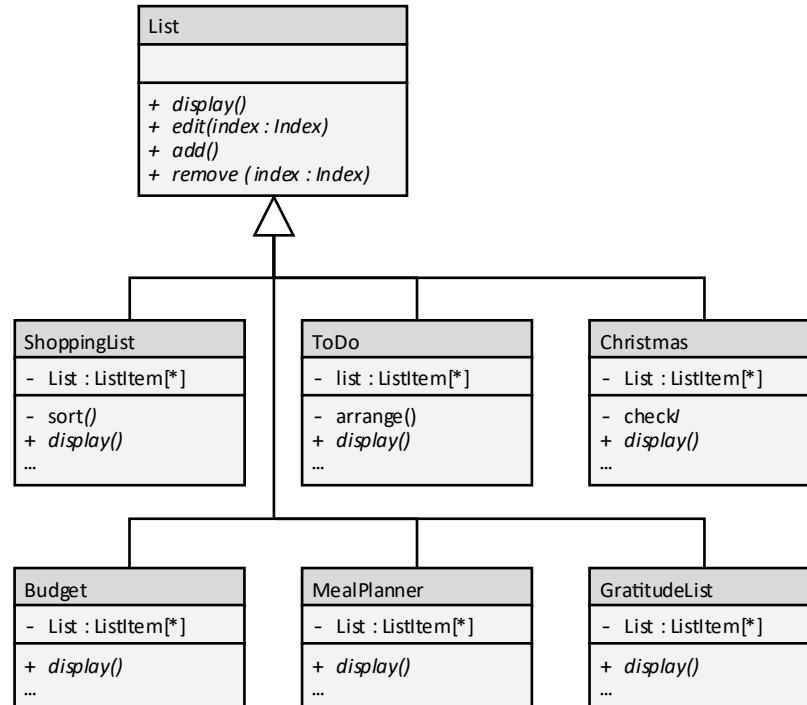
Examples

Example 33.1: Redundant

This example will demonstrate how to identify redundancy.

Problem

Identify the level of redundancy from the following class diagram:



Solution

The first clue that this design has poor redundancy is that the base class has no attributes. Further investigation reveals that all the derived classes have the same set of attributes. In other words, no attempt has been made to collect invariants into the base class.

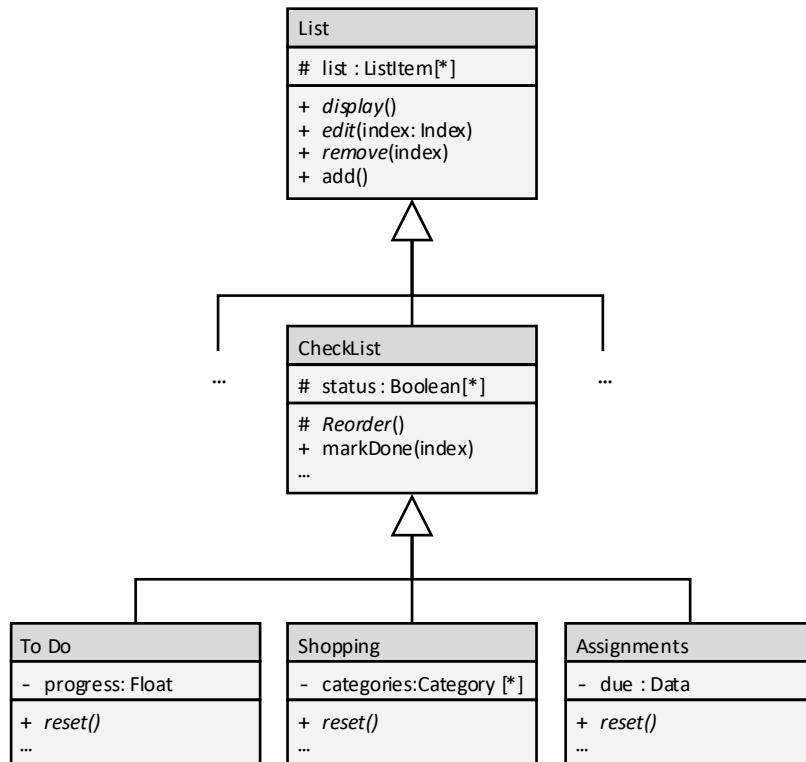
The second clue is that the purpose for this inheritance hierarchy is to collect classes for the purpose of polymorphism. All the classes share the same interface which are very generic. This appears to be the only motivation behind this design. With no apparent attempt was made to reduce or eliminate replication of attributes or operations, this can be classified as redundant.

Example 33.2: Critical Redundancy

This example will demonstrate how to identify critical redundancy.

Problem

Identify the level of redundancy from the following class diagram:



Solution

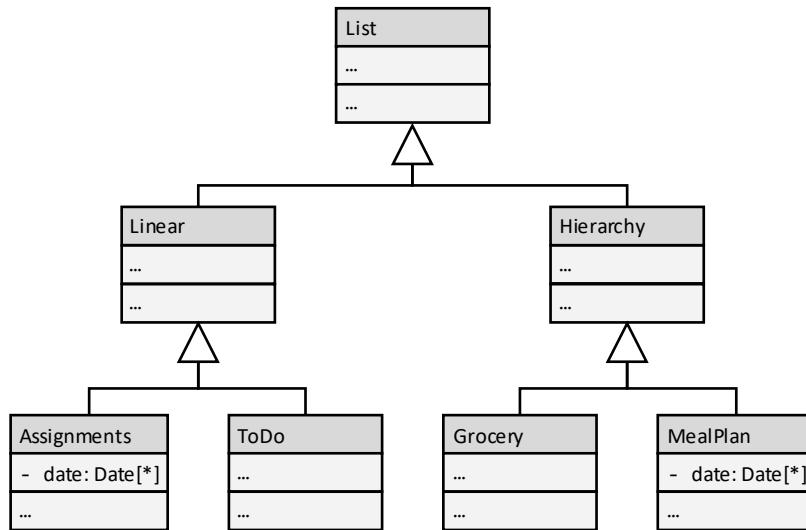
Note that the **List** base class contains the generic public interfaces as well as the collection of list items. The **CheckList** class contains two operations unique to all check lists: the ability to mark items as completed and to sort the checked-off items to the bottom of the list. At this point, there is no hint of redundancy in the design. The three derived classes contain what is unique about each list type: To-do lists have an indicator of progress, shopping lists organize items by category, and assignments have due dates. However, all three have the same redundant method: **reset()**. Since all three copies presumably do the same thing (uncheck each item in the list) and since all of these are in sibling classes, this can be classified as critical redundancy. It would be a better design to move that invariant into the **CheckList** base class.

Example 33.3: Minor Redundancy

This example will demonstrate how to identify minor redundancy.

Problem

Identify the level of redundancy from the following class diagram:



Solution

Observe how **Assignment** and the **MealPlan** derived classes both have the member variable `dates`. The **Assignment** class uses this for due dates whereas the **MealPlan** uses this for when the meal will be prepared. The uses for these two variables are different, but similar enough that they can share the variable. Note that the other classes in the hierarchy do not need this member variable. Because duplication exists and it is not widespread or contained in siblings, this is minor redundancy.

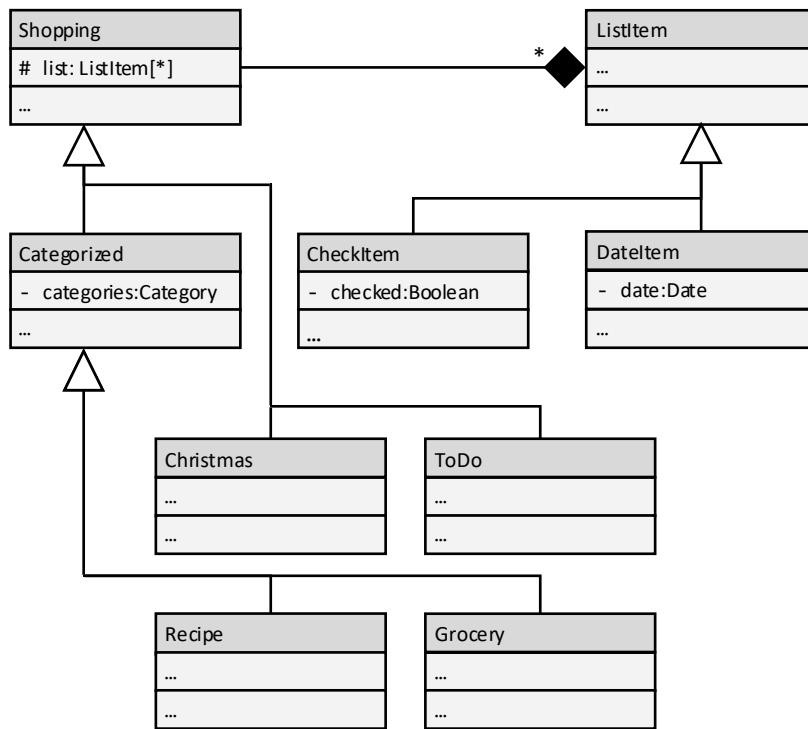
The next question is whether this should be removed. Since the least common ancestor is the **List** base class, it could be moved to that location. This would mean that most of the classes in the system would have an unneeded variable. Alternatively, the entire hierarchy could be reordered so that **Date/NonDate** classes would derive off of **List** rather than **Linear/Hierarchy**. Based on the provided information, it is not clear which design would be best.

Example 33.4: Distinct Redundancy

This example will demonstrate how to identify distinct redundancy.

Problem

Identify the level of redundancy from the following class diagram:



Solution

Though it is difficult to tell because the above class diagram is heavily abbreviated for the sake of space, there are no duplicate attributes in the inheritance tree. All the unique member variables are in the **List**, **Categorized**, **DateItem**, and **CheckItem** classes. This supports the assertion that this design is distinct.

With a complete absence of methods in the above class diagram, it is even more difficult to tell if there are duplicate operations. However, all the code pertaining to managing categories of lists is in the **Categorized** base class. Similarly, all the date code is in the **DateItem** class and the checkbox code is in the **CheckItem** class. This also supports the assertion that this design is distinct.

Exercises

Exercise 33.1: Redundancy Definitions

From memory, define each of the redundancy levels:

Level	Name
Distinct	
Minor	
Critical	
Redundant	

Exercise 33.2: Redundancy Characteristics

Characterize the level of redundancy based on the scenario:

Scenario	Level of Redundancy
In the whole inheritance tree, a single duplicate attribute was found in classes on opposite sides of the tree.	
The developer split one class into three: a base class and two derived classes. One member variable is common between the derived classes.	
Though the entire inheritance tree was carefully checked, not a single instance of duplication was found anywhere.	
The developer copied a dozen lines of code from one method and pasted it into another method on a different part of the inheritance tree.	
A private method used to verify the application state is copied into a few dozen classes in an inheritance tree.	
A private method in one class was copied into another class on a different part of the inheritance tree.	

Exercises 33.3: Fact or Fiction

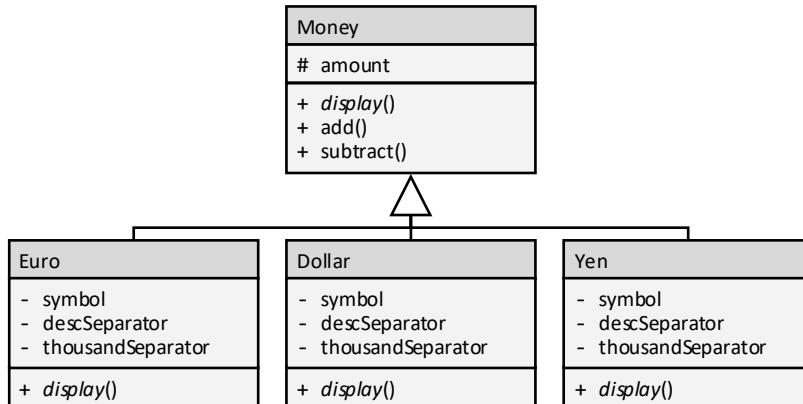
For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
It is a best practice to move variants up to the base class in an inheritance tree.	
Less redundant code is more cohesive.	
Over-optimizing on a single design metric can result in a bad design.	
Redundancy and maintainability are essentially the same metrics.	
Moving duplicate attributes to a base class results in worse coupling.	
Redundant code is more difficult to test.	
It is a best practice to move invariants up to the base class in an inheritance tree.	

Problems

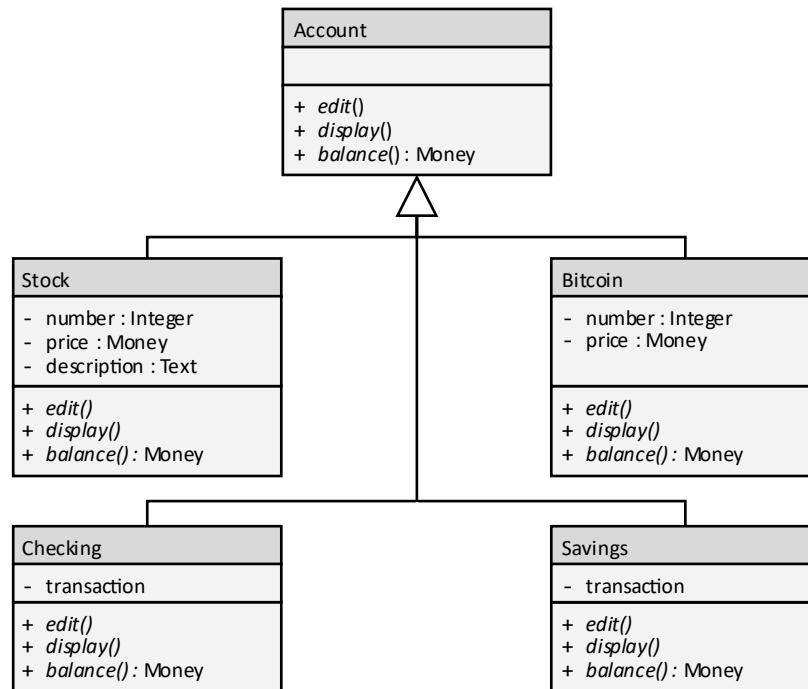
Problem 33.1: Identify Level of Redundancy

Identify the level of redundancy in the following class diagram. If the design is not distinct, suggest a design which is.



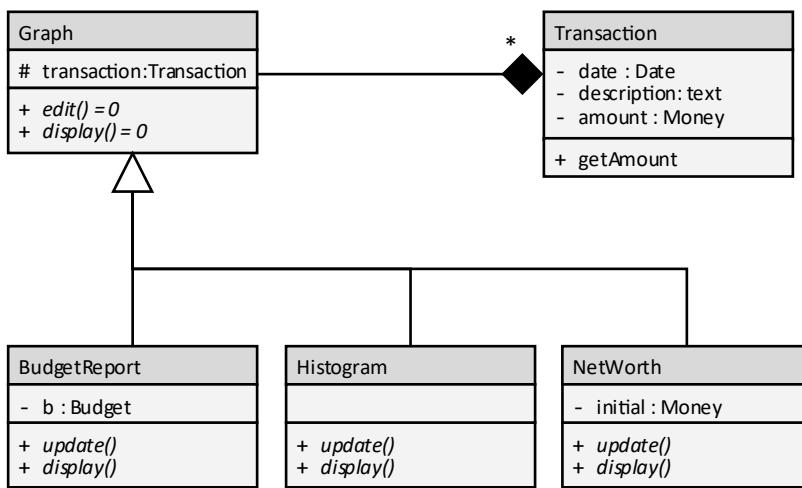
Problem 33.2: Identify Level of Redundancy

Identify the level of redundancy in the following class diagram. If the design is not distinct, suggest a design which is.



Problem 33.3: Identify Level of Redundancy

Identify the level of redundancy in the following class diagram. If the design is not distinct, suggest a design which is.



Problem 33.4: Design a Class Diagram

Design an inheritance hierarchy exhibiting distinct redundancy to satisfy the following problem definition. Justify your answer.

A single transaction in a financial application corresponds to an event which influences the balance. There are many types of transactions. A deposit will put cash into a checking or saving account. A withdrawal will remove cash out of a checking or savings account. A purchase will add several shares of stock at a given price. A sell will remove several shares at a given price. All transactions have several dates: when the transaction was initiated, when it was completed, and when it was reconciled. Transactions also have a status: Pending, Cleared, Reconciled, Voided.

Problem 33.5: Design a Class Diagram

Design an inheritance hierarchy exhibiting distinct redundancy to satisfy the following problem definition. Justify your answer.

There are several types of users on the system. There is the administrator, with unlimited access to all the system's resources. There is an auditor, able to view everything on the system but make no changes. There are normal users who have complete access to their accounts but nothing else; in other words, there is a list of accounts to which they have access. Finally, there is a restricted user. This is like a normal user but has read-only access to a fixed set of accounts. Every user has a username, a friendly name, and a password.

Challenges

Challenge 33.1: Financial Software

Create a class diagram for the following problem:

A personal finance system is designed to track the spending and budget of a college student. This system stores the user's data on a server, having both a web and a mobile user interface. The system can handle a variety of different types of accounts and produce a variety of reports.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of redundancy.

Challenge 33.2: List Application

Create a class diagram for the following problem:

A personal list mobile application allows users to keep track of a variety of types of lists: to-do lists, grocery lists, life goals, and assignment due dates. The application allows users to add and remove lists, and even create reports created from data collected from many lists. Some reports include recently added items, recently completed items, and urgent.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of redundancy.

Challenge 33.3: Video Game

Create a class diagram for the following problem:

A video game allows users to fly a ship through a 3D world filled with obstacles to be avoided. Points are added as levels are cleared and prizes are discovered. Points are deducted as walls and obstacles are hit, and as time elapses.

Your class diagram should have between 10 and 20 classes. Based on this class diagram, characterize the level of redundancy.

Chapter 34 Debugger

Though it is possible to work in large and codebases without a debugger, it is vastly more difficult. A debugger is a “power-up,” enabling programmers do things they could not do otherwise.

A debugger gives the programmer a powerful suite of tools making large and complex projects more understandable

Small programs consisting of a couple classes and a few dozen functions are relatively easy to understand and debug. The programmer can know all aspects of the design and quickly attribute anomalous behavior to the responsible line of code.

When working on medium size projects (dozens of classes and tens of thousands of lines of code), large projects (hundreds of classes and hundreds of thousands of lines of code) or gargantuan projects (tens of thousands of functions, hundreds or thousands of files, and tens of millions of lines of code), it is impossible for any one person to know an entire project. How can anyone cope with this level of complexity? Fortunately, we have a powerful tool at our disposal: debuggers.

A debugger is a tool that governs the execution of a program for the purpose of providing the programmer insight how the program is behaving. Though there are many debuggers on the market, each offering its own set of features, most provide the same core functionality:

Functionality	Description
Step	The ability to control execution in discrete increments
Watch	The ability to monitor and adjust the state of variables
Breakpoint	The ability to stop execution at a specific moment
Call Stack	The called functions from main to the current location
Disassembly	A view into what the CPU sees

It does not matter which debugger you choose to use. The most important thing is that you pick one and master it. A developer not fully understanding his or her tools is like a mechanic not knowing how to use a wrench or a truck driver not understanding how the transmission works. Make it your job to master it!

Step

Stepping is the process of controlling the execution of a program in discrete, programmer-controlled increments. During a typical program execution, the program proceeds from the beginning to the end in rapid succession, much too fast for a human to follow what is happening. Debuggers allow the programmer to slow the action down in such a way that only one statement is executed at a time. There are five main stepping features:

Stepping Feature	Description
Run	Execute the program, without pausing or stepping, from the current location.
Step Over	Execute to the next statement in the current function. If it is a function call, complete that function and return.
Step Into	Execute the next statement. If that statement is in a function call, then step into that function.
Step Out	Run to the end of the current function and pause at the point in the caller where the current function is called.
Run To	Continue execution until a specified statement is reached. Execution is paused at that point.
Set Next Statement	Modify the regular execution of the program so the specified statement is the next one executed.

During a single debugging session, each of the stepping features is used many times.

Best Practice 34.1 Memorize the shortcut keys to each of the stepping features

Just about every debugger has convenient icons for these common stepping features. While they are useful, it is better to use the shortcut keys. Unfortunately, there is no standard key assignment for debuggers; each seems to use its own mapping. Fortunately, just about every debugger allows you to customize the shortcut keys for these common stepping features. Therefore, if your new debugger does not work like your old favorite, you can just remap the keys.

Run

Run is the default state for executing any program. When you execute a program from the command line or hit the “play” button in the debugger, then “run” is effectively executed.

Best Practice 34.2 Use “run” when you expect the program to crash

When finding bugs without the benefit of a debugger, program crashes can be very difficult to localize. You know that the program crashed, but are given no clues as to where! Debuggers change all that. They pause execution at the point of the crash to show you where it occurred and the state of the program at that moment in execution. The same is true with asserts; the debugger will stop the program at the failed assert. The run command is commonly used to “fast-forward” to that point in a program.

Best Practice 34.3 Use “run” when you want to verify the normal operation of the program

Imagine the scenario when you are working on a bug in the debugger and have localized the cause of a crash. Once the offending line has been localized, you could fix the bug, recompile, and rerun the program to make sure everything after the crash is working correctly. This is a time-intensive task. A debugger streamlines this process. It is possible to step over the offending line of code and select “run.” This will continue execution as if the crash never happened. This way, more than one bug can be identified and fixed before having to recompile the project. This can be a real time saver, especially on large projects where recompilation could take many minutes.

Step Over

Step over is the process of executing the next statement in a single function. If the next statement is another function call, then that function (and all its children) is executed and control will return to the original function. Note that if the next statement is not a function, then step over and step into behave the same.

The screenshot shows a C++ code editor with the following code:

```
C++  
int main()  
{  
    Board board = readBoard();  
    interact(board);  
    saveBoard(board, getFilename());  
    return 0;  
}  
  
void interact(Board & board)  
{  
    bool done = false;  
    Coordinate coordinate;  
    Value value;  
  
    while (!done)  
    {  
        switch(getNextCommand())  
        {  
            case 'e':  
                coordinate.prompt();  
                board.edit(coordinate);  
                break;  
                ... code removed for brevity...  
            }  
        }  
    }  
  
Coordinate::prompt()  
{  
    cout << "> ";  
    string input  
    cin >> input;  
    ... code removed for brevity...  
}
```

A blue arrow points from the handwritten note to the line of code: `board.edit(coordinate);`. The line is highlighted with a blue background. The line number 438 is also highlighted in blue.

After step-over,
execution will be
paused here

Figure 34.1:
An indication of where
execution will be paused
next after step over

In the above code, execution has been paused at line 438. This means that line 438 has not yet been executed. When the step over command is issued, all the code in `Coordinate::prompt()` will be run and then execution will pause at line 439. In other words, unless execution is paused at the end of the function, step over will remain in the current function.

Best Practice 34.4 Use step over to better understand how a single function works

Step over clarifies the behavior of a single function; each step over will keep execution in the same function until the end is reached. It is also helpful in that it is hard to get lost. Unlike other stepping functions, execution does not jump between functions.

Step Into

Step into is the process of executing the next statement in the execution chain, even if that next statement is within a different function. In the case where the next statement is simple (IF statement, FOR loop, or assignment), then step over and step into perform the same action. However, when the next statement is a function call, then the behavior is different. Step over will call that function and return back to the parent function, whereas step into will pause at the beginning of the subroutine.

C++

```
int main()
{
    Board board = readBoard();
    interact(board);
    saveBoard(board, getFilename());
    return 0;
}

void interact(Board & board)
{
    bool done = false;
    Coordinate coordinate;
    Value value;

    while (!done)
    {
        switch(getNextCommand())
        {
            case 'e':
                coordinate.prompt();
                board.edit(coordinate);
                break;
                ... code removed for brevity...
        }
    }
}

Coordinate::prompt()
{
    cout <> "> ";
    string input
    cin >> input;
    ... code removed for brevity...
}
```

419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
507
508
509
510
511
512
513
514
515
581

*After step-into,
execution will be
paused here*



Figure 34.2:
*An indication of where
execution will be paused
next after step into*

In the above example, we start at line 438 which happens to be a call into the `Coordinate::prompt()` method. The next line of code to be executed is the `cout` statement inside that method. Step into will execute that function call and pause before the first statement in the function is executed. In this case, it will be line 513.

Best Practice 34.5 Use step into to view every single statement as it is executed, skipping nothing

Step into gives the programmer a much more detailed understanding of what the program is doing than step over. Step over may skip the part of the program containing the bug, giving the programmer no insight as to where the problem lies.

Step Out

Step out is the process of running to the end of the current function and pausing in the caller. If execution is already sitting at the end of a function, then step over, step into, and step out are the same. Otherwise, step out is just like a succession of step overs.

The diagram shows a snippet of C++ code in a code editor. A handwritten note on the left says "After step-out, execution will be paused here" with an arrow pointing to line 438. The code is as follows:

```
C++
int main()
{
    Board board = readBoard();
    interact(board);
    saveBoard(board, getFilename());
    return 0;
}

void interact(Board & board)
{
    bool done = false;
    Coordinate coordinate;
    Value value;

    while (!done)
    {
        switch(getNextCommand())
        {
            case 'e':
                coordinate.prompt();
                board.edit(coordinate);
                break;
                ... code removed for brevity...
        }
    }

    Coordinate::prompt()
    {
        cout << "> ";
        string input;
        cin >> input;
        ... code removed for brevity...
    }
}
```

Line numbers are listed on the right side of the code.

Figure 34.3:
An indication of where
execution will be paused
next after step out

In the above example, execution was paused in `interact()` at line 438. Note that `interact()` contains an event-controlled loop constituting the bulk of the code for the entire program. When the programmer issues the step out command, then the entirety of `interact()` is executed and control returns to the caller. In this case, `main()` called `interact()` at line 422. Therefore, execution will be paused at line 423 in `main()` when the step out command is completed.

Best Practice 34.6 Use step out when you want to return to the caller

Step out is useful if the programmer just executed step into but learned that function is not where the bug is to be found. This is an easy way to return to the function of interest without having to repetitively step through an uninteresting function.

Run To

Run to is the process of continuing execution up to a point indicated by the programmer. This point may be in the current function or in another.

The diagram shows a C++ code editor window with the following code:

```
C++
int main()
{
    Board board = readBoard();
    interact(board);
    saveBoard(board, getFilename());
    return 0;
}

void interact(Board & board)
{
    bool done = false;
    Coordinate coordinate;
    Value value;

    while (!done)
    {
        switch(getNextCommand())
        {
            case 'e':
                coordinate.prompt();
                board.edit(coordinate);
                break;
                ... code removed for brevity...
        }
    }

    Coordinate::prompt()
    {
        cout << "> ";
        string input
        cin >> input;
        ... code removed for brevity...
    }
}
```

Annotations in the code editor:

- A blue arrow points from the text "After run-to, execution will be paused here" to the line `case 'e':`.
- A blue arrow points from the handwritten note "In this case, the cursor is here" to the line `coordinate.prompt();`.
- A blue arrow points from the line `case 'e':` to the line `coordinate.prompt();`.

Line numbers on the right side of the code:

- 419
- 420
- 421
- 422
- 423
- 424
- 425
- 426
- 427
- 428
- 429
- 430
- 431
- 432
- 433
- 434
- 435
- 436
- 437
- 438
- 439
- 440
- 507
- 508
- 509
- 510
- 511
- 512
- 513
- 514
- 515
- 581

Figure 34.4:
An indication of where
execution will be paused
next after run to

In the above example, the programmer's cursor is in line 433 and execution is paused at line 438. When the programmer issues the run to command, then the remaining code in the CASE label is executed, as is the remaining code in the WHILE loop. When execution reaches line 433, it will pause at that point.

Run to offers no power that step over, step into, and step out do not already provide. In fact, many programmers never use the run to command. However, it is very useful in several scenarios.

Best Practice 34.7 Use run to in a loop to quickly see what has changed with each iteration

With the cursor set on a loop, run to is an easy way to see how the loop behaves. When the loop body consists of a dozen or more lines of code, then step over can be very tedious. Repeated run to commands will pause once per loop iteration.

Best Practice 34.8 To quickly skip over a block of code or even a loop, use run to

Usually only a few statements are relevant when fixing a bug. Run to makes it easy to skip to these statements without getting bogged down in irrelevant details.

Set Next Statement

Step over, step into, step out, and run to all behave essentially the same: continue normal execution until a certain point in the program is reached where execution is paused. Set next statement works differently. It interrupts the normal flow of the program to allow the programmer to specify a different order of execution.

The very next statement to be executed is this after set-next.

```
C++
int main()
{
    Board board = readBoard();
    interact(board);
    saveBoard(board, getFilename());
    return 0;
}

void interact(Board & board)
{
    bool done = false;
    Coordinate coordinate;
    Value value;

    while (!done)
    {
        switch(getNextCommand())
        {
            case 'e':
                coordinate.prompt();
                board.edit(coordinate);
                break;
                ... code removed for brevity...
        }
    }

    Coordinate::prompt()
    {
        cout <> " ";
        string input
        cin >> input;
        ... code removed for brevity...
    }
}
```

In this case, the cursor is here

Figure 34.5:
An indication of the next statement to be executed after set next statement

Note that even though execution was paused at line 438, the very next line of code to be executed will be 433. The function `Coordinate::prompt()` will not be called nor anything else in that case label. Set next statement will skip the remaining code in the CASE label and the loop and pause before line 433 is executed. In other words, the IP is altered without any code being executed.

Best Practice 34.9 Use set next statement to explore “what-if” scenarios

Imagine that the `Coordinate::prompt()` function contained a crashing bug blocking further execution. Set next statement can skip this code and see what would otherwise happen. It enables the programmer to explore the body of an IF statement even though the Boolean expression is always false, jump out of an infinite loop, or even execute the same line of code several times with different input values. This gives the programmer great power to see how code will behave in a wide variety of situations without having to stop and recompile the code.

Watch

Watch is a family of features allowing the programmer to view and even alter the state of variables in the running program. There are several flavors of watch: local variables, programmer-specified, and the immediate window.

Watch Local Variables

The most commonly used watch window displays the variables are in scope at a given moment in execution. It will also show the values of the variables. The local watch window, coupled with the stepping features, allows the programmer to get the same information as would be gleaned from a program trace. For example, consider the following code where execution is paused at the highlighted line:

```
Python
sum = 0
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]

# compute the sum of the primes
FOR prime IN primes:
    sum += prime
print(sum)
```

Figure 34.6:
Execution is paused
inside this loop on
the fifth iteration

At this point, there are three variables in scope: `sum`, `primes`, and `prime`. The local watch window would report the following on the fifth iteration:

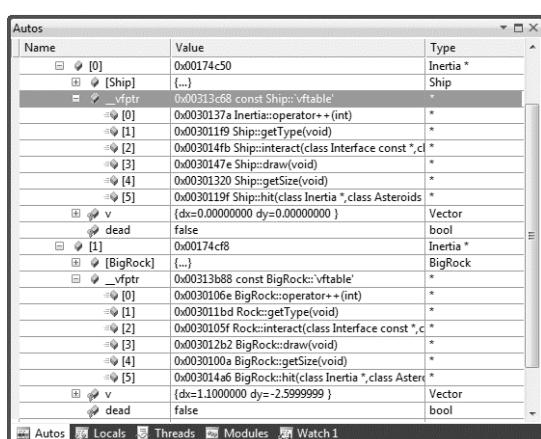
Variable	Value
sum	17
primes	[2, 3, 5, 7, 11, ...]
prime	11

*The values of
the variables
just before the
line is executed*

Figure 34.7:
The state of the watch
window from the code in
Figure 34.6

Figure 34.8:
A screenshot of an actual
watch window revealing
the data type of variables

Many debuggers allow the programmer to edit the value of variables in the watch window. This enables running multiple what-if scenarios to see how the code would behave in a variety of situations. This can be thought of as an ad hoc unit test: the pre-condition is set in the watch window; a collection of statements are executed through set next statement and run to; and the result can be viewed.



Another powerful feature of the watch window is that it can indicate a variable's data type. This is particularly helpful when debugging an inherited variable or a polymorphic variable. In figure 34.8, we have an array of polymorphic objects. The first one appears to be a `Ship` while the second appears to be a `BigRock`, both of which inherit from `Inertia`. Notice that the `BigRock` class defines its own `draw()`, `getSize()`, and `hit()` virtual functions, but inherits `getType()` and `interact()` from its base class `Rock`. The watch window allows the programmer to verify that methods are inherited and that objects are populated in an array the way that was expected. This type of information would be very difficult to glean any other way.

Programmer-Specified Watch

Most debuggers enable the programmer to specify the set of variables to be monitored in a special watch window. The programmer can enter the variables to be watched even if they are not currently in scope. This enables a very powerful debugging technique.

A programmer-specified watch simplifies the process of tracing a value through multiple variables and functions

Frequently bugs are the result of a value being different than the programmer expects. Usually this value underwent several transformations before it arrived in the erroneous state. If a complete and accurate DFD exists of the program, it is possible to trace the value to its origin. Say, for example, that a given value existed in a dozen variables at various points in execution. One could enter each of those variable names in the programmer-specified watch window and observe the values as execution passed through several functions. This way, the value in question can be monitored without the distraction of other variables that have nothing to do with the variables of interest.

Immediate Window

The immediate window is an edit control that lets the programmer enter the name of a variable and immediately see the value. On the surface, this may seem to be an inconvenient version of a watch window. However, the immediate window also allows the user to also enter expressions and even function calls.

The screenshot shows a portion of a C program in an IDE. The code is as follows:

```
c
int main()
{
    char board[8][8];
    displayBoard(board);
    interact(board);
    saveGame(board);

    return 0;
}
```

A handwritten note on the right side of the screen says: "We forgot to call initialize()". A blue arrow points from this note to the line of code "displayBoard(board);".

Figure 34.9:
Code with a bug: a
missing function call

In the above example, we have code to play the game of chess. When we call `displayBoard()`, a mistake is readily apparent: the chessboard is not filled with chess pieces neatly in their rows! Instead it is filled with garbage. To figure out what is going on, we pause execution just before `displayBoard()` is called. To our surprise, the board array is not initialized. Clearly, we forgot to call `initialize()`. Fortunately, this can be easily fixed. In the immediate window, we type: "`initialize(board);`" Even though that function call does not exist in the codebase (but fortunately the function itself does!), the immediate window will execute that function as if it was. We can now see that the `board` array is filled with the expected values, and we can run to continue normal operation of the program.

The immediate window facilitates running through multiple “what-if” scenarios without recompiling the code

The immediate window is a powerful tool giving the programmer the ability to run through multiple scenarios in the program (including code not existing in the codebase) without having to recompile the project. This way, the programmer can be confident that a solution will work even before the code is compiled.

Breakpoint

When debugging code, the programmer often needs to get to a specific location in the execution. While it is always possible to step-to that location, that may be a long and tedious process. To automate this task, it is possible to set a breakpoint.

A breakpoint is a tool enabling the programmer to pause execution at certain points in the program

A breakpoint is a marker set in the code indicating that execution is to stop when the marker is reached. In many ways, the “run to” stepping feature is just like a breakpoint. The difference is that breakpoints give the programmer a great deal more flexibility how execution can be halted. For example, the programmer can introduce multiple breakpoints and even have breakpoints pause execution at certain conditions.

Standard Breakpoint

A standard breakpoint simply halts execution when the instruction pointer reaches a given line of code. The user can set any number of breakpoints in the code. Say, for example, the programmer wants to monitor all the places in the code where data are written to a file. The programmer can locate all these locations and set breakpoints. There may be several dozen breakpoints in this scenario! When the program is executed, it will be halted at each file operation, allowing the programmer to see the sequence of things being written.

Best Practice 34.10 Set breakpoints at all the points of interest before executing the code

When debugging an object-oriented program, it is often the case that the bug can be localized to a single class. The problem is that you don’t know who is using the class or what methods are being called. To discover these things, you can set breakpoints at the beginning of all the public methods in the class. When the program is executed, this will tell you exactly who is using the class and in what way.

Best Practice 34.11 Save valuable breakpoint sets

Imagine this scenario: you have just finished tracking down a particularly nasty bug that involved setting dozens of breakpoints in the codebase. Now that the task is complete, you naturally clear your breakpoints and work on the next issue. As often happens, the next bug could benefit from the same set of breakpoints!

Most debuggers allow programmers to name breakpoints, disable them, and save them. If you have just created a particularly valuable breakpoint set, take a moment to preserve it. Give each breakpoint a label (such as “`Account::Initialize`”) and save the set to a file (“`Account breakpoints`”). You will thank yourself when fixing the next bug in that class!

Conditional Breakpoint

Most debuggers allow the programmer to specify conditions on which a program halts. The most useful conditions are threshold (when the value of a variable is above or below a certain value) and value (when a variable is at a certain value).

A screenshot of a Java debugger interface. The code window shows the following Java code:

```
Java
{
    float sum = 0;
    for (float grade : grades)
        sum += grade;
    average = sum / grades.length;
}
```

An annotation on the right side of the code window indicates a condition: "grade < 90". A curved arrow points from this annotation to the line of code "sum += grade;".

Figure 34.10:
A conditional breakpoint
is set at this point in the
program

For example, imagine a program tracking a student's grade. Though all the inputs are A's, for some reason the average grade is reported as 15%. A conditional breakpoint can be set that fires every time a grade is lower than an A. When we execute this code, the breakpoint pauses execution on the fifth item. Here, we expect all the values to be between 0% and 100% and, for this student, all the values to be between 90% and 100%. The fifth item has a value of -1,000%. This is just the clue we are looking for! The bug is not in this averaging code, but in the `grades` array!

Notification Breakpoint

Many debuggers allow the programmer to create a breakpoint associated with a value rather than a line of code. At the declaration of a variable, the programmer can ask to be notified each time the variable changes. Thus, if some unforeseen statement is modifying a variable (such as an out-of-bounds array reference), the program will be halted at that change. This helps pinpoint errors that are otherwise difficult to detect.

A screenshot of a C++ debugger interface. The code window shows the following C++ code:

```
C++
{
    float grades[10];
    bool isValid = false;
    readGrades(grades);
    // convert 99.0 --> 0.99
    for (int i = 0; i <= 10; i++)
        grades[i] /= 100.0;
    assert(!isValid);
}
```

An annotation on the right side of the code window indicates "Notify us when isValid changes". An arrow points from this annotation to the line of code "bool isValid = false;".

Figure 34.11:
A notification breakpoint
is set on the variable
`isValid`

In the above code, notice now `isValid` is set to `false` and is never changed. We would therefore not expect the assert to fire. For some reason, it does! To figure out what is going on, we set a notification breakpoint on the `isValid` variable. For some reason, the breakpoint is tripped in the FOR loop. In fact, it is tripped when `i` equals 10. At this point, `grades[10]` equals some totally crazy number, not looking like a grade. Then we discover the reason: the loop is going between 0 and 10 when it should be 0-9. The eleventh element in the array is a combination of the Boolean variable `isValid` and the random three bytes after it in memory!

Call Stack

Consider the following structure chart where execution is currently in the `Rook::getValues()` method:

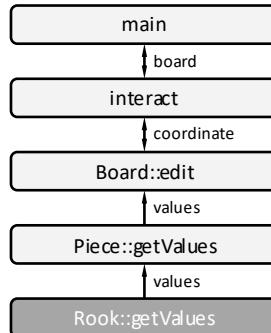


Figure 34.12:
The call stack for a chess-playing program

In order for execution to reach `Rook::getValues()`, it had to be called from `Piece::getValues()` and `Board::edit()` before it. The path in the structure chart from the current function to `main()` is called the call stack.

The call stack is a list of functions that represent the path program execution took to proceed from the start of the program (`main`) until the function where execution is halted. It is called the call stack for two reasons. First, it represents the functions that

have been called at a given moment in time (the “call” part of “call stack”). Second, the data structure representing the call stack is a stack. A stack is a “first-in, last-out” data structure, meaning the first element added to the stack (`main()`, in our case), is the last one removed. We can deduce the call stack from a structure chart by tracing the path from a given function up to `main()`. Traditionally, the call stack feature in a debugger is built bottom-up. This means that the first function is at the bottom and the current function is at the top.

Call stack

Chess.exe Rook::getValues(board:0x04019180)
Chess.exe Piece::getValues(board:0x04019180)
Chess.exe Board::Edit(coordinate:0x4020078)
Chess.exe interact(board:0x04019180)
Chess.exe main()

The call stack feature also allows the user to select the various functions for the purpose of viewing the state of their local variables. Thus, just clicking on an entry in the call stack will adjust the watch window to reflect the local variables at that point in execution. Most debuggers will also scroll the source code, so it is easy to tell where the function call was made.

Best Practice 34.12 Use the call stack in conjunction with the watch window for preliminary debugging investigation

When a program crashes, when an assert fires, or when a breakpoint is hit, the debugger halts execution at the offending line of code. Usually the first thing the programmer does is to check the watch window to see the state of the local variables. Next, the programmer looks at the call stack to see how execution reached that point. A few clicks on the call stack will reveal how parameters were passed to the relevant function and how the parameters were set. This way, the programmer can find all the necessary information in just a few moments.

Assembly & Machine

All high-level programming languages must be converted to machine language to be executed by a CPU. For most languages, this is accomplished by a compiler. Other languages (such as VB and Java) are interpreted. Regardless of how code is converted to machine language, it is occasionally helpful to know exactly what the CPU “sees” for a given line of code. This is where disassembly and memory watch come to play.

Memory Watch

As the name implies, the memory watch window allows the programmer to see what a given segment of raw memory contains. If, for example, the location of an object (which is obtainable through the immediate window) is known, then the programmer can view that location in the memory watch window to get a feel for how the compiler stored the data. Consider the following C# code:

```
C#
string text1 = "Software";
string text2 = "XXXXYYYY";
```

Figure 34.13:
A simple program with
two strings declared

We would like to know how this is stored in memory. If we pause the debugger, we can drag `text1` into the memory window. This gives us the following:

Memory 1		
Address:	0x0000000002620FF4	
0x0000000002620FF4	55 00 6e 00 69 00 63 00	U.n.i.c.
0x0000000002620FFC	6f 00 64 00 65 00 00 00	o.d.e...
0x0000000002621004	00 00 00 00 00 00 00 00
0x000000000262100C	00 00 00 00 10 0b 64 b2d.
0x0000000002621014	fe 07 00 00 04 00 00 00	b.....
0x000000000262101C	54 00 65 00 78 00 74 00	T.e.x.t.

Figure 34.14:
A memory watch
on two strings

Notice that each letter in `text1` takes two bytes. We can therefore deduce that C# stores text as Unicode characters. Memory windows are also useful for following linked lists. Here, the first node is at `0x249df0`. The second row of the memory watch points to the second node at `0x2492e10`. Can you see it? The bytes are reversed because this computer is little-endian (the byte order is reversed).

Memory 1		
Address:	0x0000000002492DF0	
0x0000000002492DF0	58 68 11 5c fe 07 00 00	Xh.\p...
0x0000000002492DF8	10 2e 49 02 00 00 00 00	..I.....
0x0000000002492E00	05 00 00 00 00 00 00 00
0x0000000002492E08	00 00 00 00 00 00 00 00
0x0000000002492E10	58 68 11 5c fe 07 00 00	Xh.\p...
0x0000000002492E18	00 00 00 00 00 00 00 00

Figure 34.15:
A memory watch on
two linked list nodes

Disassembly

Many debuggers provide a view where the assembly or machine code corresponding to the high-level language code is visible to the programmer. Usually this code is presented adjacent to or inline with the corresponding source code.

C++ with Disassembly

```
if (input > 5)
013C2595 cmp      dword ptr [input],5
013C2599 jle    main+7Eh (013C25AEh)
    cout << "Five";
013C259B push    offset string "Five" (013C9B34h)
013C25A0 mov      eax,dword ptr [_imp__cout@std@@3V?
$basic_ostream@DU?
$char_traits@D@std@@@1@A (013CD0D0h)]
013C25A5 push    eax
013C25A6 call    std::operator<<
<std::char_traits<char> > (013C120Dh)
013C25AB add     esp,8
```

Figure 34.16:
Disassembly of
an IF statement

In the above example, there are two lines of source code (an **IF** statement and a **COUT** statement) and the corresponding disassembly. Notice that the **IF** statement is just two lines of code (one for the Boolean expression and one for the conditional jump). However, the **COUT** statement results in five: one to push the text of the parameter into the call stack so it can be passed as a parameter, two to copy the **OSTREAM** parameter into the call stack so it can be passed as a parameter, the actual act of calling the function **COUT**, and popping the parameters off of the call stack.

Best Practice 34.13 Use the disassembly view to see exactly how a given programming construct is compiled

The disassembly view allows the programmer to see how much code gets generated from a given line of source. For example, the following **IF** statement looks very innocuous. However, it actually requires four function calls!

C++ with Disassembly

```
if (board[posMove] == ' ' && board[posCheck] == ' ')
000AF12E push    20h
000AF130 lea     eax,[ebp-7Ch]
000AF133 push    eax
000AF134 mov     ecx,dword ptr [board]
000AF137 call    Board::operator[] (093393h)
000AF13C mov     ecx,ecx
000AF13E call    Piece::operator== (093BE5h)
000AF143 movzx  ecx,al
000AF146 test   ecx,ecx
000AF148 je     Pawn::getMoves+36Ah (0AF1DAh)
000AF14E push    20h
000AF150 lea     eax,[ebp-88h]
000AF156 push    eax
000AF157 mov     ecx,dword ptr [board]
000AF15A call    Board::operator[] (093393h)
000AF15F mov     ecx,ecx
000AF161 call    Piece::operator== (093BE5h)
000AF166 movzx  ecx,al
000AF169 test   ecx,ecx
000AF16B je     Pawn::getMoves+36Ah (0AF1DAh)
```

Figure 34.17:
Disassembly of
a complex statement
masquerading as
a simple one

Examples

Example 34.1: The Most Difficult Bug Ever!

This example will demonstrate how to use the debugger to fix a difficult bug.

Problem

Many years ago, the author was working on a gargantuan desktop application. A bug was reported where, on quit, the operating system was rebooted. At first, he was suspicious. Did they mean that the application crashed, or the operating system rebooted? After executing the four steps listed in the bug report, his computer did in fact reboot! Note that if the program crashed, then his debugger would have halted at the offending line. With an OS reboot, he had no such luck!

Solution

The first step was to figure out where the program crashed. This involved stepping into `main()` and then executing step over. When the function was discovered causing the reboot, then the next iteration involved stepping into that function. This process continued until the offending function was discovered: the code tearing down the toolbars on exit.

Due to a poor design, the toolbar teardown code was an $O(n^3)$ algorithm. This was because each toolbar was destroyed one at a time. With each toolbar destruction, the remaining were repositioned using an $O(n^2)$ algorithm. After stepping through this loop several times, it was discovered that the 213th instance of `repositionToolbar()` was the cause. Here, a conditional breakpoint was set where it triggered on instance number 213. Note that it took 213 system reboots to discover this!

From here, a few step intos revealed the offending line of code. On this system, the process of loading a dynamic system library was achieved with a `loadLibrary()` call and, when done, it was released with `deleteLibrary()`. If `deleteLibrary()` was called on a library not currently loaded, this version of the operating system would reboot (note: this behavior was reported to the operating system folks and they went on to fix it). The fix in our code was simple: remove the extra `deleteLibrary()`.

Not wanting to ever have to fix a bug like this again, the author added some debug code. An instance counter was incremented each time `loadLibrary()` was called, and the counter was decremented with each call to `deleteLibrary()`. An assert was added immediately before `deleteLibrary()` that instance was greater than zero. A few weeks later, this newly added assert fired. Wasn't he relieved! The extra `deleteLibrary()` call was found and fixed in a couple minutes, whereas the first bug took almost three days.

Example 34.2: Memory Overwrite

This example will demonstrate how to find a bug using a debugger.

Problem

Consider the following code fragment to play the game Tic-Tac-Toe.

```
C++  
int main()  
{  
    int board[3][3] = {};  
    // initialize to all zeros  
    int score = 0;  
    readBoard(board);  
    interact(board);  
    writeBoard(board);  
    return 0;  
}  
  
void readBoard(char board[][3])  
{  
    ifstream fin("game.txt");  
    for (int row = 1; row <= 3; row++)  
        for (int col = 1; col <= 3; col++)  
            fin >> board[row][col];  
    fin.close();  
}
```

There is a bug in the `score` variable has a nonzero value after reading the board from a file.

Solution

The first step is to put a notification breakpoint on the `score` variable in `main()`. The breakpoint is triggered in line 03 when the variable is initialized. This is expected and not a bug. The breakpoint is triggered again in `readBoard()` at line 15. This is not expected; that variable is not even in scope in `readBoard()`! From this point, we look at the local variable watch window and discover that the value of `row` is 2 and the value of `col` is 3.

From the watch window, the address of the board is determined: `0xbfff2018`. The memory watch window is brought up and we can see the 1s (corresponding to 'X'), 2s (corresponding to 'Y'), and 3s (corresponding to the space) for the Tic-Tac-Toe board. Interestingly, there is nothing in the first four slots; they still have the value zero. Why are we skipping the first four slots?

Going back to the source code, we can see that we are counting from 1 to 3 for the indices of the array. Now we remember: arrays start counting at 0 and not at 1. Thus, we should be counting from 0 to 2 and not 1 to 3. Further investigation reveals that the `score` variable is immediately after the board in memory. We can check this by adding the number 999 in the first slot in memory after the tic-tac-toe board. When we use the call stack feature to look at the local variables for `main()`, we see the `score` variable set to 999.

To verify the solution, the start- and end-conditions of the FOR loop are modified and, when reran, the notification breakpoint for `score` does not fire in `readBoard()`.

Example 34.3: Recursion

This example will demonstrate how to debug a recursive function using a debugger.

Problem

Consider the following code.

```
Python
def factorial(number):
    if number == 1:
        return 1
    else:
        return factorial(number) * number
93
94
95
96
97
```

When the code is executed, it is an infinite loop. The program hangs until it crashes when Python runs out of stack space. How can the bug be found using recursion?

Solution

We will start by calling this function with the number 5 as a parameter. This should go through five iterations and yield: $5 \times 4 \times 3 \times 2 \times 1 = 120$. However, it crashes instead.

We will next set a breakpoint at line 94 (the first line of the function body) and set a conditional breakpoint: pause on the fourth time the line of code is reached. When the program is executed, it pauses at that line. The call stack is the following:

```
Call Stack
factorial(number:5)
factorial(number:5)
factorial(number:5)
factorial(number:5)
```

Notice that the input parameter is the same with every call to the function. We will verify this by moving down the call stack. In every level, the value of `number` is the same. The problem apparently is that we are not decrementing `number` with every successive call to `factorial()`.

We will verify this assumption by starting over and removing the conditional part of the breakpoint. Initially the value of `number` is 5. As we “run” from there, the next breakpoint is hit. We will change the value to 4 manually in the watch window. We will continue with this process until we set the value 1 on the fifth call of the function. Now the call stack is:

```
Call stack
factorial(number:1)
factorial(number:2)
factorial(number:3)
factorial(number:4)
factorial(number:5)
```

We will fix the bug by decrementing the value of `number` by 1 in the call to `factorial()`. As we execute the code for the third time, we can see how the parameter is changed and the return value is multiplied on each successive call.

Exercises

Exercise 34.1: Debugger Tools

From memory, name and explain the following debugger tools.

Watch Local Variables	
Disassembly	
Step Over	
Conditional Breakpoint	
Memory Watch	
Set Next Statement	
Run To	
Notification Breakpoint	

Exercise 34.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Step out pauses execution in a function called from the current location.	
A watch allows viewing and editing variables.	
Step over always pauses execution in the caller.	
You can jump between scopes with the call stack.	
Debuggers cannot show compiled code.	
Run to alters program flow to a given location.	
The memory watch can only show stack memory.	

Exercise 34.3: Stepping

Consider the following code with the cursor at line 20 and execution halted at 26:

```
Swift
func A(value: Int) -> Int {
    B(1) ← Cursor
    C(2)
    return 3
}

func B(value: Int) -> Int {
    C(4)
    value *= 4
    return value
}

func C(value: Int) -> Int {
    return 7
}
```

19
20
21
22
23
24
25
26
27
28
29
30
31
32
33

Where will execution be paused if the user specifies each of the following?

Stepping	Line number
Step Over	
Step Into	
Step Out	
Run To	
Set Next Statement	

Exercise 34.4: Stepping

Consider the following code with the cursor at line 95, execution paused at 93, and the current value of `number` at 4:

```
Python
def factorial(number):
    if number == 1:
        return 1 ← Cursor
    else:
        return factorial(number - 1) * number
```

93
94
95
96
97

Where will execution be paused and what will be the new value of `number` if the user specifies each of the following?

Stepping	Line number	Value of number
Step Over		
Step Into		
Step Out		
Run To		
Set Next Statement		

Problems

Problem 34.1: Crash in a Library

Consider the following scenario:

Your program crashed, which halted in the debugger. Unfortunately, the crash happened in a generic library function used in hundreds of places in the program. You would like to know which line of code in your program called this library function.

Identify which debugger tool(s) would be helpful in diagnosing the problem. Next, describe how you would go about using the tool or tools.

Problem 34.2: Polymorphic Variable

Consider the following scenario:

You have the program halted in a function that works with a polymorphic variable. It would really help to know which derived class was associated with the variable you are looking at right now.

Identify which debugger tool(s) would be helpful in diagnosing the problem. Next, describe how you would go about using the tool or tools.

Problem 34.3: Blocking Bug

Consider the following scenario:

It took you 10 minutes to get the program in a right state where the program exhibits the bug. Unfortunately, a simple error in the code is preventing you from moving on and discovering the real problem. You really don't want to recompile and spend another 10 minutes to get to the same spot.

Identify which debugger tool(s) would be helpful in diagnosing the problem. Next, describe how you would go about using the tool or tools.

Problem 34.4: Reading a File

Consider the following scenario:

There is a problem in the file-reading code in your application. After reading about 100 records from a file, the subsequent ones are corrupt. You don't want to use step over 100 times! That will take forever.

Identify which debugger tool(s) would be helpful in diagnosing the problem. Next, describe how you would go about using the tool or tools.

Problem 34.5: Disassembly

In the language of your choice, implement the following program:

Pseudocode

```
PROMPT for a number
GET number

SWITCH number
CASE 1:    value = 0x101    BREAK
CASE 2:    value = 0x102    BREAK
CASE 3:    value = 0x103    BREAK
CASE 4:    value = 0x104    BREAK
CASE 5:    value = 0x105    BREAK
CASE 6:    value = 0x106    BREAK
CASE 7:    value = 0x107    BREAK
CASE 8:    value = 0x108    BREAK
CASE 9:    value = 0x109    BREAK

SWITCH number
CASE 1:      value = 0x101    BREAK
CASE 10:     value = 0x102   BREAK
CASE 100:    value = 0x103   BREAK
CASE 1000:   value = 0x104   BREAK
CASE 10000:  value = 0x105   BREAK
CASE 100000: value = 0x106   BREAK
```

Execute the program and break into the debugger. Check the disassembly for this code. How does your compiler treat the two SWITCH/CASE statements differently? Please read the Chapter 07 section on “Many Options” and see if you can classify the implementation of each as IF/ELSE-IF, Iteration, Binary Search, or Jump Table.

Challenges

Challenge 34.1: Trace Verification

In the project of your choice, insert breakpoints at all the locations where data are written to a file. Execute your program and write down on a sheet of paper what you expect the contents of the file will be. When finished, compare your notes with the actual contents of the file.

Challenge 34.2: Bubble Sort

Consider the following algorithm for a bubble sort.

Pseudocode

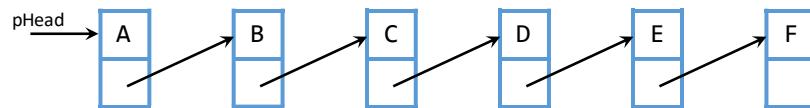
```
FOR iSpot ← numItems ... 0
    FOR iCheck ← 0 ... iSpot - 1
        IF array[iSpot] < array[iCheck]
            swap( array[iSpot], array[iCheck] )
```

Please do the following:

1. Implement this algorithm in the programming language of your choice.
2. Create an array with the following values: [72, 63, 49, 55, 81, 93].
3. Perform a manual program trace.
4. Verify your manual trace with the debugger. Check off each line of your manual trace as you get to that point in the debugger.

Challenge 34.3: Linked List

Consider the following linked list:



Please do the following:

1. Implement this linked list in the programming language of your choice.
2. Write the code to loop through the linked list, displaying the values.
3. See if you can swap the nodes (not the values!) of node B and node D in the debugger.
4. Verify that the values are actually swapped by writing the code to display the contents.

Challenge 34.4: Call stack

Consider the following pseudocode:

Pseudocode
main() number ← 255 text ← "The function main()" function1(42) END function1(parameter) array ← 1, 2, 3, 4, 5, 6, 7, 8 float ← 3.14159 function2("parameter to function2") END function2(text) boolean ← TRUE character ← z END

Please do the following:

1. Implement this code in the programming language of your choice.
2. Set a breakpoint at the line indicated with the blue highlight.
3. Locate **boolean** and set the value to FALSE, locate **character** and set the value to 'A'.
4. Locate **array** and set the values to [2, 3, 5, 7, 11, 13, 17, 19], locate **float** and set the value to 2.71828.
5. Locate **number** and set the values to 999, locate **text** and set the value to "Debugger".
6. Find the address of **boolean**. Look up that location in the memory watch. Can you find the other local variables in the various functions? What does this tell you about how your compiler implements the call stack?

Polymorphism

Polymorphism is the concept of one class having more than one variation, each of which honors the same contract but has different behaviors.

Imagine an object of type **Account**, but it could be of the checking, savings, or credit card variety. Imagine an object of type **List**, but it could be of the grocery list, to-do list, or bucket-list variety. Imagine a game having a collection of **Obstacles**, containing any number of walls, barriers, or enemies. In each of these cases, the program thinks it has an instance of a base class, but it actually is an object of a derived class. We call this technique polymorphism.

Polymorphism is an OO strategy where the various derived classes have different implementations but the same interfaces

object could be an instance of the derived class. To see how this works, consider an **Account** type for a financial application. This class provides the following public virtual methods or interfaces: **computeReport()**, **insertTransaction()**, and **getBalance()**. Any reference to an **Account** object can call these three interfaces. It turns out that **Account** is a base class and has three derived classes: **Checking**, **Savings**, and **CreditCard**. Each of these has unique implementations of these three interfaces. **Savings**, for example, has an interest feature in **getBalance()** whereas **CreditCard** will add in the annual fee.

Polymorphism is an object-oriented programming strategy where the various derived classes have different implementations of a shared interface. This shared interface is defined in the base class.

When working with an object of this base class, this

Each of the derived classes honors the same contract as the base class

Every Account object is actually from one of the derived classes

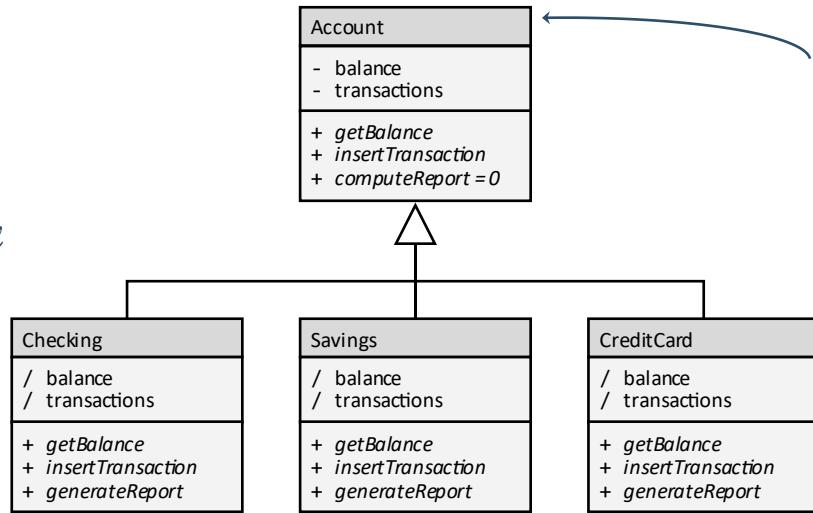


Figure 35.1:
Class diagram of a simple polymorphism scenario

Polymorphism is made possible through late binding, which is the process of the program realizing specifics of an object at runtime rather than at compile time.

Early and Late Binding

Binding is tying named programming entities to locations in memory

Binding is the process of tying named programming entities to physical locations in memory. It could be tying the variable name `count` to a memory slot where an integer is stored. It could be tying the function name `interact()` to the place where the compiled code from that function is located. All programming constructs are compiled and are placed in memory. This is true for variables, loops, arrays, functions, classes, methods, member variables, and objects. All programming constructs

are also named, and binding ties the name to the compiled code. There are two basic types of binding: early and late binding. The former refers to binding at compile time, the latter at runtime.

Early Binding

Early binding, also known as static binding and compile time binding, is the flavor of binding that the compiler performs before the program is executed. Here, the compiler has all the information necessary to tie a named element to compiled code. This is how variable names, function names, and class names are connected to the code that is produced in the executable.

Early binding can happen when the compiler has enough information to resolve a programming entity

Variable Binding Table

Consider the code to declare an integer named `count`. When the compiler encounters this declaration, two things happen. First, a location in memory is set aside for this variable. Let's say that location is `0x00409008`. Second, an entry in the binding table is created where the label `count` is associated with `0x00409008`. From the point where the variable is declared until it falls out of scope, all references to `count` will refer to location `0x00409008`. The compiler maintains this binding table for variables, so it knows exactly which location in memory is associated with every variable name, and it knows the data type of each variable.

At this point in compilation, there are four variables

Name	Data-type	Location
index	integer	0x00409000
firstName	string	0x0040A0DD
lastName	string	0x0040A0E0
count	integer	0x00409008

From the binding table, we can see that count is an integer and where it resides in memory

Figure 35.2: Variable binding table

This variable binding table is the source of many compiler errors. What happens if we attempt to declare a variable of name `count` when there is already a `count` in the binding table? The compiler will notice this duplication and throw an error. What happens when we reference the variable `count` when there is no such entry in the binding table? Again, the compiler will throw an error. Finally, what happens when we attempt to assign a string into an integer? The compiler notices that the `count` entry in the binding table refers to integers, but we are attempting to assign a string. Since the compiler maintains the variable binding table, the compiler generates errors when discrepancies exist.

Function Binding Table

Function binding tables work much the same way as variable binding tables. When a function is declared, a location in memory is set aside for the compiled code. Let's say the function `interact()` is given the location `0x4002D20F`. This name and location are placed in a function binding table much like the variable binding table. The main difference is that the "data type" field of the function binding table is much more complex. It includes the complete function signature, including the return type, the types of the parameters, and other language-specific information.

Name	Return Type	Params	Location
main	integer	void	0x40029840
getFileName	string	void	0x4002A004
readFile	Board	string	0x4002A05D
writeFile	void	string, Board	0x4002A114
interact	void	Board	0x4002D20F

The code for
`interact()`
is located here:

Figure 35.3:
Function binding table

When the function body itself is compiled, all the memory instructions associated with the function are places in sequence starting at `0x4002D20F`. When the function `interact()` is invoked, the compiler knows to jump to location `0x4002D20F` to start executing the compiled code that is stored there. If there is a missing entry in the function binding table at the end of compilation, the compiler will throw an error such as "`undefined reference to `interact(void)``". If there is not a missing entry, then all reference to named components in the code will be replaced with the actual addresses in memory in which they correspond.

Data Type Binding Table

Compiling classes works somewhat differently than other code. Here, the compiler needs to store instructions for how to build an object matching the definition of the class. Most methods in a class are stored directly in the function binding table. Member variables are stored differently. Here, the compiler needs to know how much space is required to create an object, and details about the individual member variables. The "other stuff" in the binding table includes data type information and other details needed to represent the member variable. These details are important for understanding how compilers work but are beyond the scope of this chapter.

The data type binding table is a list of all the known data types. When a new data type is defined (such as a class), then a new entry is created in this table.

Name	Size	Other Stuff
int	4	
float	4	
bool	1	
char	1	
Board	64	

Figure 35.4:
Data type binding table

There are five known
data types at this
point in the program

All the stuff needed
to interpret a data
type or translate
one is located here.

For illustration purposes, the built-in data types are pre-populated in the binding table. Many languages mix built-in and user-defined data types in the data type binding table, but some do not. All binding tables are created and maintained by the compiler. They represent everything the compiler knows about a named programming entity.

Late Binding

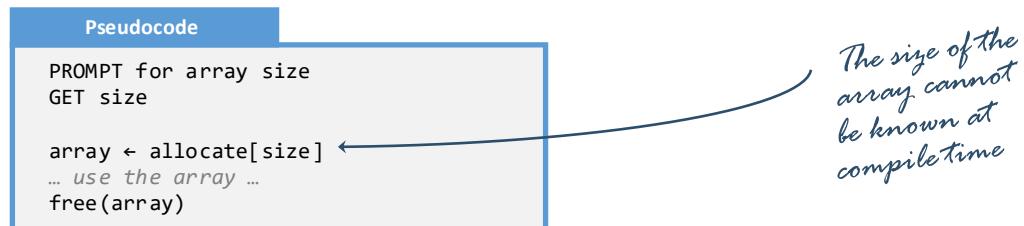
Late binding happens at runtime

Late binding, also known as dynamic or runtime binding, is the process of the program connecting named elements to locations in memory at runtime. This is usually done because the compiler does not have enough information to perform the binding at compile time.

Arrays

Array declaration can usually be handled at compile time using early binding. There are times, however, when this is impossible. Consider, for example, the code to allocate an array of integers where the size is determined by the user. In this case, the data type of each element in the array is known (an integer in this case), but the number of elements is not (specified by the user). It is impossible for the compiler to set aside memory for this array because a critical piece of information is missing. This linkup must happen at runtime. Here, the compiler will insert code that will request the necessary memory and, when the memory is procured, will put the resulting address into the associated variable.

Figure 35.5:
Dynamically allocated
array

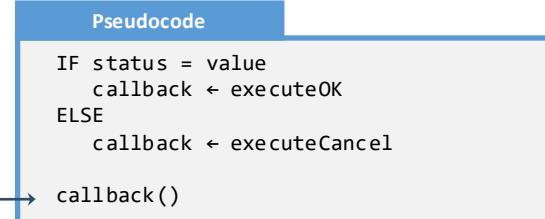


There are several errors which are impossible for the compiler to catch. One is a memory allocation error. The compiler cannot anticipate if there is enough memory for the array. Another is bounds checking. If the compiler cannot tell how many elements are in the array, then it cannot determine if a given index is within the valid range. In both cases, we must rely on IF statements to detect error conditions. These IF statements are runtime checks, which are late binding.

Callbacks

Recall that a callback is a function that is passed to another function as a parameter for the purpose of being executed at some pre-appointed time. During a normal function call, the compiler knows exactly which function is to be executed at the time when a function call is made: the function name that has a corresponding memory location in the function binding table. With a callback, this is not the case. The program logic could assign any one of several functions to this callback, meaning the compiler cannot know which function is eventually called.

Which function
will be executed?
The compiler cannot
tell!

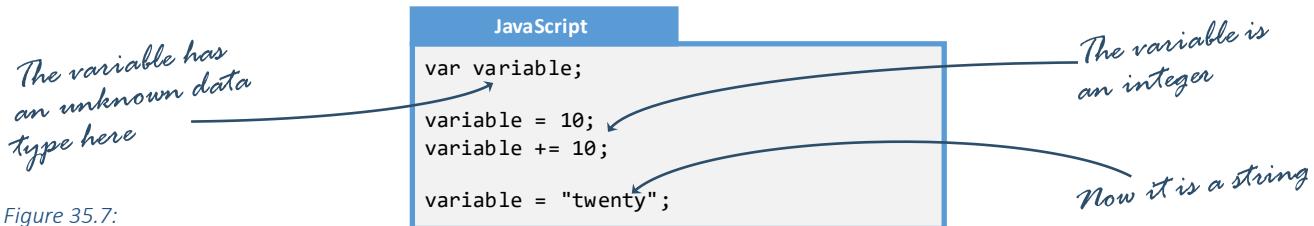


In cases like this, the decision of which function to be called occurs at runtime. This is an example of late binding.

Figure 35.6:
Callback

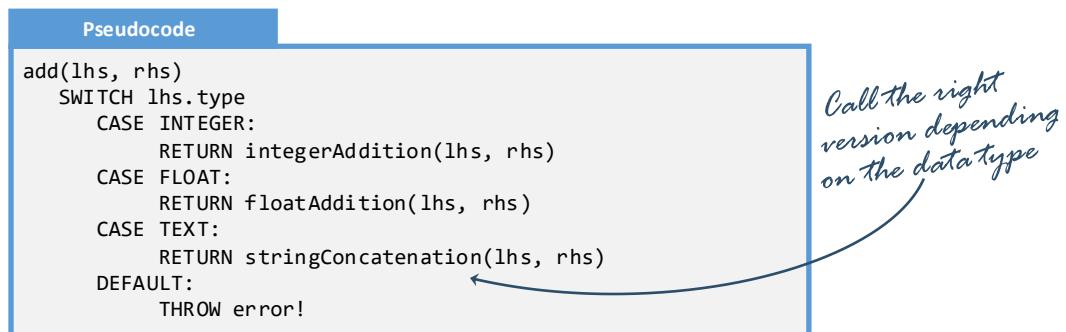
Dynamic Data Types

Many programming languages provide the capacity to declare a variable without specifying the data type. Some even allow for the data type to change during the lifetime of a single variable. This provides quite a bit of flexibility, but also precludes early binding.

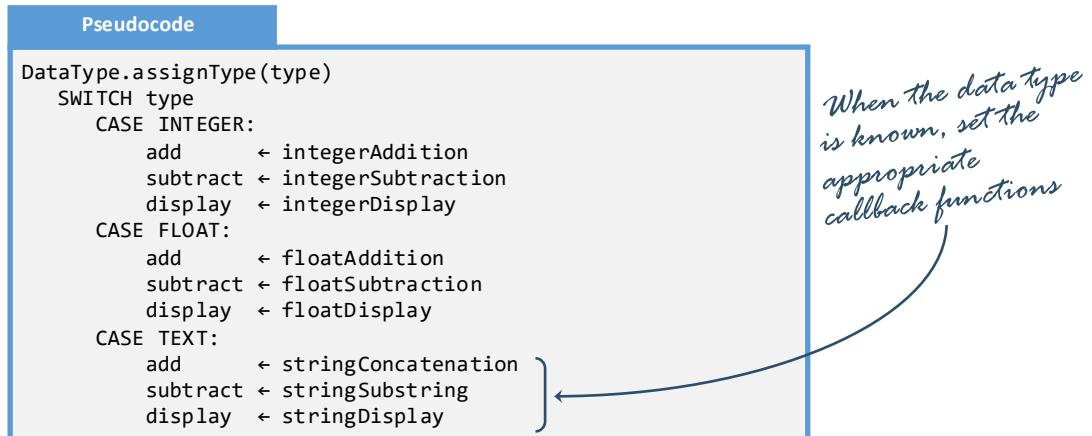


To handle this case, the authors of JavaScript need to give each variable the capacity to be any data type, the selection of which occurs at runtime. Thus, the variable's data type information is encapsulated with the variable itself and not maintained in a variable binding table.

To see how this works, imagine that JavaScript has just three build-in data types: integers, floats, and strings. Programmers will use the plus operator, requiring addition to be defined for integers, floats and strings. One way to implement this functionality is with a `SWITCH` statement, calling the appropriate variant of addition depending on the data type:



Modern implementations assign the “string plus” function to the “plus” callback. This makes calling functions with dynamic type variables nearly as efficient as doing the same with standard variables.



Polymorphism

Polymorphism is the process of determining at runtime which flavor of a class is associated with a given object. This is accomplished using the same mechanism used with dynamic data types. A base class can have two types of methods: virtual methods and standard methods. Virtual methods can be overwritten by unique implementations defined in derived classes, whereas standard methods cannot. These two types of methods are handled very differently.

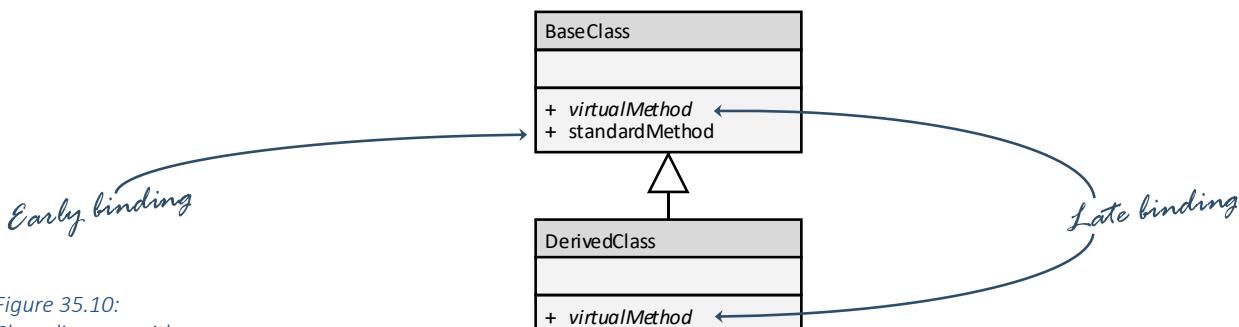


Figure 35.10:
Class diagram with a
standard method and a
virtual method

Standard methods (those of the non-virtual variety) are handled by the compiler in a standard function binding table. In other words, aside from scope restrictions (methods can only be called in the presence of objects belonging to that class), standard methods are defined globally and are entries in a function binding table. Any error pertaining to declaring or calling a standard method results in a compile error rather than a runtime error.

Standard methods use early binding

All standard methods are in the function binding table

Name	Scope	Location
main	global	0x40029840
standardMethod	BaseClass	0x4002A004
readFile	global	0x4002A05D
writeFile	global	0x4002A114
interact	global	0x4002D20F

Figure 35.11:
A standard method placed
in a function binding table

Virtual methods (those that may be redefined in derived classes) are handled by v-tables. A v-table (virtual method table) is a set of callbacks (otherwise known as function pointers) referring to the virtual functions associated with a given derived class. You can think of this v-table as a function binding table that is maintained at runtime rather than at compile time. When a virtual function is called, then the appropriate callback is invoked.

Virtual methods use late binding

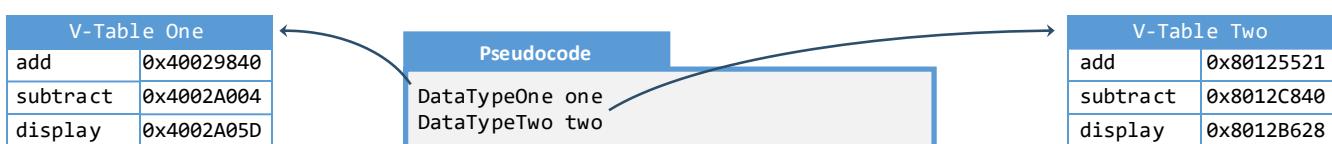


Figure 35.12:
Two v-tables associated
with different objects

There is a slight performance cost to using polymorphism. First, when an object is instantiated, the appropriate v-table is assigned to the object. Second, the object needs to carry around its member variables as well as a reference to the appropriate v-table. Finally, invoking a function involves a function call to a callback, which takes a few more clock cycles than it would for a standard method.

Implementing Polymorphism

Polymorphism can seem like magic and, as a result, can be difficult to internalize. To remove this mystery, it is helpful to see how a language such as C++ would implement it. Consider a simple **Date** class that has three variants: **ShortDate** (1/1/00), **OfficialDate** (01.01.2000), and **LongDate** (1st of January, 2000).

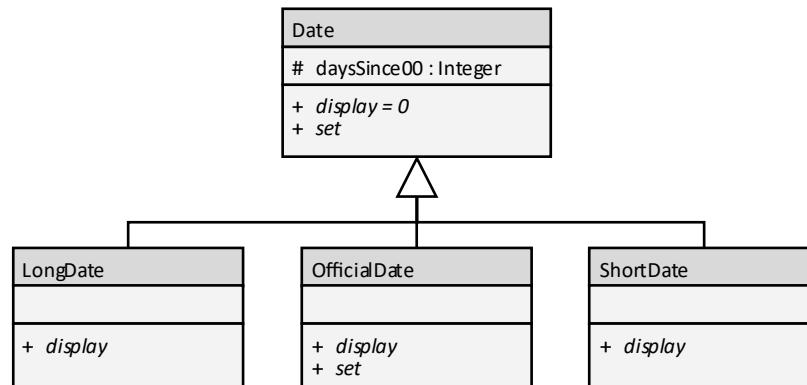
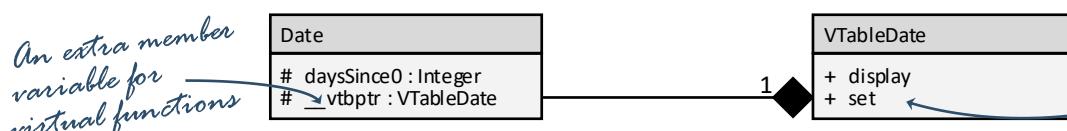


Figure 35.13:
Class diagram of
polymorphism with two
virtual methods

V-Table

C++ and many other object-oriented programming languages implement virtual methods as a collection of callbacks (otherwise known as function pointers). This collection is placed in a structure containing all the virtual methods, the v-table. Each object that has a virtual method has an extra member variable: a pointer to the v-table. By tradition, the member variable is called `_vtbptr`. Thus, internally the class diagram consists of two structures: the base class and the v-table.



Virtual functions
are stored here

Figure 35.14:
Class diagram of a
class' v-table

Since there are two virtual methods (`display()` and `set()`), then there are two elements in the v-table (each being a callback or function pointer). Note that the first parameter is an instance of the class of which it is a member (called `pThis`).

```

C++
struct VTableDate
{
    bool (*set)    (      Date * pThis, int y, int m, int d);
    void (*display)(const Date * pThis);
};
  
```

Figure 35.15:
An implementation
of a v-table:
two callback functions.

Class Implementation

At compile time, all methods in a class get partitioned into standard methods and virtual methods. The standard methods are placed in the function binding table and do not impact the size of the class in any capacity. The virtual methods need to be attached to the object itself. This occurs by adding the v-table as a member variable to the class.

```
C++  
struct Date  
{  
public:  
    VTableDate * __vtbptr; // the name most compilers use  
protected: // for the v-table  
    int daysSince00; // the member variable  
};
```

Figure 35.16:
How a v-table is attached
to a class definition.

From here, we can see the size penalty of virtual methods: a single pointer. If there were no virtual methods associated with this `Date` class, then the size of an object would be `sizeof(int)` which is 4 on most systems. With the virtual methods, the size is now: `sizeof(int) + sizeof(*)` which is 12 for a 64-bit computer.

Binding

The final step is to assign the appropriate v-table to each object upon creation. This will allow each object to “know” which class it belongs to. We will create a function called `bind()` which will assign the appropriate v-table to each object.

```
C++  
void bind(Date * pThis, DateType dt)  
{  
    static VTableDate DateShort = { &setDate, &displayShort };  
    static VTableDate DateOff  = { &setDate0, &displayOff };  
    static VTableDate DateLong = { &setDate, &displayLong };  
  
    switch (dt)  
    {  
        default: assert(false);  
        case SHORT: pThis->__vtbptr = &DateShort; break;  
        case OFFICIAL: pThis->__vtbptr = &DateOff; break;  
        case LONG: pThis->__vtbptr = &DateLong; break;  
    }  
}
```

Figure 35.17:
The process of binding a
v-table to a specific
object.

In languages such as C++, the binding operation only occurs when an object is instantiated. However, as we saw with JavaScript, many languages allow an object to change its data type at any time. This means the `bind()` function can be called freely.

Designing Polymorphism

Recall the definition of polymorphism: an object-oriented programming strategy where the various derived classes have different implementations of a shared interface. The three key elements of this definition are the fact that each derived class has a shared interface and different implementations.

Derived Classes

The collection of all classes that share the same common interface and can stand in for each other is called a polymorphic set. The common interface shared with all members of a polymorphic set is defined in the base class. This is because polymorphism is built with inheritance; every class in a polymorphic set must derive from the same base class. Looking back to our financial account example, if we wished to use polymorphism to represent the different flavors of accounts, then the set of accounts deriving from the `Account` base class would constitute the polymorphic set. Consider the following hypothetical inheritance tree:

A polymorphic set is the collection of classes that derive from a single base class

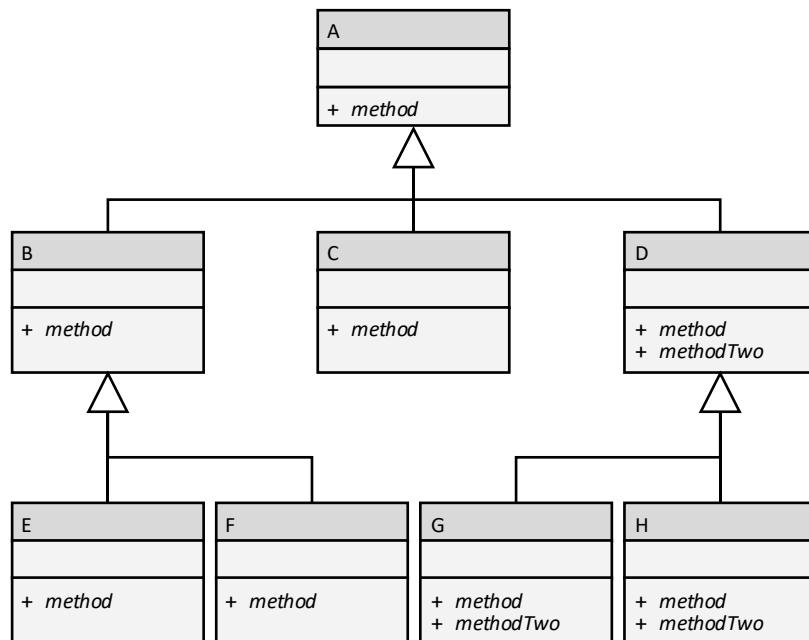


Figure 35.18:
Class diagram illustrating
a multi-layer inheritance
tree

If we are using A as the base class, then { A, B, C, D, E, F, G, and H } could all be part of the same polymorphic collection. This is because they all derive from A. This means that any derived class under A in the inheritance tree can stand in for A.

If we were using D, on the other hand, then only { D, G, and H } would be part of the same polymorphic collection. Observe that A or B could not stand in for D because they do not define `methodTwo`.

Best Practice 35.1 All members of a polymorphic collection must derive from the same base class

The first step in every polymorphic design scenario is to create an appropriate inheritance tree. The definition of the base class is critical because it will define the shared interfaces which all derived classes will need to implement.

Best Practice 35.2 Every derived class should be able to stand in for its base class

This design rule was coined by Robert Martin and is called the Liskov Substitution Principle. It is part of his SOLID suite of design guidelines (where this is the L of SOLID). Think of the base class as a contract: it constitutes a set of actions that are guaranteed to be fulfilled. If I were to utilize a class, then I would expect all the public methods to work as advertised. The Liskov Substitution Principle states that any of the children of that class will fulfill the same contract. They can, in effect, stand in for the base class. There are two important components to this principle: what they expect of the client, and what they provide to the client.

The base class constitutes a contract which all derived classes must fulfill

Best Practice 35.3 A derived class should expect no more of the client than the base class

This is the first part of the Liskov Substitution Principle. Perhaps it is best explained by example. If a class was designed to represent `Time`, then it might have a `setHour()` method which expects an integer with the range of 0-24 (because it works with military time). Now I implement a derived class `TimeAmerican` which works on a 12-hour clock utilizing a.m. and p.m. This derived class implements `setHour()` and expects an integer with the range of 1-12. Here, we are violating the Liskov Substitution Principles because the precondition of the derived class is stronger than that of the base class.

The preconditions of the derived class should constitute a superset of those of the base class

Ideally, the preconditions of the derived class should be looser than that of the base class. This means it can accept a wider range of input. In other words, the preconditions of the derived class should constitute a superset of those of the base class.

Best Practice 35.4 A derived class should deliver no less to the client than the base class

This constitutes the second part of the Liskov Substitution Principle. It means that for any method the base class provides, the derived class provides it as well. In other words, disinheritance (when a derived class explicitly does not implement a method of the base class or decides to not inherit it) is a violation of the Liskov Substitution Principle.

The postconditions of the derived class should constitute a subset of those of the base class

There is a second component to this best practice. If one were to create a test assessing the quality of the output of a base class method, then it would pass for the derived class as well. Note that we would expect the nature of the output of the base class and the derived class to be different—otherwise the two provide exactly the same functionality and there is no difference! The requirement is that the quality of the output would be at least as high. For example, consider a class called `File` which both reads and writes the contents of the user's document to the file system. One could create a derived class called `FileJSON` which honors the same contract and provides the same functionality, but does so differently. Instead of writing to an XML document, it writes to a JSON document. Both implementations can round-trip the user's document (write to a file then read it, without losing any data) so they pass the same quality test. Thus, `FileJSON` can stand in for `File`.

Shared Interface

Every derived class in a polymorphism collection must honor the same interface. In fact, only this shared interface is visible to the client. Again, this is best explained by example. Consider an **AccountList** which is a collection of bank accounts. The number of accounts and the type of each account depend on the specific needs of the user. There are several types of accounts: cash accounts, investment accounts, and loan accounts.

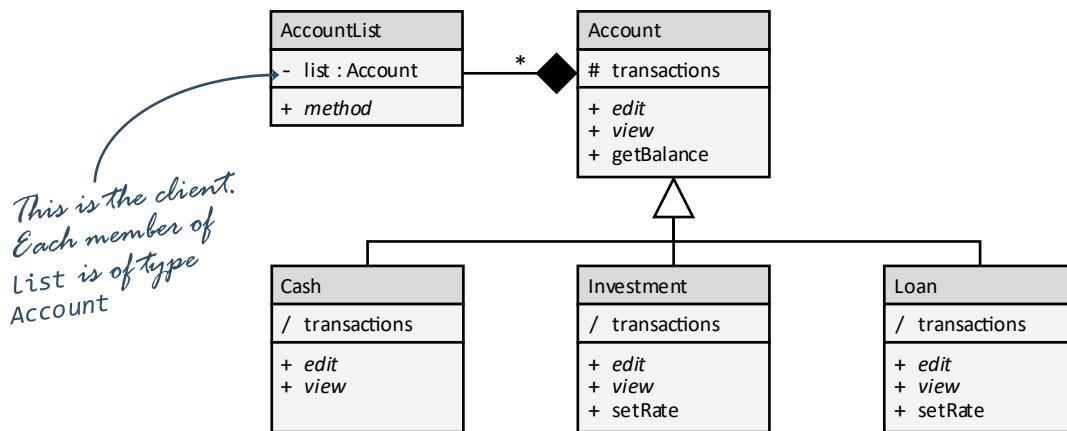


Figure 35.19:
Class diagram illustrating
the shared interface that
each derived class must
implement

Notice that **AccountList** has a collection of **Account**. In reality, each member of that collection is of type **Cash** or **Investment** or **Loan**. It might be that the collection consists completely of **Loan** accounts. It might be that there is an even mixture of **Cash** and **Investment**. **AccountList** cannot tell what the composition will be at compile time; it may even change during execution.

Best Practice 35.5 Make sure the interfaces in the base class are enough

The client (**AccountList** in this case) only has access to the public interfaces of the base class. Any public interfaces in the derived class (such as **Loan.setRate()** or **Investment.setRate()**) are not accessible to the client. In fact, this is perhaps the single most important decision: exactly what interfaces will be exposed to the client through the base class? Some things to consider:

- **Enough for all derived classes:** The set of interfaces exposed to the client are enough to meet the needs of each and every derived class. In the above example, both the investment and the loan class need to set an interest rate. How can this be done when the client does not have access to it?
- **Meaningful for all derived classes:** Every virtual method in the base class should be relevant and meaningful in every derived class. If one derived class cannot implement a base class' virtual method, then there is a problem. Perhaps the virtual method was poorly designed or perhaps the derived class does not belong in the polymorphic set.
- **All unique interface needs should be in the constructor:** When an object is created, the specific derived class is known. This is the time to expose the specific needs of a given derived class. In the above example, the interest range of the investment and loan class should be set in the constructor rather than as a standard method.

Best Practice 35.6 Extend a base class, but do not change it

Robert Martin introduced the Open-Closed Principle in 1997 and later made it the O in his SOLID design principles in 2000. He defined the open-closed principle as:

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification (Martin, 1997)

In the context of polymorphism, this means that we should work to make base classes available to adaptation in a derived class but seek to avoid modifying the base class itself. As a rule, the higher up a class is in the inheritance hierarchy, the more careful we should be in making changes.

This brings us to a very difficult problem. If we make a class too general (meaning there are too many extension points), then many will not be used and will be difficult to understand. This is called “overengineering”, the source of many performance problems and cost overruns. If we make a class too specific (no obvious ways to extend it), then the cost of making alterations is too high. This is called “underengineering” or failing to meet the client’s needs. What is the correct level of abstraction? There is no easy answer.

Overengineering is making a design too general and should be avoided

If we make a class too specific (no obvious ways to extend it), then the cost of making alterations is too high. This is called “underengineering” or failing to meet the client’s needs. What is the correct level of abstraction? There is no easy answer.

Underengineering is making a design too limited and should be avoided

Best Practice 35.7 Design for extension in the face of uncertainty

When you know that the client’s requirements are in flux or are poorly understood, then design for extension. When you know that there are many implementation details which are not yet worked out, then design for extension. In cases like these, carefully design a shared interface that can accommodate many possible variations and have several obvious extension points.

Best Practice 35.8 Avoid unnecessary abstractions when the project is well understood

When the client’s requirements are well understood, when many of the implementation details are known and there seems to be little left to discover, then do not go through the extra work of making the class unnecessarily versatile and generic. Just get the job done.

The YAGNI principle captures the necessity of avoiding overengineering. YAGNI stands for “You Ain’t Gonna Need It.” Basically, we should not do any work that is not necessary, including creating extensions or abstractions that are not immediately or likely to soon be necessary.

YAGNI: Don’t do any work until there is a clear reason for doing so

Since we don’t want to over or under engineer our designs, how do we find the Goldilocks balance that is “just right?” It takes experience, a deep understanding of the client and the client’s needs, and a good feeling for the state of the project. A good engineer is constantly looking for clues from the client and from the project. Seek out these clues and treasure them; you never know when they will be needed at a critical decision juncture in the project.

Different Implementations

The whole point of polymorphism is for derived classes to be variations or specializations of the base class. They must implement the interfaces provided by the base class while reflecting that which they represent. Note that not all inheritance relationships are appropriate for polymorphism. The possible types of inheritance relations are:

Inheritance type	Description
Reimplementation	A method from the base class has functionality unique to the derived class
Fall through	A derived class method relies on the implementation of the corresponding base class without modification
Extension	A new method is in the derived class which is not in the base class
Disinheritance	A method in a base class is not implemented in a derived class

These inheritance types are sorted from most appropriate to least.

Best Practice 35.9 Most derived class methods should utilize reimplementation inheritance

In most polymorphic scenarios, derived class methods should use reimplementation inheritance. They should use the same interface as provided by the base class but an implementation appropriate to that which the derived class is designed to represent.

Best Practice 35.10 Fall through inheritance should be used to minimize redundancy

In many cases, the base class or a parent class in the inheritance hierarchy provides an implementation enough for one or more derived classes. This is appropriate, especially in scenarios when multiple distinct derived classes have common characteristics. Note, however, that if a derived class consists solely of fall-through inheritances, then one must wonder why it is needed at all. If there is nothing unique about a class, when why does it exist?

Best Practice 35.11 Use protocol extension only for private methods and in restricted circumstances

Protocol extension is useful when a private method is added to a derived class to simplify its implementation. When a public method is added, then there is a good chance that something went amiss. With polymorphic classes, only the base class' public interfaces are utilized by the client. If another method is needed to make a derived class function properly, then how can the client access it? Extension is an acceptable strategy for nonpolymorphic inheritance but only makes sense in polymorphic situations in very restricted circumstances.

Best Practice 35.12 Avoid disinheritance in all polymorphic scenarios

The most dangerous type of inheritance relation in polymorphic scenarios is disinheritance. Here, a public method present in a base class is not allowed in a derived class. Since the client can call any public method defined in the base class, what will happen when that object is built from the derived class lacking that method? This is in violation of the Liskov Substitution Principle.

Examples

Example 35.1: Time

This example will demonstrate how to determine whether polymorphism is an appropriate strategy for a given situation, and how to represent a solution with a class diagram.

Problem

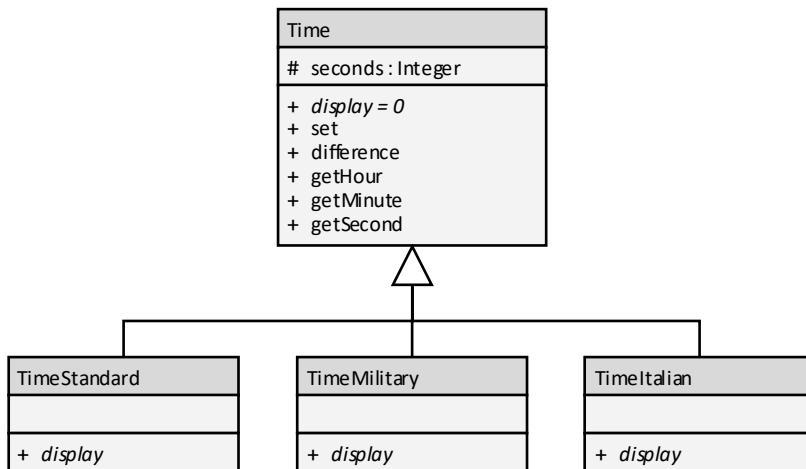
Design a class to represent three flavors of time: standard time, military time, and Italian time. The program can configure the program to use any of these varieties of time.

Solution

Every instance of a **Time** object will require the same operations: setting the time, incrementing the time by a fixed amount, displaying the time, determining how much time elapsed, and retrieving the hour, minute, and second. However, the way that time is displayed appears to be the one implementation difference. Based on this, we seem to have the shared interfaces:

Interface	Use
Set	Set the time to a certain value
Display	Display the current time, in a variety of formats
Difference	Compute the elapsed time by comparing two Time objects
Get	Get the hours, minutes, and/or seconds

We appear to have four standard methods and one virtual method. Further investigation reveals that there are actually three “get” methods: hours, minutes, and seconds. The class diagram is the following:



Example 35.2: Contacts

This example will demonstrate how to determine whether polymorphism is an appropriate strategy for a given situation, and how to represent a solution with a class diagram.

Problem

Create a class diagram for a design to match the following problem:

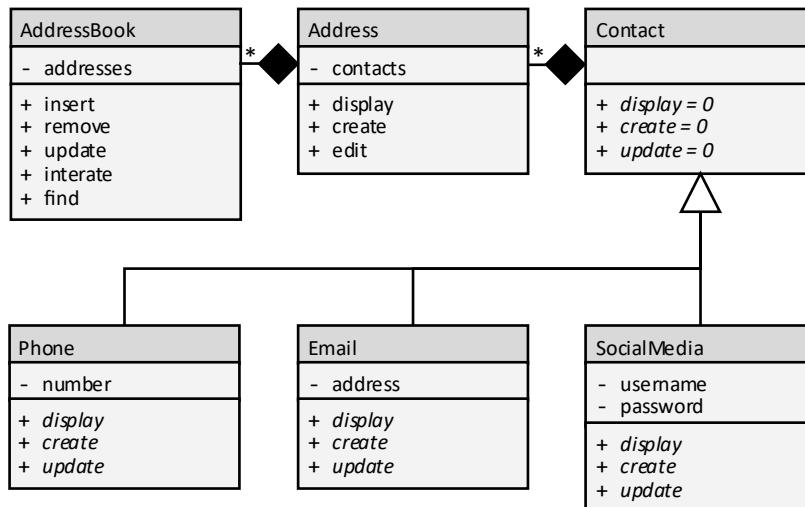
An address book contains a collection of addresses. Each address can have one or more contact. A contact could be a phone number, an e-mail address, or a social media identification.

Solution

There will be only one type of address book so **AddressBook** is not a good candidate for polymorphism. An **AddressBook** will simply be a collection of addresses.

There will be only one type of address so **Address** is not a good candidate for polymorphism. An **Address** will simply be a collection of contacts.

A contact will have three flavors: phone number, e-mail address, and social media identification. Each one will need to be created, updated, and displayed. These three operations appear to be good candidates for a shared interface. Thus, **Contact** looks to be a good candidate for a polymorphic base class.



Notice that the **Contact** base class has no attributes. In fact, it only contains pure virtual methods. Its only purpose is to provide a shared interface for the derived classes. The derived classes, on the other hand, are responsible for maintaining their own state.

Exercises

Exercise 35.1: Terms and Definitions

In your own words, define the following terms.

Term	Definition
Binding	
Binding Table	
Early Binding	
Late Binding	
Callback	
Dynamic Data Type	
Polymorphism	
Virtual Function	
V-Table	
Disinheritance	
Extension	
YAGNI	
Liskov Substitution Principle	
Overengineering	
Underengineering	

Exercise 35.2: Binding

For each of the following, state whether it pertains to only early binding, only late binding, or either early or late binding. Justify your answer.

Construct	Early or Late Binding
Dynamic Data Types	
Runtime	
Function Calls	
Dynamic Memory Allocation	
Polymorphism	
V-Table	

Exercise 35.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, cite the best practice that applies.

Fact or Fiction	Best Practice
The public interfaces in a base class should be meaningful and enough for all derived classes.	
Make classes as abstract as possible, especially when the problem domain is well known.	
When the client's requirements are poorly understood, then design for extension.	
Most derived class methods should utilize disinheritance.	
Each child class should fulfill the same contract as its parent.	
One should change a base class, not extend it.	
Any method the base class provides, the derived class should provide it as well.	

Problems

Problem 35.1: Chessboard

Consider the following problem:

A chessboard consists of a collection of chess pieces. Each chess piece has a value corresponding to its worth (10 points for a queen, 5 points for a rook, etc.), a letter indicating its type ('Q' for queen, 'R' for rook, etc.), and rules governing how it moves. In other words, you can ask a piece for a set of moves are possible from its position on the board and the other pieces that reside there.

Create a class diagram describing this chessboard class with all the related classes necessary to complete it. For the sake of simplicity, only include three piece types: a pawn, a bishop, and a queen.

Problem 35.2: Shapes

Consider the following problem:

A drawing program contains a collection of drawings, each of which is a type of shape. A shape can be many things: a circle, line, polygon, curve, square, oval, triangle, or diamond. Each shape has an outline color and width. Each shape with a volume can also have a shading color. Of course, each shape has a position and size, though the notion of size will depend on the shape.

Create a class diagram describing the notion of drawings and of shape. Note: only include attributes and methods mentioned or implied in the problem description.

Problem 35.3: Vehicles

Consider the following problem:

A list program allows a user to maintain a collection of vehicles. Each item can be one of the following: bicycle, car, skateboard, skis, motorcycle, and inline skates. There are several flavors of bicycles (road, mountain, cyclocross, BMX, unicycle, and gravel). There are several flavors of cars (truck, sports car, passenger, compact, SUV, and convertible). The user will be able to edit each vehicle, add a new instance, and display the contents. Each vehicle will have a variety of attributes specific to the type of vehicle.

Create a class diagram describing this program. Note: only include attributes and methods mentioned or implied in the problem description.

Challenges

Challenge 35.1: Financial Application

Consider the following problem:

A personal finance application is designed to track the spending and budget of a college student. The system can handle a variety of different types of accounts and produce a variety of reports.

Please do the following:

1. Identify 5–10 types of financial reports that would be useful to you personally.
2. Identify a base class to represent a report.
3. Create a class diagram to represent the base class and the 5–10 derived classes.

Challenge 35.2: List Application

Consider the following problem:

A personal list mobile application allows users to keep track of a variety of types of lists: to-do lists, grocery lists, life goals, and assignment due dates. The application allows users to add and remove lists, and even create reports created from data collected from many lists.

Please do the following:

1. Identify 5–10 types of lists that would be useful to you personally.
2. Identify a base class to represent a list.
3. Create a class diagram to represent the base class and the 5–10 derived classes.

Challenge 35.3: Video Game

Consider the following problem:

A video game allows users to fly a ship through a 3D world filled with obstacles to be avoided. Points are added as levels are cleared and prizes are discovered. Points are deducted as walls and obstacles are hit, and as time elapses.

Please do the following:

1. Identify 5–10 types of obstacles that would be useful in a game such as this.
2. Identify a base class to represent an obstacle.
3. Create a class diagram to represent the base class and the 5–10 derived classes.

Is-a and Has-a

When a class relation can be described with the phrase “is a,” then inheritance is probably the correct choice. When a class relation can be described with the phrase “has a,” then composition is probably right.

When designing classes, it is often difficult to tell which variation of class relations would result in the highest overall design quality. It may seem that there are too many options, each of which has subtle advantages and disadvantages. However, a few simple principles can greatly simplify the process.

Most class relations can be classified as *is-a* or *has-a*. *Is-a* refers to inheritance, where a derived class *is-a* specialization or variation of the base class. You can usually tell if two classes are related by *is-a* when you use that phrase to describe the relationship: A savings account “is a” type of account. *Has-a* refers to composition, where an object is a member variable in another class. You can usually tell if two classes are related by *has-a* when you use that phrase to describe them: a spaceship “has a” position.

Has-a refers to composition, when one class is a member variable of another

Is-a refers to inheritance, when one class is a version of another

There are four basic options when working with class relation design:

Option	Description
Neither	Neither <i>is-a</i> nor <i>has-a</i> is applicable
<i>Is-a</i>	Only <i>is-a</i> is applicable in this circumstance
<i>Has-a</i>	Only <i>has-a</i> can work here
Either	Either will work, but one choice is better

One of the first steps in the class-relation design process is to figure out which option is best.

Neither *Is-A* nor *Has-A*

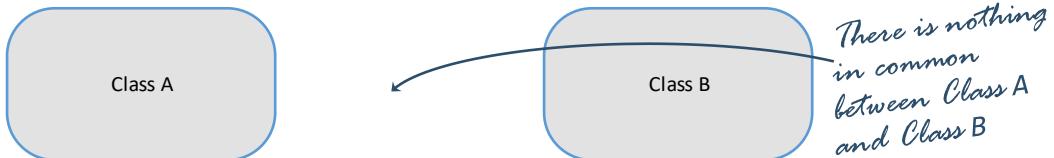
Though *is-a* (inheritance) and *has-a* (composition) are the most common flavors of class relations, there are several situations where neither will solve the problem nor is helpful. These are disjoint classes (unrelated), weakly related classes, classes related by scope (nested classes), and siblings (two classes sharing the same parent).

Disjoint Classes

A Venn diagram is a diagram that illustrates the relationship between sets. These sets could be members of a population (people who have brown hair, people who have blue eyes, people who have both, and people who have neither) as well as shared functionality. We will use Venn diagrams to illustrate the possible relationships between classes.

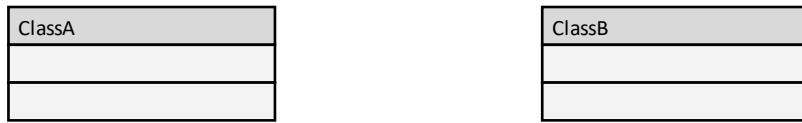
The first possible relationship are two classes that represent disjoint concepts and are generally used independently of each other.

Figure 36.1:
Venn diagram of two
disjoint classes



Here, there is nothing in common between Class A and Class B. It would therefore be inappropriate to characterize their behavior as *is-a* or *has-a*; they are two completely independent classes.

Figure 36.2:
Class diagram of two
disjoint classes

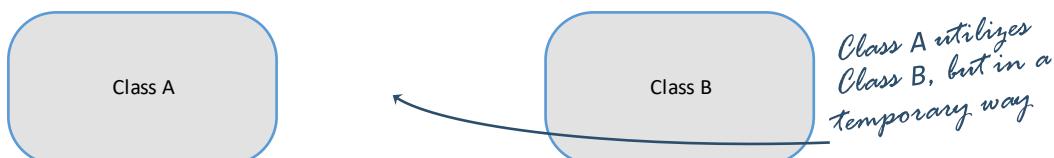


Disjoint classes are represented in class diagrams as two classes without joining lines. When implementing such classes, **ClassA** makes no reference to **ClassB** and vice versa.

Weakly Related Classes

Two classes can also contain no shared properties or attributes but still be weakly related. The Venn diagram looks the same as disjoint.

Figure 36.3:
Venn diagram of two
weakly related classes



In this case, Class A contains a local variable of type Class B. This local variable is not part of Class A's identity so it is not composition, but there is still a weak dependency relationship. This is called association and is represented with the solid line and the open arrow.

Figure 36.4:
Class diagram of two
classes related by
association



Note that it would be a mistake to use a member variable rather than a local variable: this temporary instance is not part of the **ClassA**'s identity. If we made it *has-a*, then the overall fidelity of the class would be reduced to extraneous.

Classes Related by Scope

It is sometimes the case that one class is only relevant or useful inside another. Here, the class is defined inside another, but not part of the enclosing class' identity.

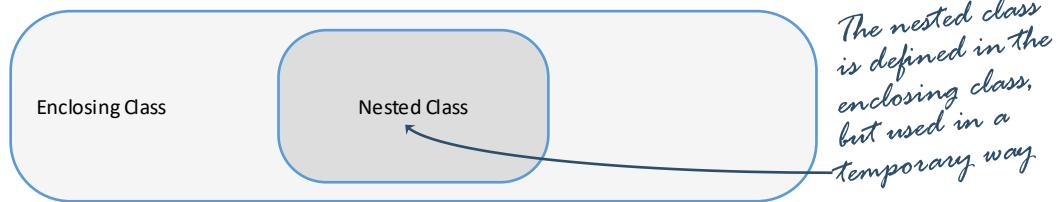


Figure 36.5:
Venn diagram of
a nested class

The Venn diagram is useful for showing containment relationships. In the case of classes, there can be many types of containment. In this case, the nested class' definition is contained within the scope of the enclosing class. However, is not part of the enclosing class' identity. In other words, it does not necessarily contain a member variable that is defined by the nested class.

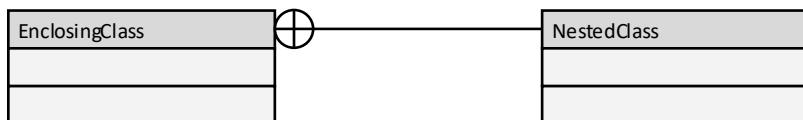


Figure 36.6:
Class diagram of
a nested class

When one class is defined within another, we call that a nested class and represent it with a pinwheel symbol in a class diagram.

Best Practice 36.1 Use a nested class when the class only makes sense in the context of the enclosing class

The modularization concept of coupling captures the notion of scope minimization: that programs are less complex when the lifespan of variables and the flow of information is tightly controlled. The same also applies to functions and classes. If a function should only be used within a class, it should be private. If a class should only be used in conjunction with another class, it should be nested.

Best Practice 36.2 Make a nested class private if only the enclosing class uses it

When a nested class is used as a utility to support the enclosing class, then it probably has no use outside the enclosing class. In situations like these, make the nested class private. Note that if the nested class could be used by others besides the enclosing class, then we must question why the class is nested at all.

Best Practice 36.3 Make a nested class public if it supports the mission of the enclosed class to the client

Consider the iterator, a class enabling the client to iterate through all the members of a collection. Iterators only make sense in the context of the collection they work with. It is therefore a common practice to make iterators a public nested class of the container. If we made the iterator a standalone class, it would be incomplete by itself. This is because it only makes sense in the context of the container it is meant to serve. A better design is to nest the iterator in the container class.

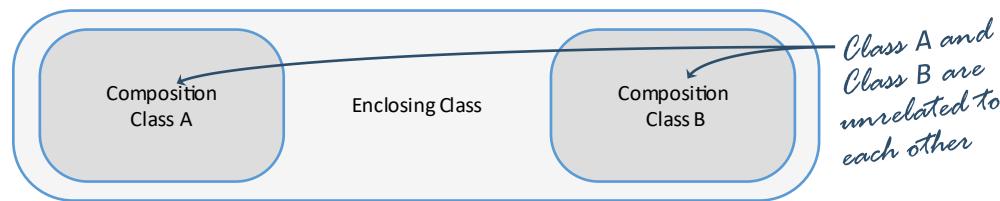
Best Practice 36.4 Use composition in conjunction with nesting when the nested class is part of the enclosing class's identity

It is common for a nested class to also be a composition class. In other words, a nested class can be utilized as a member variable in a class. In this case, *has-a* would be an appropriate description.

Composition Siblings

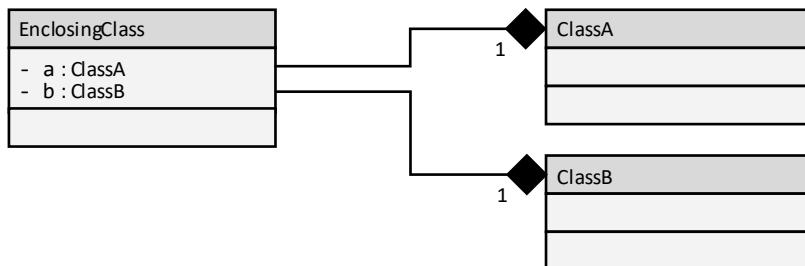
Composition siblings are when two classes are unrelated except that they find themselves next to each other as member variables in a class.

Figure 36.7:
Venn diagram of
composition siblings



In the above Venn diagram, we can see that Class A and Class B share no common properties nor attributes. They are completely independent of each other. However, they are both part of the same enclosed class. The enclosing class has a member variable of type Class A and another of type Class B. Class A and Class B appear to be unrelated; however, the enclosing class is related to Class A through composition, as it is to Class B.

Figure 36.8:
Class diagram of
composition siblings



We know we have composition when the enclosing class has an instance of **ClassA** and of **ClassB**. However, in this case, we are considering the relationship between **ClassA** and **ClassB**.

Best Practice 36.5 Composition siblings should have no knowledge of each other

As a rule, composition siblings should have no knowledge of each other. This means they demonstrate no knowledge of each other's implementation details and do not contain references of each other. To do so would violate the encapsulation principle of abstraction.

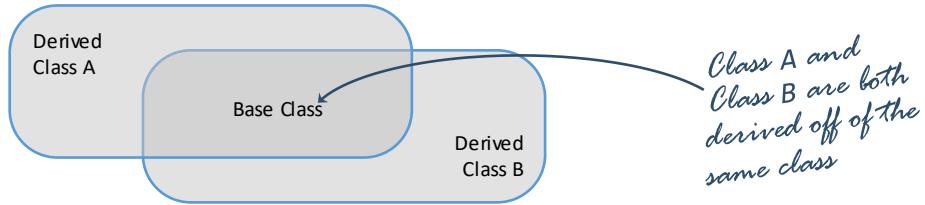
Best Practice 36.6 Treat composition siblings as disjoint classes

The principle of encapsulation states that classes should be as independent from the rest of the program as possible. In other words, the client should be shielded from any implementation details and be presented with a convenient interface. This principle applies equally when the client is a function utilizing two classes as local variables as when the client is a class utilizing two classes as member variables. The principle is the same.

Inheritance Siblings

Inheritance siblings are two classes sharing the same base class. Each derived class can have unique properties or attributes, as well as shared properties and attributes with the base class. In this case, there are two siblings.

Figure 36.9:
Venn diagram of
inheritance siblings



In the above Venn diagram, the overlapping properties and attributes are contained in the base class and the unique properties are in the derived classes. This is implemented in the following class diagram:

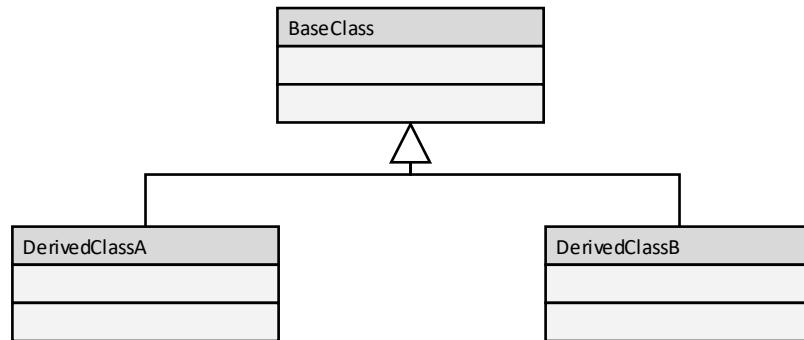


Figure 36.10:
Class diagram of
inheritance siblings

As with composition siblings, inheritance siblings should not be related by *is-a* nor *has-a*. Any shared properties should be in the base class. It turns out that it is often possible to abuse this relationship. Take, for example, a checking account and a savings account. If we were to align this to our problem domain, then they both would be derived classes to an account base class. However, it is possible to come up with an alternate design; see Figure 36.11.

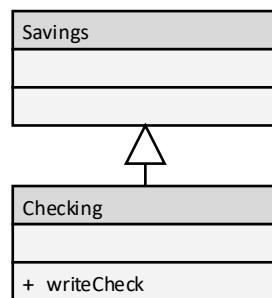


Figure 36.11:
Class diagram of an
inappropriate application
of inheritance

The above class diagram states, “A checking account is a special type of savings account where you can write checks.” This might work now, but what happens when features are added to our savings account (like interest, minimal balance, and ATM access)? Our checking class will be an unwilling beneficiary.

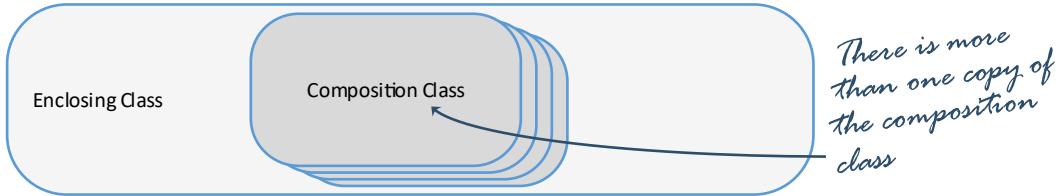
Best Practice 36.7 Do not force *is-a* or *has-a* relations; keep them closely aligned with design concerns

Whenever our class design strays from the design concern, unintended consequences such as these seem to follow.

Must Be Has-A

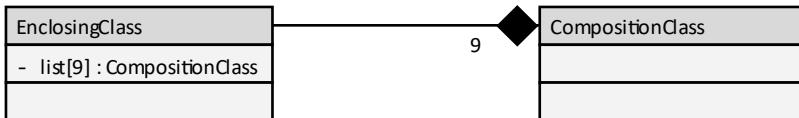
There are several class-relation scenarios where *has-a* is the only option. When one class has more than one instance of another, then *has-a* is the only option.

Figure 36.12:
Venn diagram of
composition multiplicity



In the above Venn diagram, we can see that the enclosing class contains more than one instance of the composition class. Multiplicity cannot be represented with inheritance!

Figure 36.13:
Class diagram of
composition multiplicity



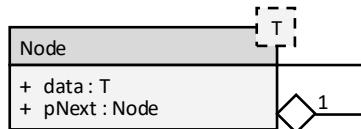
In the above example, the enclosing class contains nine instances of the other class. This relation is composition when the enclosing class governs both the creation and destruction of the other class. There are, however, cases when the enclosing class just maintains a reference. This means someone else instantiated the class and someone else will destroy it.

Figure 36.14:
Class diagram of
aggregation multiplicity



One final example of a class relation that must be aggregation is when one class contains a single reference to another, such as a linked list. Each node in a linked list points to the next node in the sequence. Since destroying one node does not necessitate the destruction of the entire tree, composition is not possible. Also, since multiple instances of a node are present in a single linked list, inheritance is not possible. Aggregation is the only option.

Figure 36.15:
Class diagram of
aggregation for a node

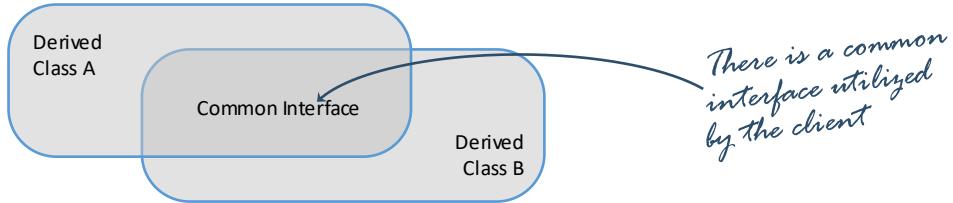


Note how this is a structure, not a class, because there are no member variables. It is called a recursive declaration because a member variable refers to itself. Recursive declarations cannot use inheritance.

Must Be Is-A

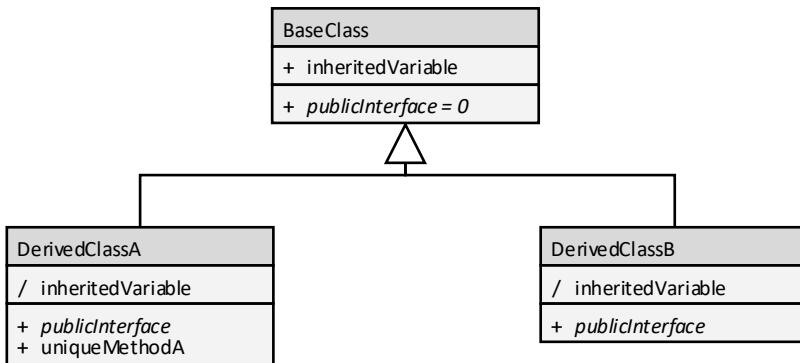
While inheritance is a useful design tool for many programming problems, it is only required for one scenario: polymorphism.

Figure 36.16:
Venn diagram of
polymorphism



Recall that polymorphism is the process of one class having more than one variation, each of which honors the same contract but has a different behavior. The contract is the common interface shared between all the derived classes as specified in the base class. When the client has a polymorphic object, it only has access to the public interfaces of the base class, each of which could potentially have a unique implementation in the derived classes.

Figure 36.17:
Class diagram of
polymorphism



In the above class diagram, the base class has an inherited variable that the derived classes have access to. There is also a public method which both the derived classes independently implemented. The client only has access to `publicInterface()`, not to the unique method for `DerivedClassA`.

It is not possible to implement polymorphism without inheritance. Some may argue that this statement is not true: you can implement polymorphic behavior with function pointers and callbacks. Note that this approach is utilizing polymorphism because the author is just hand-implementing the polymorphic mechanisms rather than using what is provided in the programming language.

Best Practice 36.8 When more than one class must share the same interface, use polymorphism

Any programming problem where more than one class must share the same interface is a good candidate for polymorphism and, by extension, inheritance. Several use cases are obvious from the problem definition (multiple car types need to work on the same road, multiple product types need to coexist on the same grocery store shelf, etc.) and some are more subtle (all printers need to honor the same contract when connected to a computer).

Either *Is-A* or *Has-A*

There are several scenarios when either composition or inheritance could represent the design concern. These scenarios involve a collection of classes sharing some common core.

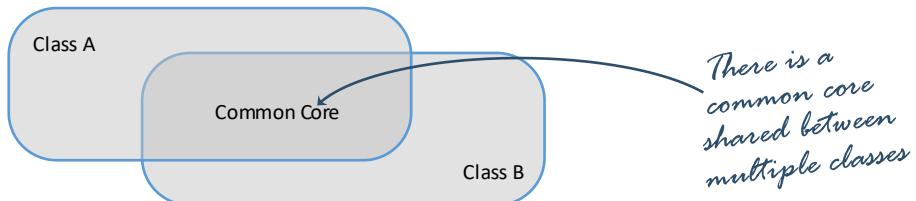


Figure 36.18:
Venn diagram of two
classes sharing a common
core

In this case, Class A and Class B have unique properties and attributes, but they share some as well. These can be implemented two ways:

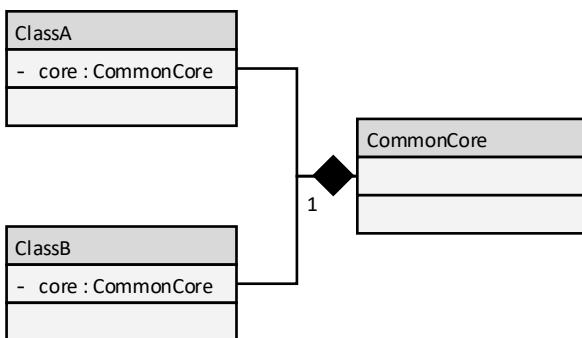


Figure 36.19:
Class diagram of a
composition solution

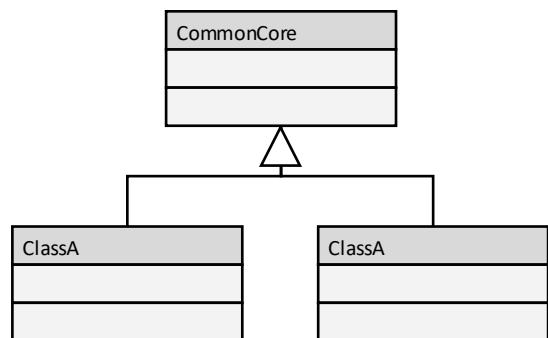


Figure 36.20:
Class diagram of an
inheritance solution

The composition implementation on the left (figure 36.19) places all the shared attributes and operations in the **CommonCore** class, each of which are reached through the public interfaces it provides. Note that **ClassA** and **ClassB** need to explicitly call **CommonCore**'s methods to honor the interfaces that the client expects. The inheritance implementation on the right (figure 36.20) places all the shared attributes and operations in the **CommonCore** base class. If the public methods provided by **CommonCore** match the interfaces the client expects, then “it just works”; no handoff is required by the derived classes to make the base class interfaces accessible.

These two implementations will both work. The question is: Which is better?

Best Practice 36.9 Class relation designs should strive to achieve complete alignment

In most cases, one design will align better with design concerns than another. Usually it is not clear which is better until a draft of each design is sketched out. In times like these, it is a good idea to consult with the subject matter expert. Oftentimes, he or she can offer insight as to which design better models the system. Designs better aligned to the design concerns typically require less refactoring when the inevitable alterations are required.

Best Practice 36.10 Prefer composition over inheritance

In those cases when no clear design is better, prefer *has-a* over *is-a*. Most software developers find composition easier to understand and bugs are easier to find. From a maintainability standpoint, composition is superior to inheritance.

To explore this question, consider a program representing drawn objects (dots, circles, lines, and rectangles). This can be represented with *has-a* in the following way:

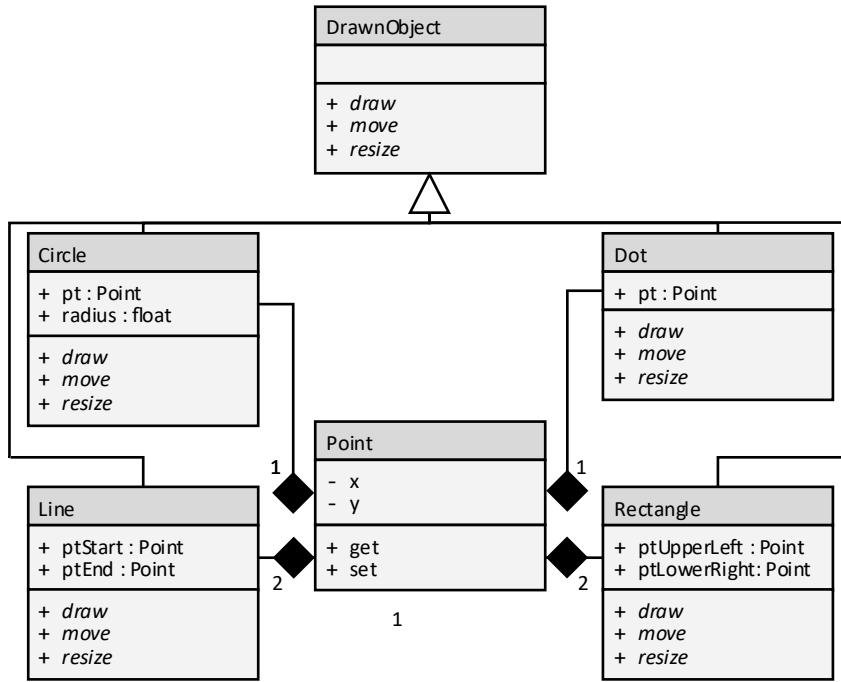


Figure 36.21:
Class diagram a
composition solution

This exact same problem can also be represented with *is-a*.

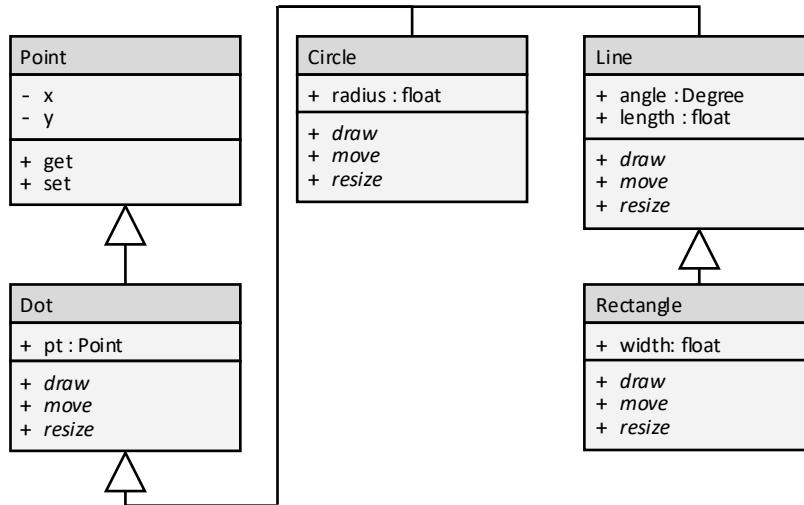


Figure 36.22:
Class diagram of an
inheritance solution

Notice that we can think of a circle as a point with a radius. We can think of a line as a point with an angle and a distance. We can think of a rectangle as a line with a width. Both designs satisfy the design concerns and work. However, they are not equally adaptable. If one were to introduce a new property (such as fill color being distinct from outline color), then it would be much more difficult with the *is-a* design than with the *has-a* design. Thus, the *has-a* design is superior in this circumstance.

Examples

Example 36.1: Address Book

This example will demonstrate how to choose an appropriate class relation strategy.

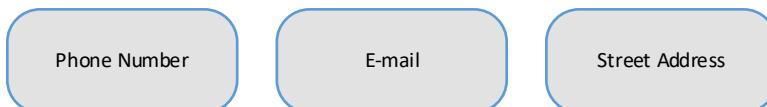
Problem

Create a class diagram for the following scenario:

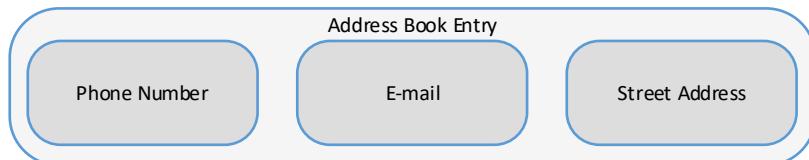
There are three types of contact information in an address book application: a phone number, an e-mail address, and a street address.

Solution

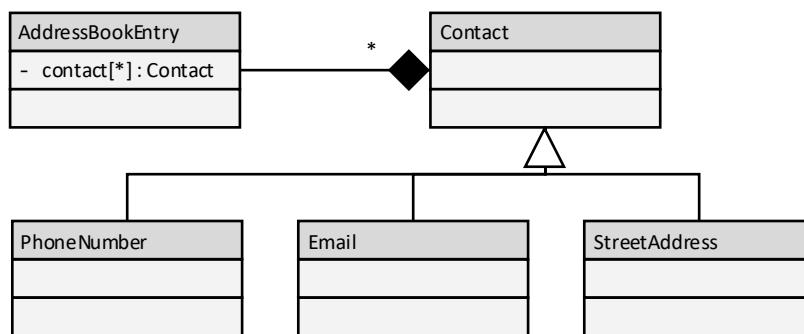
A Venn diagram for these classes is the following:



From this, we can see that we only have five possibilities: disjoint, weakly related, related by scope, composition siblings, or inheritance siblings. Since each will be part of an address book entry, they are either composition siblings or inheritance siblings. Thus, the Venn diagram is actually as the following:



Note that presumably we can have any number of phone numbers, e-mail addresses, and street addresses. The best design is to make them part of the same polymorphic collection: a collection of contacts. Since all polymorphic solutions must involve inheritance, then our three classes must inherit from the same base class.



Example 36.2: Chess

This example will demonstrate how to choose an appropriate class relation strategy.

Problem

Create a class diagram for the following scenario:

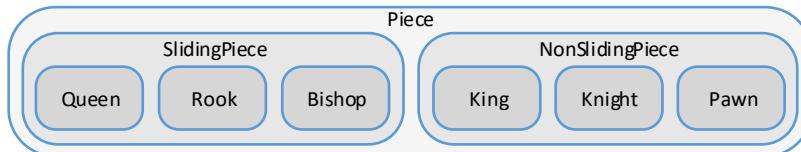
A chess program can have many types of pieces on the board: king, queen, rook, etc. Each piece knows its symbol (K, Q, R, etc.) and how it can move.

Solution

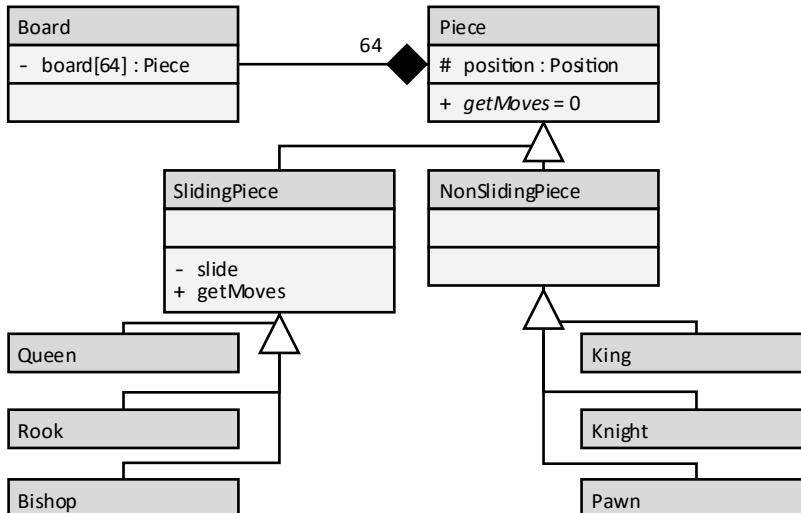
At first glance, the various chess pieces seem to have nothing in common:



However, we will quickly realize that there are actually two classes of pieces: those that slide and those that do not. In other words, you can look at the King as a Queen that slides (aside from the extra functionality afforded by the castling maneuver). All of these are special types of piece. Thus, we have the following:



From this, we can see that we only have two possibilities: composition or inheritance. Note that the board can have any number of each type of piece on the board (within reason) due to pawn promotion. Thus, polymorphism is required; we cannot know how many queens are on the board at compile time:



Note that we could avoid the extra level of inheritance by putting `slide()` in the `Piece` base class, but that would result in a method that half the derived classes do not use. A solution better aligning to the problem domain has the extra level.

Example 36.4: Velocity

This example will demonstrate how to choose an appropriate class relation strategy.

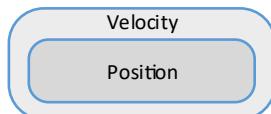
Problem

Create a class diagram for the following scenario:

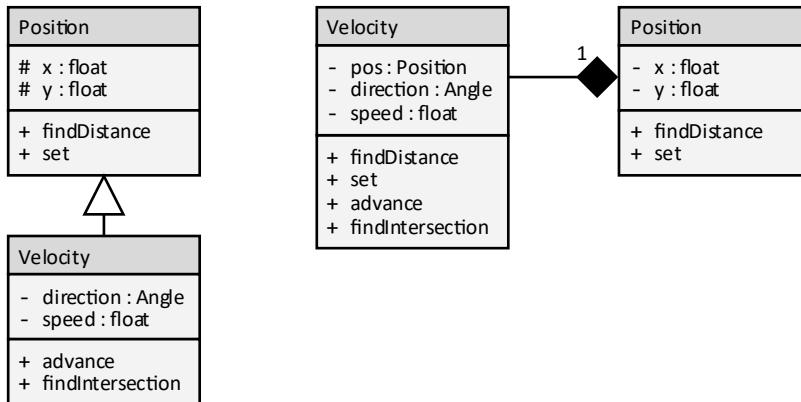
A velocity consists of a direction, a speed, and a position. Direction and speed are just scalars (floating-point numbers), but position is a class.

Solution

A Venn diagram for these classes is the following:



From this, we can see that we only have two possibilities: composition or inheritance. There is no multiplicity forcing a composition solution and there is no polymorphism forcing an inheritance solution. We could say that velocity contains a position, or that velocity is an extension of position:



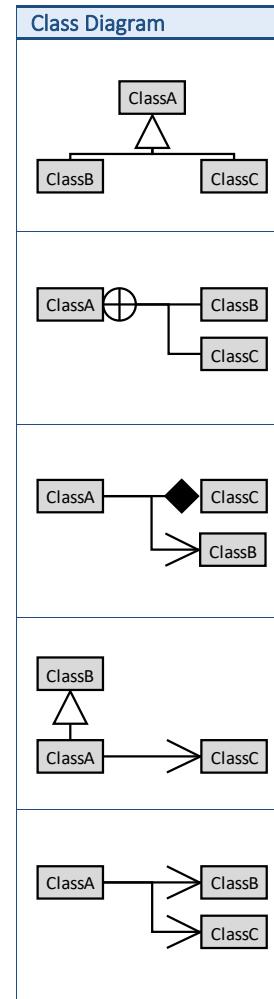
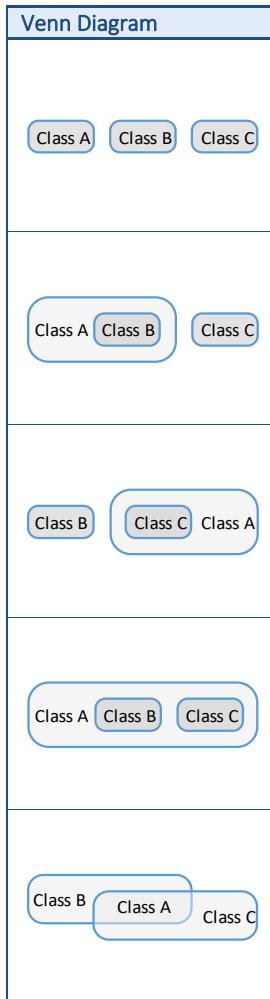
Both solutions will equally fulfill the client's need and work with equal efficiency in most programming languages. The question is: which is better?

From an alignment perspective, the composition solution is superior. Most clients would not say that a velocity is a special type of position; they would say that velocity has a position component. Even if the clients were neutral on this issue, the general rule of favoring composition over inheritance applies in this situation.

Exercises

Exercise 36.1: Venn Diagrams

Match the Venn diagram with the corresponding class diagram.



Exercise 36.2: Scenarios

For each of the following scenarios, identify whether it must be accomplished with *is-a*, *has-a*, if neither is acceptable, or if either is acceptable. Justify your answer.

Scenario	Justification
Two unrelated classes are both utilized by the same class.	
Classes are related by polymorphism.	
One class utilizes another class in a parameter passed to a public interface.	
One class is utilized many times by another class.	
Two classes share a common core.	
An iterator is defined within the container class which it supports.	
A class has a recursive declaration.	
Two otherwise unrelated classes both derive from the same base class.	

Exercise 36.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
When both options appear equally good, <i>has-a</i> is usually better than <i>is-a</i> .	
Avoid nested relations when the class only makes sense in the context of the enclosing class.	
Composition siblings should be highly interdependent.	
Freely utilize inheritance and composition whenever it is programmatically convenient.	
When more than one class must use the same interface, use composition.	
Design class relations with the design concerns in the forefront of your mind.	
Make a nested class public if it supports the mission of the enclosed class.	

Problems

Problem 36.1: Types of Notes

Create a class diagram for the following scenario:

A note-taking application can store any number of notes. Each note can be one of three types: a list, a drawing, or free-form text. These types of note all respond to edits, display requests, file open, and file save operations.

Justify your design decisions according to the principles presented in this chapter.

Problem 36.2: Guessing Game

Create a class diagram for the following scenario:

A guessing game has 24 envelopes, each with a different amount of cash. The first has \$1, the second has \$2, the third has \$4, and the 10th has \$2¹⁰ or \$1,024. The player is given an envelope at random and can choose to walk away with the cash. The dealer will offer to buy the envelope off the user for the average of all the cash in an envelope. If the user does neither, then one envelope is opened and the dealer recomputes a new offer. This process continues until the user has accepted the dealer's offer, has walked away with the envelope in hand, or the last free envelope is opened.

Justify your design decisions according to the principles presented in this chapter.

Problem 36.3: War

Create a class diagram for the following scenario:

War is played with two or more players. The deck of 52 standard playing cards are evenly distributed to all the players of the game. With each turn, all the players simultaneously lays down the top card from his or her stack. The player with the highest card wins the turn and collects all the cards. If there is a tie, then each player turns up one card face-down and another face-up. Again, winner takes all or the tie procedure continues. The game ends when one player has all the cards in the deck.

Justify your design decisions according to the principles presented in this chapter.

Problem 36.4: Skeet

Create a class diagram for the following scenario:

The game of skeet consists of a gun controlled by the user and targets. The gun has five bullets. This means that the gun can only fire when there are less than five bullets currently moving towards the target. One type of target is a clay pigeon (called the pigeon for short). Hitting a pigeon will give the user a point and missing one (when the pigeon makes it to the end of the range without being hit) will result in the loss of a point. There is also a special target which is smaller and travels faster. Hitting it will give the user five points but missing will not result on a loss. There is also a real bird which the user is forbidden to hit. If this is struck, then the user loses 10 points. The bullets and the targets all obey the laws of physics: they fall to the earth due to gravity and they slow down due to the effects of wind resistance.

Justify your design decisions according to the principles presented in this chapter.

Challenges

Challenge 36.1: Transaction

Create a class diagram for the following scenario:

In a previous version of a design for a personal financial management application, a single transaction consisted of a fixed number of properties: the date, the amount, the name, and the category. We would like to expand the functionality of the transaction class so it can handle any type of property, even properties we have not thought of when the code is compiled. Each property needs to have a sort function, a name, and a value.

Challenge 36.2: Measurements

Create a class diagram for the following scenario:

A recipe application has an ingredient list. Each ingredient on the list has a measurement, a name, and a space for any additional note. A measurement consists of a unit and an amount. There are several types of units: there are weight units, counts, length units, and volume units. Weight, length, and volume units are in metric (such as grams, liters, meters, etc.) and imperial (ounces, tablespoons, inches, etc.). There is a translation feature for each unit to convert from one unit to another.

Challenge 36.3: Galaxian

Research the 1979 arcade game Galaxian. Create a class diagram describing all the major components of this game.

Object Creation

Object creation strategies are ways to hide class implementation details from the client so the process of creating objects is easy.

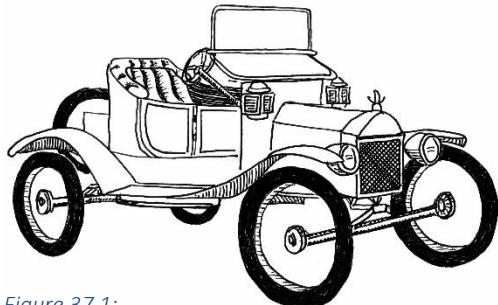


Figure 37.1:
Ford Model T

In the late 1800s, Henry Ford had a vision to make automobiles accessible to the general public. Before this time, cars were impractical and expensive. Each one was custom built to the taste of the wealthy owner. In September of 1908, Henry Ford thought he had an answer. After many years experimenting with different designs and technologies, the Model T was unveiled. A year later, Henry Ford remarked in a small gathering of salesmen, "Any customer can have a car painted any colour that he wants so long as it is black" (Ford, H. (1922), *My Life and Work*). Since all instances of the Model T were the same, the construction process for each Ford Model T was identical.

In contrast, one of Porsche's most popular models is the 911. They made over 35,000 in 2019. Even though they made so many 911s that year, only two of them were identical, and these cars were specifically made to be exactly the same for a single customer. Currently, there are 24 versions of the 911. Each version has a plethora of different options and packages. Clearly, the construction process of the Porsche 911 is more complex than that of the Ford Model T.

Creating objects can be much like creating cars. When all instances of the object are the same, a simple model like Ford's is preferred. When there are more complications, a more advanced strategy is needed. This is where creational design patterns come into play.

There are six creational design patterns:



Figure 37.2:
Porsche 911 GT2 RS

Design Pattern	Use
Simple Object Creation	When creating an object is not complicated
Factory	When multiple versions are to be created
Abstract Factory	When there are collections of versions to be created
Builder	When creating an object is a multi-step process
Prototype	When you need to create a duplicate of an object
Singleton	When the application can have only one instance of an object

Simple Object Creation

The simplest way to create an object is to use the default object instantiation mechanism built into a given programming language. This mechanism works for most situations. In fact, it is the best strategy in many situations.

Best Practice 37.1 Use simple object creation when there is only one variation

Use simple object creation when there is only one variation of an object to be created. In cases like this, abstracting or encapsulating the object creation process serves no purpose other than adding complexity to an otherwise simple process. The Ford Model T would be a great candidate for simple object creation because there are no options. Another candidate is the `Name` class in the following example:

C#

```
authorName = new Name("Helfrich");
```

Swift

```
let authorName = Name("Helfrich")
```

Figure 37.3:
Simple object creation

Notice that each `Name` object can contain different data, but they behave the same and. Things might be different if the constructors required a plethora of complex options, each of which the client would have to understand to correctly instantiate an object. As it currently stands, however, the input parameters are simple so the non-default constructor handles the object creation duties perfectly.

Best Practice 37.2 Use simple object creation when the class is not part of a polymorphic set

Recall that a polymorphic set is the collection of classes that derive from a single base class. Instantiating an object that is a member of a polymorphic set often requires a more complex object creation strategy. For most other cases, namely for classes that are not part of an inheritance hierarchy, simple object creation is the best strategy.

For example, a class representing a position on the earth has three member variables (latitude, longitude, and altitude). Since `Position` is not a base class nor a derived class, then it too would be a great candidate for simple object creation.

JavaScript

```
var new_york = new Position(40.7128, 74.0060, 33.0);
```

Kotlin

```
val new_york = Position(40.7128, 74.0060, 33.0)
```

Figure 37.4:
Creating objects with
multiple parameters

If this class were to be extended to handle different types of positions (such as positions in space, on the moon, or on a chess board), then perhaps a more elaborate object creation strategy might be necessary.

Factory

The factory design pattern is a method used to instantiate the requested variation of the object. The goal here is to insulate the client from having to know about the inheritance tree, only giving him or her the variant that is needed. There are two main variations of the factory design pattern: the factory function and the factory method.

Factories facilitate creation of objects that are members of a polymorphic set

Factory Function

The factory function is a single function that oversees the instantiation of a variety of forms of a class. It accepts as input an identifier which specifies the type of object to be created, and it returns a newly created object.

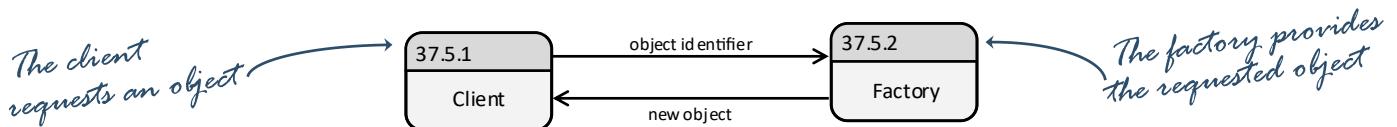


Figure 37.5:
DFD of a factory function

Back to our Porsche 911 example, there are four main variations: the Carrera, Turbo, Targa, and GT. A class diagram representing these cars could be the following:

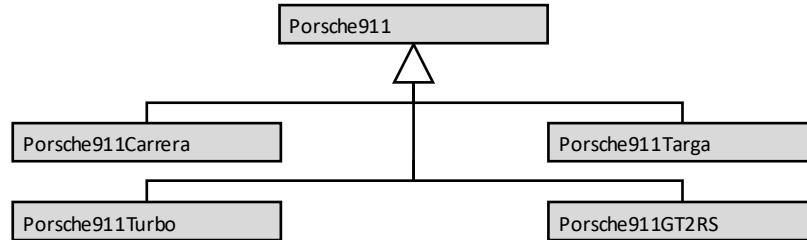


Figure 37.6:
Class diagram of four
variations of Porsche
911s to be built

If Porsche were to require all the dealers to be aware of the multitude of options that comprise each of these variations, then it would be massively confusing and problematic. Any time Porsche altered the contents of a package, then each dealer would need to be consulted and their order form would need to be updated. To avoid this problem, Porsche simply has the dealer order their car from the package name (such as "Carrera"). These package names are given a convenient number (called a model number) to simplify the ordering process.

Figure 37.7:
An enumeration of
911 models

```
C++
enum Models { CARRERA, TURBO, TARGA, GT };
```

With a model number in hand, the dealer can now order a new car by passing the identifier to the factory function. Note that this enumeration corresponds to the client's needs. It may be that each model number corresponds directly to a derived class in the inheritance tree. It is also possible that a model number may correspond to a setting passed to the class's constructor, which is considerably more complex. These implementation details are shielded from the client, who only needs to select from the provided set of options.

How would the actual code for the factory function look? It would be very simple: just a function with a big SWITCH statement that selects which derived class to create.

Figure 37.8:
A factory function
creating new 911 cars
based on model number

```
C++  
Porsche911 * factory(Models model)  
{  
    switch (model)  
    {  
        case CARRERA:  
            return new Porsche911Carrera();  
        case TURBO:  
            return new Porsche911Turbo(); ←  
        case TARGA:  
            return new Porsche911Targa();  
        case GT:  
            return new Porsche911GT2RS();  
    }  
    assert(false);  
    return NULL;  
}
```

The client receives an instance of a 911, but does not need to know which derived class created it

This design pattern has several advantages.

1. **Convenience is seamless.** It is easy for the client to obtain a new object. The client simply calls the factory function passing the model number as a parameter. The convenience of this design can be described as seamless.
2. **Abstraction is complete.** The client is shielded from the complexity of the object creation process. Details about the structure of the inheritance tree, the parameters that need to be passed to the various constructors, and even the names of the derived classes are not exposed to the client. The abstraction of this design can be described as complete.
3. **Malleability is good.** It is easy for Porsche to adjust the model offerings. Different engine and gearbox options, new paint schemes, and extra interior trim levels can be added or removed from the models without modification of client code. The malleability of the design is increased by localizing all these changes to a single function rather than spreading them throughout the client's code.

To see how this works, observe some client code ordering two Porsche models: a Carrera and a GT2 RS.

Figure 37.9:
A client ordering two cars
with the factory function

```
C++  
void PorscheEnthusiastAnnualOrder()  
{  
    ... code removed for brevity ...  
  
    Porsche911 * pCarOne = factory(CARRERA);  
    Porsche911 * pCarTwo = factory(GT);  
  
    ... code removed for brevity ...  
}
```

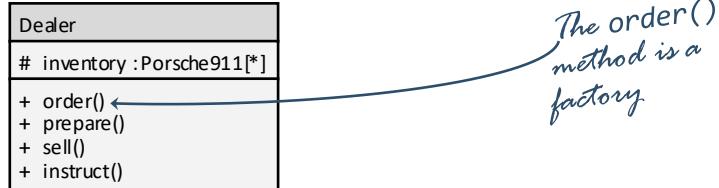
Let's say that the client wishes to order both a Carrera and a GT2 RS every year for a decade. Of course, Porsche would alter the specifics of the car from year to year as the engineers busily tune and improve the car. This would require alterations to the factory function, but the client code would remain unchanged.

When there are multiple versions of a class that the client may wish to select and when those versions can be easily enumerated, then the factory function design pattern is probably the best option.

Factory Method

Back to our Porsche example, we already have the **Car** class which contains the 911 and all its many variations. We also have a **Dealer** class which encapsulates what the dealer does with a new car before delivering it to the customer. The dealer will store the inventory of cars yet to be purchased, order new cars, prepare them for delivery, sell cars to customers, and instruct the customer how to use the various functions of their new car.

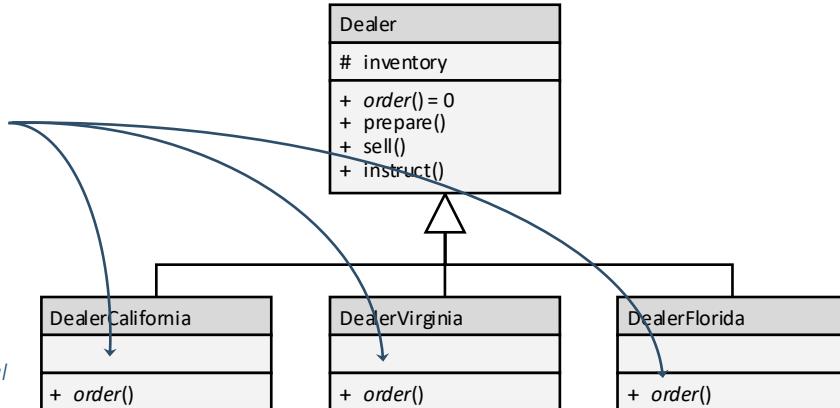
Figure 37.10:
Class diagram of a
Dealer class



It turns out that not all dealers are the same. The California dealer only sells convertibles (called Cabriolets). A race school in Virginia only sells GTs. Finally, the Florida dealer sells all the variations. The class diagram for the **Dealer** classes is the following:

Each dealer implements an order() method

Figure 37.11:
Class diagram of several dealers



Notice that each **Dealer** class has a factory method called **order()**. The base class's **order()** method is pure, meaning that it relies on the derived classes for implementation. The California and the Virginia factory methods are trivial because only one variation of car is ever ordered.

Java

```
class DealerCalifornia {
    public Porsche911 order() {
        return new Porsche911CarreraCabriolet();
    }
}

class DealerVirginia {
    public Porsche911 order() {
        return new Porsche911GT3();
    }
}
```

These factories
are trivial
because only
one variation
is ever built

Figure 37.12:
Implementation of two
dealers, each exclusively
ordering different cars

The Florida dealership offers the full range of Porsches. This factory method will look a great deal like the factory function presented earlier (in Figure 37.8).

Java

```
class DealerFlorida {
    public Porsche911 order() {
        int modelNumber = promptForModelNumber();
        switch (modelNumber) {
            case 0:                      // For Carrera models
                return new Porsche911Carrera();
            case 1:                      // For a Turbo
                return new Porsche911Turbo();
            case 2:                      // For a Targa
                return new Porsche911Targa();
            case 3:                      // For a GT2 RS
                return new Porsche911GT2RS();
        }
        return null;
    }
}
```

Figure 37.13:
Implementation of a
dealer class that orders
the full range of models

There are two things to notice about the factory method variation of the factory design pattern. First, the **Dealer** inheritance tree removes the need for the big switch statement that is part of each factory function. When the California and Virginia dealership opened, they effectively selected the model they would be selling at that point in time rather than selecting the model with a switch statement. Second, it is still possible for an individual factory method to have the functionality of a factory function. The Florida dealership illustrates that point. Note that if every dealership has exactly the same offerings of cars, then it would be best to have a single factory method in the base class rather than duplicate the code in all the derived classes. This would improve the redundancy of the design from critical to distinct.

Best Practice 37.4 Use a factory method when the clients of a class are restricted to a family of classes

When all the clients of a given class exist in a single inheritance tree, then the factory function should be a member function within that tree. Occasionally, all the client's derived classes require equal access to a factory method. In these cases, the factory method should be in the client's base class. However, when the client's derived classes are distinct, then each of the client's derived classes should have their own factory method implementation.

Abstract Factory

An abstract factory can be thought of as a collection of factory classes, each of which is customized to the specific needs of the client. To see how this works, consider how Porsche builds cars for an international market. Though all 24 models are available to any international customer, a 911 Turbo shipped to France is quite different than one shipped to England. For example, most cars in the world have the steering wheel on the left side of the car (called left-hand drive or LHD). This includes France and the United States. The remaining countries (75 out of 240) are right-hand drive (RHD), including England and Japan. This requires all cars shipped to England and Japan to have the steering wheel on the right side of the car. In addition to this, various nations have different safety and emissions regulations. Porsche must handle all these contingencies. While this may seem to hopelessly complicate the car ordering process, the abstract factory design pattern makes this quite easy.

An abstract factory is a collection of factory classes, each customized to the specific needs of the client

An abstract factory is, as the name suggests, a factory class that is abstract. In the case of Porsche, it would be a **Porsche911Factory** class that is aware of all the models and variations of the 911. For example, we will have two derived classes: **Porsche911FactoryRHD** (for England and Japan) and **Porsche911FactoryLHD** (for most of the countries in the world). The UML class diagram of this abstract factory would be the following:

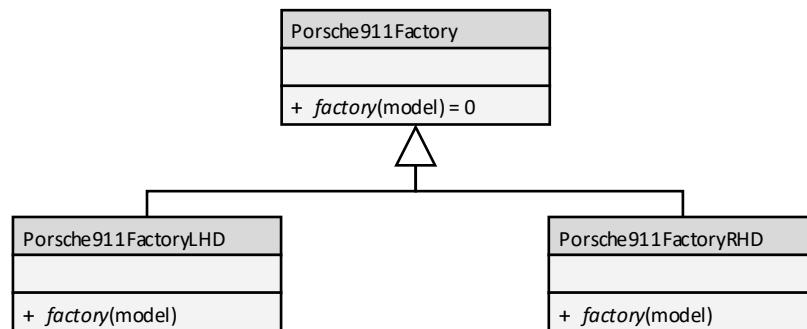


Figure 37.14:
Class diagram of two types of factories

There are several advantages to this design.

1. **Factory configuration information is specified only once.** When a dealership opens in California, they only need to specify that they want LHD cars once. This makes it impossible that a careless salesman will order a RHD car for a customer.
2. **Derived factory classes are insulated from changes in the constructed class.** Porsche can freely add and remove models from their **Porsche911Factory** class without worrying about which side of the car the steering wheel is to be placed. This makes sense; adding a Targa roof to a Turbo model does not involve the steering wheel at all! With the abstract factory, many changes can be made to the Porsche cars and their model offerings without changing code in the dealership classes.
3. **Auditing is easy.** A given Porsche dealership can easily verify that the cars they order are compliant with the regulations and needs of the customer. Since the factory derived class associated with a given dealership only contains code specific to the country, there is little chance that an important option is overlooked and not set correctly.

Pseudocode

```

Porsche911FactoryLHD :: factory(model)
    driveSide ← Left
    SWITCH model
        CASE Carrera
            RETURN NEW Porsche911Carrera(driveSide)
        CASE Turbo
            RETURN NEW Porsche911Turbo(driveSide)
        CASE Targa
            RETURN NEW Porsche911Targa(driveSide)
        CASE GT
            RETURN NEW Porsche911GT2RS(driveSide)
    
```

Figure 37.15:
Implementation of a left-hand drive factory

The above figure illustrates a factory derived class that handles only a single configuration option. The right-hand-drive version of this class would look very similar. In fact, it is so similar that one would wonder if we could optimize this design to reduce redundancy. This can be done by creating a protected factory method in the base class that does all the work, leaving the factory methods in the derived class only to specify what is unique about their variation. We will accomplish this by creating a nested configuration class. The derived factory methods will fill out a configuration option and then leave the protected factory method in the base class to instantiate objects.

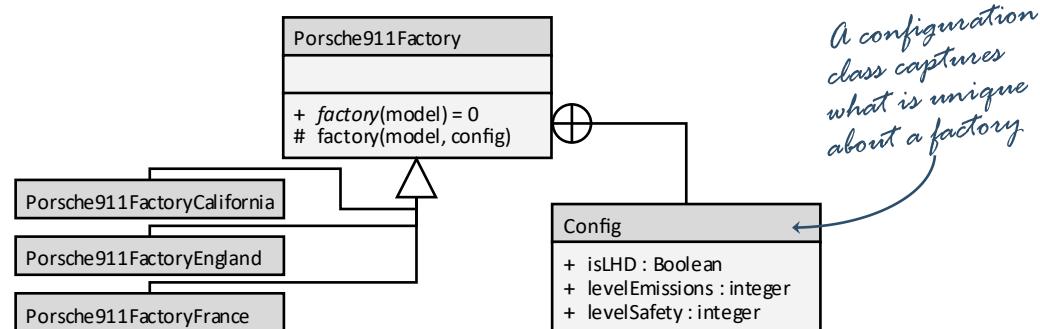


Figure 37.16:
Class diagram of an abstract factory

Now the derived factory method only specifies what is unique about cars shipped to California and leaves the factory method in the base class to construct the car. As a result, it is easy to verify that cars sent to California are correctly configured.

Pseudocode

```

Porsche911FactoryCalifornia :: factory(model)
    config.isLHD ← true
    config.levelEmissions ← High
    config.levelSafety ← Medium

    RETURN Porsche911Factory :: factory(model, config)

```

Figure 37.17:
Implementation of the California factory

Best Practice 37.5 Use an abstract factory when the client requires several variations of a factory

While the factory design pattern is commonly used in most projects, the abstract factory pattern is not as widespread. To be truly useful, the client needs to have several variations of a factory representing distinct categories of objects to be created. A video game, for example, may employ this strategy when creating elements in a playing field. Each level of the game may utilize the same basic elements, but the behaviors of the elements may be distinct on each level. Thus, each level would have its own factory method that would instantiate appropriate elements for the level.

Builder

The builder pattern is a variation of the factory method where an object is constructed in phases. In other words, it is sometimes too laborious to enumerate every possible permutation of objects that can be built. Instead, related objects are grouped into phases, each of which is assembled with a factory method.

To see how this works, we will go back to the Porsche 911. Recall that there are 24 variations. These can be broadly grouped into the model, drivetrains, and roof styles.

	Carrera	Targa	Turbo	Grand Touring
Standard	Carrera (Coupe) (Carrera) Cabriolet			GT3
All Wheel Drive	Carrera 4 Coupe Carrera 4 Cabriolet	Targa 4	Turbo (Coupe) Turbo Cabriolet	
Sport	Carrera S (Coupe) Cabriolet Carrera S			GT3 Touring
Sport All Wheel Drive	Carrera 4S (Coupe) Carrera 4S Cabriolet	Targa 4S	Turbo S (Coupe) Turbo S Cabriolet	
Extreme	Carrera GTS (Coupe) Carrera GTS Cabriolet			GT3 RS
Extreme All Wheel Drive	Carrera 4 GTS (Coupe) Carrera 4 GTS Cabriolet	Targa 4 GTS	Turbo S Exclusive Series	GT2 RS

Even though there are 24 models, they all fit into several well-defined categories:

Categories	Description
Model	This affects the number of seats and many other things
Drivetrain	Either the car has power to two or four wheels
Performance	The sport setting has firmer suspension and special aero
Roof	Coupe is a fixed roof while a cabriolet is a convertible

Rather than creating a factory which treats each of these 24 models as distinct entities, a builder function will leverage what is common between the various cars. For example, the convertible roof on the eight cabriolet models is the same. The all-wheel-drive system on the various '4' models is the same. We can leverage these similarities by creating four factories: model factory, drivetrain factory, performance factory, and roof factory. A builder function would then construct a car by assembling the parts from the four factories. We will start with the model factory. We will put our model factory in a factory method which will be part of a builder class.

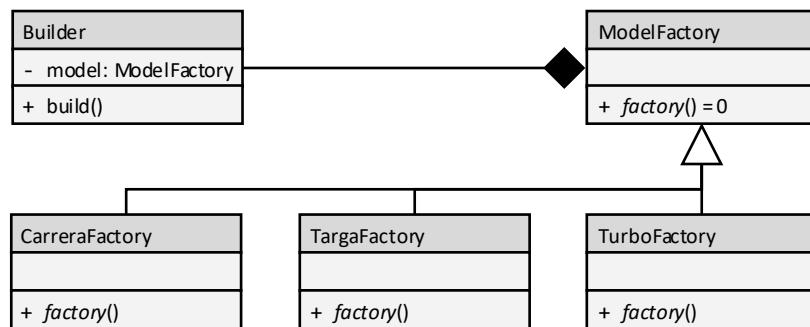


Figure 37.18:
Class diagram of a one
part of a builder class

We will need a total of four factories, each of which look somewhat like our model factory. Each factory will be a private member variable in our builder class.

Simple Builder Class

We will have a single public method in our **Builder** class which the client will call to order a car. The class diagram is the following:

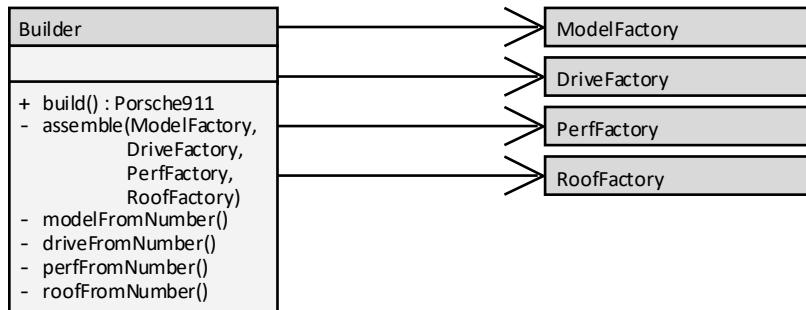


Figure 37.19:
Class diagram of a simple
builder class

The only method in the builder class that can be complicated is the **build()** method itself. This method will need to determine the various model, drive, performance, and roof options from the model number. From this, it will create the necessary components and assemble them into the returned object.

Pseudocode

```
Builder :: build(modelNumber)
modelType <- modelFromNumber(modelNumber)
driveType <- driveFromNumber(modelNumber)
perfType <- perfFromNumber(modelNumber)
roofType <- roofFromNumber(modelNumber)

model <- ModelFactory(modelType)
drive <- DriveFactory(driveType)
perf <- PerformanceFactory(perfType)
roof <- RoofFactory(roofType)

RETURN assemble(model.factory(), drive.factory(),
perf.factory(), roof.factory())
```

Figure 37.20:
Pseudocode of the build
method of a Builder class

The simple builder class has several qualities:

1. **Distinct Redundancy.** There is only one place in the code which knows how to build a roof for a Porsche 911. In the previous designs, eight factories would need to know how to build a convertible: Cabriolet factory, Carrera 4 Cabriolet factory, etc.
2. **Enabling Adaptability.** It is very easy to add a new roof type to the Porsche 911 lineup. If new roof options were to be added to the 911 lineup (such as a speedster option), then only **RoofFactory()** would need to be changed. It is also easy to create a new 911 model through assembly of different components (such as a Porsche 911 GT2 RS Cabriolet).
3. **Extraneous Fidelity.** There is one problem with the **Builder** class as presented here. **Builder** does two things: it determines which components are assembled to create a given object, and it does the assembly itself. A better design would separate these two concerns into two independent classes.

To address the fidelity problem with the simple builder class, often a second class is needed: a director.

Director

The first step addressing the fidelity problem with the **Builder** class is to remove the component selection functionality. The class diagram is now somewhat simpler.

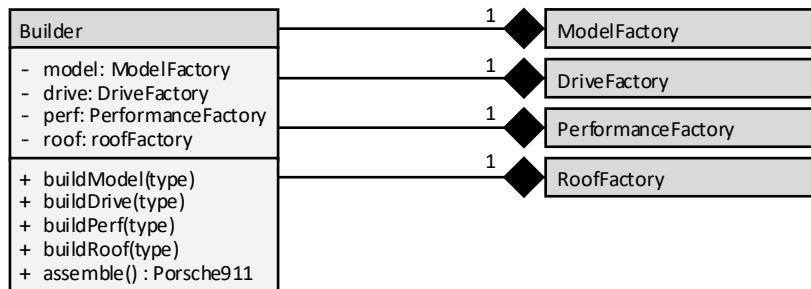


Figure 37.21:
Class diagram of a builder
class without director
functionality

Notice that this **Builder** class has no notion of model numbers. It only knows about model types, drive types, performance types, and roof types. With these pieces, it will assemble a finished object with the **assemble()** method. The only thing that is missing is a **Director** class. This class will accept the model number as input and, from it, figure out the various drive and roof types.

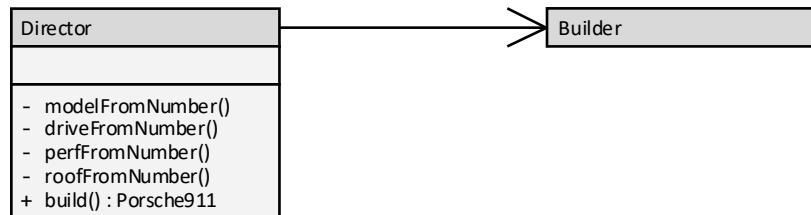


Figure 37.22:
Class diagram of a
director class utilizing a
builder

The **Director** is now the only class that understands the relationship between the various models and the options that are assembled to constitute a model. For example, the **Director** class will understand that the “Carrera 4S Cabriolet” model takes the Carrera body, the all-wheel-drive drivetrain, the convertible roof, and the more powerful engine.

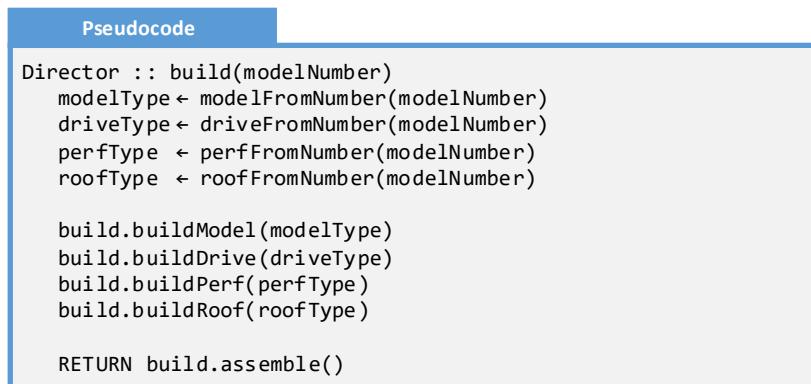


Figure 37.23:
Pseudocode of the build
method of a Director class

Notice that the complexity of the object creation process is shielded from the client; they simply ask for an object and receive it.

Best Practice 37.6 Use a builder when object creation can be subdivided into several components or phases

When object creation is a multi-step process, then the builder design pattern is a good fit. This is true even when there are just two or three steps.

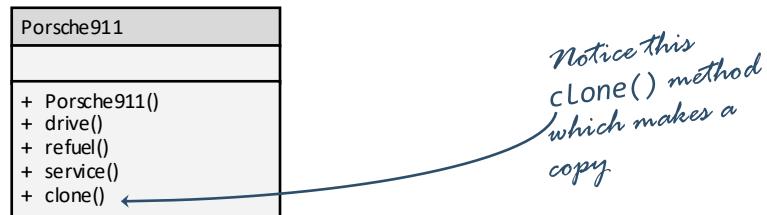
Prototype

The prototype pattern is a mechanism used to create a copy of an existing object. Note that for simple variables such as an integer or a Boolean value, there is no need to create a special prototype function or class—one can simply use the assignment operator to make a copy. Simple classes can also utilize the assignment operator when their attributes use primitive data types. Most programming languages facilitate object duplication by simply copying the member variables. This, unfortunately, does not work for complex classes. A **list** data structure, for example, consists of a linked list. Each node in the linked list is connected to the others through a collection of pointers. Any duplication of the **list** needs to be aware of the implementation details of the linked list and needs to copy each node one at a time. To ensure this is done correctly, a prototype method needs to be added.

A prototype is a mechanism to facilitate creating copies of an existing object

Back to our car example, many prospective customers of Porsches are attracted to the high-performance characteristics of the cars. This performance potential, coupled with drivers who appreciate speed, results in the occasional accident. This prompts customers to approach their Porsche dealer with the request to recreate their beloved (and now destroyed) car. Fortunately, Porsche has thought ahead. They have attached a **clone()** method to each order allowing them to produce a new copy of the car the customer has previously purchased.

Figure 37.24:
A class diagram depicting
a class with a clone
function



The **clone()** method facilitates the client creating a copy of an object. To accomplish this, the **clone()** method must be aware of all the implementation details about the object and of any class the object references. Ideally, these implementation details are hidden from the client, making it impossible for him or her to know if the **clone()** method simply copies member variables or does something more complex such as our **list** data structure previously mentioned.

Best Practice 37.7 Whenever a client needs to copy an object, provide prototype functionality

Even when a class is trivial and utilizes simple member variables, we should provide a **clone()** method. This way, if the class needs to change to have more complex functionality, then no client code needs to change.

Best Practice 37.8 Use the assignment operator and copy constructor if your programming language supports it

Many programming languages allow a class to define the assignment operator and the copy constructor. If this is true with your language, then the **clone** method should be under those names.

Singleton

The final creational design pattern is called the singleton. Unlike all the other creational design patterns which facilitate the client constructing a new copy of an object, the singleton prevents such things. In fact, the singleton's job is to ensure that no more than one copy of an object exists. The mechanics of how to create a singleton varies from one programming language to another. A few examples will be presented. First, you can create a singleton in C++ with a private constructor.

```
C++
class Singleton
{
private:
    static Singleton * pSingleton;
    Singleton() {} ←

public:
    static Singleton * getInstance()
    {
        if (pSingleton == NULL) ←
            return pSingleton = new Singleton();
        else
            return pSingleton;
    }
};

Singleton * Singleton::pSingleton = NULL;
```

The instance pointer pSingleton must be static so all objects share it

The constructor must be private so only a method can create a Singleton instance

getInstance() ensures there is only one copy

Figure 37.25: A C++ singleton class

Observe how the constructor is private, making it impossible for the client to instantiate a **Singleton** object. Another component of C++ is a static member variable. Because it is **static**, there is only one instance of the member variable shared between all objects. This instance must be initialized at compile time. Finally, the **getInstance()** method returns an instance if one already exists, or creates an instance if one does not exist (it can do this because it has access to the private constructor). These same components exist in Java.

```
Java
class Singleton {
    private static Singleton singleton = null;
    private Singleton() {}

    public static Singleton * getInstance()
    {
        if (singleton == null)
            return singleton = new Singleton();
        else
            return singleton;
    }
}
```

Figure 37.26:
A Java singleton class

The singleton design pattern uses access control mechanisms and static scope modifiers to make it impossible for the client to instantiate a **Singleton** object directly. If they attempt to do so, the result would be a compile error. One could argue that modern programming languages should explicitly accommodate this pattern rather than having to use such a roundabout way to accomplish it.

While the mechanics of a singleton are somewhat obscure, their use is even more so. It turns out that there is a considerable debate in the OO design community over whether a singleton is an acceptable design pattern.

The problem with a singleton is that it is roughly equivalent to a global variable. Rather than having a single global instance, would it not be better to just pass the instance to various functions as a parameter? However, there are a few scenarios where a singleton is probably the best tool for the job.

“There can be only one”
(Highlander, 1986)

Best Practice 37.9 Use a singleton for logging

Many applications need to contain logging functionality so transactions can be audited, or events can be reconstructed. Since logging code might need to be added in almost any function or method in a program, the scope of a logging variable needs to be immense. Rather than pass a logging instance to every class and function in the application, the coupling of the program can be reduced with a singleton. This design has two advantages: it removes the amount of unnecessary parameter passing for those classes and functions that do not need to participate in logging, and it makes it easier to add logging to a given location in the program.

Best Practice 37.10 Use a singleton for external interfaces

When an application initiates two or more concurrent network or file interfaces to the same resource, a variety of bad behaviors may result. The interface could move into an error state, data could be lost, or the integrity of data could be compromised. A singleton can prevent such conditions by ensuring that exactly one instance exists and by warning the client that a connection is already open. While there certainly are other concurrency mechanisms which offer similar assurances (such as semaphores and mutexes), singletons are often part of the solution.

Best Practice 37.11 Use a singleton to remove a global variable

Global variables are harmful programming constructs. There are two main reasons for this. First, it is difficult to tell what part of the program is reading a variable. Second, it is difficult to tell who most recently set a variable to a given value. While singletons have global scope like that of a global variable, they have an advantage – it is easy to add getters and setters which facilitate answering these two critical questions. That being said, moving a global variable to a singleton is usually only part of the solution. A more comprehensive solution would be to reduce the scope of the variable.

Examples

Example 37.1: Simple Object Creation

This example will demonstrate simple object creation.

Problem

Select an appropriate object creation strategy for the following scenario:

A class called **Money** represents American currency.

Justify why the strategy is the best and demonstrate how it will be implemented.

Solution

The **Money** class is a good candidate for simple object creation because there is no inheritance tree. For this reason, the factory, abstract factory, and builder are overkill—there is only one variety of **Money** and there is no inheritance tree. The **Money** class is potentially a candidate for the prototype pattern because the client may wish to copy a **Money** object and that object may become nontrivial in the future. It is not a good candidate for a singleton because the program can contain any number of **Money** objects.

The best design is to use simple object creation with object cloning capabilities. The **Money** class should simply provide a default constructor (setting the value to \$0.00), a few non-default constructors (one taking cents as an integer and another taking dollars as a floating-point number), a copy constructor, and an assignment operator.

Money
- cents : Integer
+ Money()
+ Money(dollars : float)
+ Money(cents : Integer)
+ Money(rhs : Money)
+ operator=(rhs : Money)
...

This design honors Best Practice #37.1 (use simple object creation when there is only one variation) and Best Practice #37.2 (use simple object creation when the class is not part of a polymorphic set).

Example 37.2: Factory

This example will demonstrate a factory method.

Problem

Select an appropriate object creation strategy for the following scenario:

A drawing program allows the user to place various shapes on a canvas: a dot, a line, a rectangle, a triangle, and an oval. The canvas can contain any number of shapes, and in any combination.

Justify why the strategy is the best and demonstrate how it will be implemented.

Solution

Simple object creation is a possibility for creating shapes, but the number and variety of shapes would place an undue burden on the client. A factory method is a good candidate because each shape is part of a polymorphic set. The abstract factory pattern is not a good fit because there are not variations of individual shapes. The builder pattern would not be a good fit because instantiating a shape is a trivial, one-time event; it is difficult to imagine how it could be better done in phases. The prototype pattern may be useful because the program may need to clone shapes. Finally, the singleton pattern would not be a good fit because more than one instance of a shape will be needed.

The factory method will take a shape type (an enumeration) as a parameter and return a newly created object.

Pseudocode

```
shapeFactory(shapeType)
SWITCH shapeType
CASE DOT
    RETURN new Dot()
CASE LINE
    RETURN new Line()
CASE RECTANGLE
    RETURN new Rectangle()
CASE TRIANGLE
    RETURN new Triangle()
CASE OVAL
    RETURN new Oval()
```

This design honors Best Practice 37.3 (use a factory function when the client needs to select from a variety of different classes). Here, each class represents a shape type.

Example 37.3: Abstract Factory

This example will demonstrate the abstract factory design pattern.

Problem

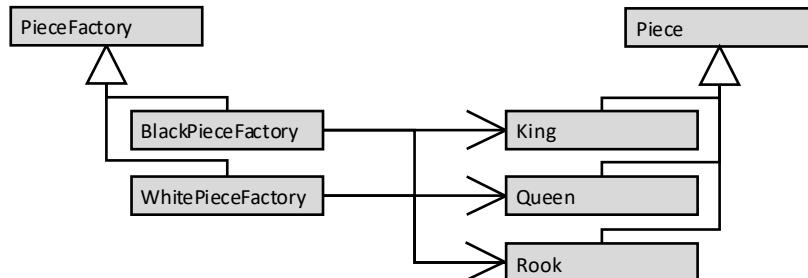
Select an appropriate object creation strategy for the following scenario:

A chess game has several types of pieces (king, queen, rook, knight, bishop, and pawn) that are either black or white.

Justify why the strategy is the best and demonstrate how it will be implemented.

Solution

Simple object creation is a candidate because pieces are only created in two situations: when the board is set up, and when a pawn is promoted to a queen. The factory pattern is also a candidate because it allows the client to ask for a knight without even knowing the name of the derived class. However, there is a bit of redundancy in the piece creation process when the color is specified. The abstract factory is a nice fit here, eliminating the redundancy between the white and black pieces. The builder pattern is overkill because a piece is created in a single distinct phase. There is little need to copy a piece, so the prototype pattern does not apply. Finally, there is no need to restrict the existence of duplicate pieces, so the singleton pattern does not apply. The solution is to create two factories: one for white pieces and one for black.



This design makes it possible to easily generate a new board. When the client creates a new piece, its position is already known. Note that there are 32 pieces on the chess board (with 16 enumerations for each side) and only six derived classes from **Piece**. This means that the king's rook and the queen's rook will both call the **Rook** constructor but will pass different initial positions.

Pseudocode

```
WhitePieceFactory :: factory(pieceType)
SWITCH pieceType
    case QUEENS_ROOK
        RETURN new Rook(0, 0, WHITE)
    case KINGS_ROOK
        RETURN new Rook(7, 0, WHITE)
    ...

```

This design honors Best Practice 37.5 (use an abstract factory when the client requires several variations of a factory) because having a variety of factories simplifies the piece creation process.

Example 37.4: Builder

This example will demonstrate the builder design pattern.

Problem

Select an appropriate object creation strategy for the following scenario:

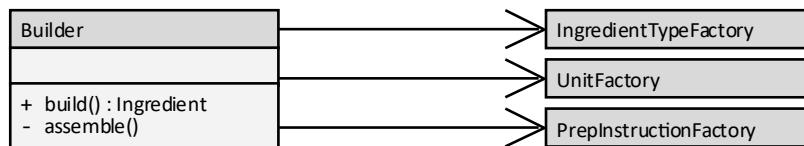
An ingredient in a recipe program consists of an ingredient type, quantity, units, and preparation instructions.

Justify why the strategy is the best and demonstrate how it will be implemented.

Solution

Simple object creation is probably not the best strategy here because instantiating an ingredient object is a complex, multi-step process. A factory would not be much better than simple object creation because there are so many permutations of ingredient types, units, and preparation instructions. The abstract factory pattern would not have an advantage over the factory unless there were different styles of ingredients for different nationalities. The builder pattern would be a good candidate because each part of an ingredient could be a separate factory. One might want to include prototype functionality to facilitate copying of ingredients. Finally, the singleton offers no advantage in this scenario.

This solution will employ three factories: an ingredient factory, a unit factory, and preparation instruction factory.



When the client creates an ingredient, he or she will specify the ingredient type (oregano, flour, etc.), the unit type (liters, count, ounces, etc.), and the preparation type (chop, sauté, mix, etc.). Though there are thousands of permutations of ingredients, there are only a few dozen options for the client to specify. This will be easier for the client and provide more flexibility. We could even provide a director class to simplify creation of common combinations while retaining access to the Builder when the client needs something special.

This design honors Best Practice 37.6 (use a builder function when object creation can be subdivided into several components or phases) because the ingredient creation process happens in three distinct phases.

Example 37.5: Prototype

This example will demonstrate the prototype design pattern.

Problem

Select an appropriate object creation strategy for the following scenario:

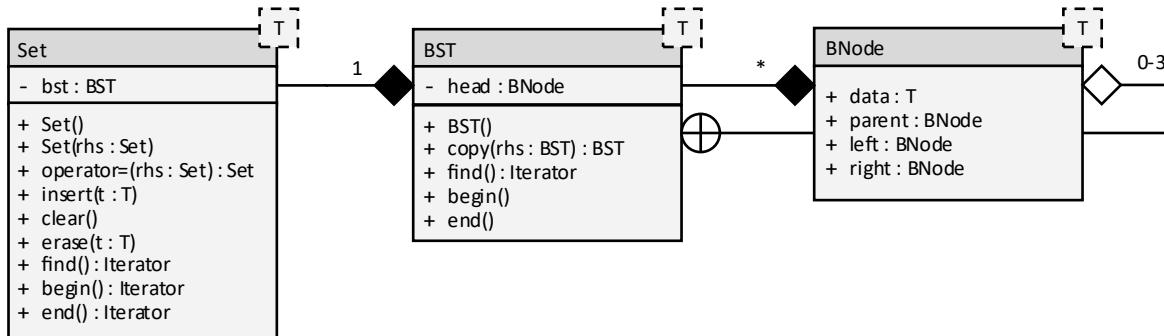
A set is an abstract data type built on top of a binary search tree. The client needs to be able to easily copy or clone a set.

Justify why the strategy is the best and demonstrate how it will be implemented.

Solution

Simple object creation is a good choice here because a set is not built from an inheritance tree. For this reason, the factory pattern and the abstract factory pattern are not good candidates. The builder pattern would not help because object creation cannot be subdivided into phases. The prototype pattern is a good fit because copying a set is a common action and the copy process is nontrivial. Finally, the singleton is not a good fit because it is likely that the client will need more than one set for their application.

This solution will define both the copy constructor and the assignment operator for the set to make it as easy as possible for the client to make copies.



The client is only aware of the **Set** class, not being exposed to the underlying **BST** or **BNode** class. Through the use of the prototype design pattern, copies of **Set** can be made without concerning the client with these implementation details.

This design honors Best Practice 37.7 (whenever a client needs to copy an object, provide prototype functionality) and Best Practice 37.8 (use the assignment operator and copy constructor if your programming language supports it).

Example 37.6: Singleton

This example will demonstrate the singleton design pattern.

Problem

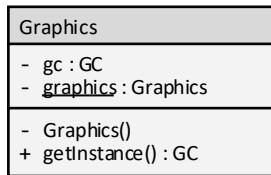
Select an appropriate object creation strategy for the following scenario:

A video game has to keep an instance of the graphic context (GC). There must be exactly one instance of this GC object during the game; if there are more, the game will display multiple windows on the screen.

Justify why the strategy is the best and demonstrate how it will be implemented.

Solution

Instantiating a GC object involves creating a new window. Therefore, the object creation strategies that involve creating new windows do not apply (simple object creation, factory, abstract factory, builder, and prototype). Since we would like to guarantee that only a single GC object exists in the program, and we would like this GC object to be available anywhere in the game, the singleton is a natural choice.



Notice that the constructor is private, an instance member variable is static (the underline), and the `getInstance()` method returns a graphics context. Through the use of this singleton, the client can easily request the GC from anywhere in the program without worrying about multiple GCs being instantiated and without having to pass this object to every function and class in the application.

This design honors Best Practice 37.10 (use a singleton for external interfaces). Here, the external interface is the drawing engine used by the video game.

Exercises

Exercise 37.1: Creational Design Patterns

From memory, name and explain the six creational design patterns.

Name	Description

Exercise 37.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

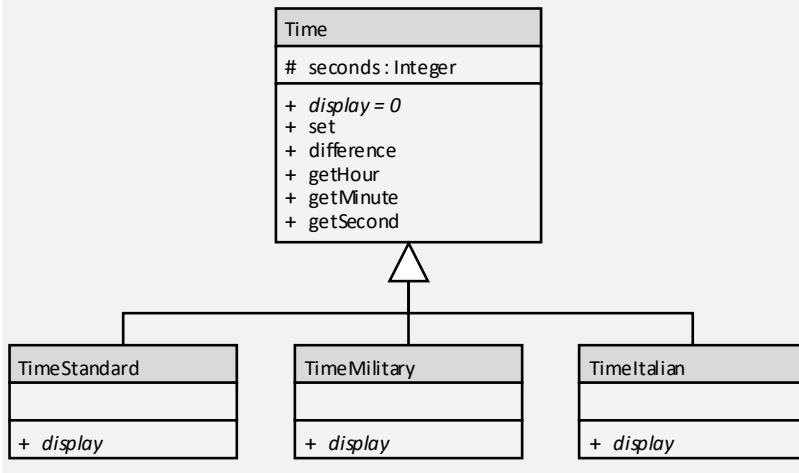
Fact or Fiction	Justification
Most programming languages have direct support for singletons.	
The builder pattern is useful for creating objects in phases.	
An abstract factory is a factory class with a pure virtual build method.	
A factory function takes an object as a parameter, which is the object to be duplicated.	
The singleton facilitates duplication of objects.	
A director often accompanies a builder.	

Problems

Problem 37.1: Time

Select an appropriate object creation strategy for the following scenario:

A **Time** class has three variations: standard, military, and Italian. The class diagram for the Time class is the following:



Justify why the strategy is the best and demonstrate how it will be implemented.

Problem 37.2: Date

Select an appropriate object creation strategy for the following scenario:

A **Date** class has five major formats: 1) the long format with the day of the week, day of the month, month name, and year; 2) the medium format with the day of the month, month name, and year; 3) the short format with the day of the month, the month number, and the year; 4) the month name and year format; and 5) the year-only format. Of these five formats, each one has a version specific to the language: English, German, Italian, French, and Spanish. A single client is likely to use several formats but only one language.

Justify why the strategy is the best and demonstrate how it will be implemented.

Problem 37.3: D&D

Select an appropriate object creation strategy for the following scenario:

Dungeons & Dragons (D&D) is a dice-based role playing game involving a dungeon master (the individual who orchestrates the game) and several players. Each player controls his or her character through a series of adventures. A key part of the game is creating a character that complements the strengths and weaknesses of the other characters in your band. A character consists of a race, a class, and a collection of different abilities. Some character traits are selected by the player, others through a roll of dice.

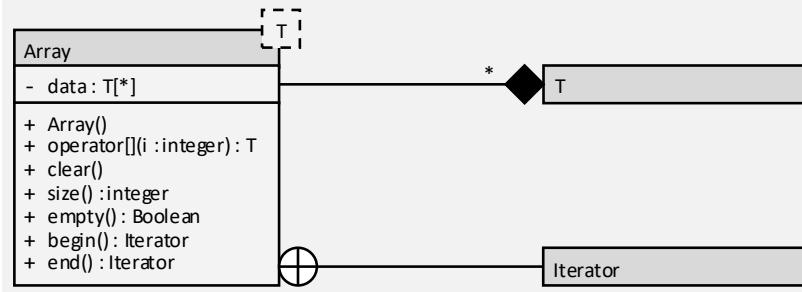
Our job is to facilitate the creation of a D&D character.

Justify why the strategy is the best and demonstrate how it will be implemented.

Problem 37.4: Array

Select an appropriate object creation strategy for the following scenario:

An array is an abstract data type that can hold only a fixed number of elements. The class diagram for the array is the following:



Justify why the strategy is the best and demonstrate how it will be implemented.

Problem 37.5: Error Logging

Select an appropriate object creation strategy for the following scenario:

I have a complex application that can encounter a variety of errors: critical errors causing the application to halt, serious errors requiring user intervention, recoverable errors that the development team should know about, and status reports which help administrators audit the application. Errors can occur anywhere in the program. Each error is written to the same log file. It is important that only one instance of the error logging file code exists to protect entries from being overwritten. It must also be very easy to add error code to the application.

Justify why the strategy is the best and demonstrate how it will be implemented.

Challenges

Challenge 37.1: Financial Accounts

Consider the following scenario:

A financial program contains a collection of accounts: checking, savings, credit card, retirement (401(k)), education (529), car loan, and mortgage. Each account type has a collection of parameters which must be specified at creation time (interest rates, annual fees, transaction fees, etc.).

Please do the following:

1. Find out what types of accounts your financial institution offers. For each account, determine the parameters that describe each account.
2. Select an appropriate object creation strategy that facilitates creation of a new account at the user's request. Justify your strategy selection.
3. Create a class diagram and pseudocode to describe how the account creation process will work.
4. Write some code that walks the user through the account creation process. The code will first prompt the user for the account type, then for any of the necessary parameters.
5. Implement the account creation code from Step 2.

Challenge 37.2: Orbit Simulator

Consider the following scenario:

An orbit simulator has several types of object: the earth, Sputnik, a GPS satellite, the Hubble Space Telescope, the SpaceX Dragon, and the SpaceX Starlink. Each of these satellites draws differently and, when hit with orbital debris, breaks into a different number of fragments. In addition to this, a satellite can exist in several orbits: low earth orbit (2,000km above the earth), standard orbit (20,200km), or geosynchronous orbit (35,786km).

Please do the following:

1. Select an appropriate object creation strategy that facilitates creation of a new satellite at the user's request. Justify your strategy selection.
2. Create a class diagram and pseudocode to describe how the satellite creation process will work.
3. Write some code that facilitates the satellite creation process. Note that any satellite can exist in any orbit.

Algorithm Abstraction

Algorithm abstraction is the process of taking a potentially complicated algorithm and reducing it to its components, allowing the components to be reused and reassembled to create new combinations.

The process of making the Italian dish penne with marinara sauce is straightforward. You first prepare the vegetables and cook them. You then prepare the meat and cook it. Finally, you prepare the carbohydrates (pasta in this case) and cook it. When finished, you combine the three parts into a single meal. The process is very similar when making a chicken burrito (vegetables → meat → carbohydrates → combine). Note that the vegetables are different, as are the ways they are prepared and cooked. The same can be said about the meat and the carbohydrates. However, the overall algorithm for creating many meals is essentially the same. If one were to create a cooking program, then they should leverage these similarities to avoid redundant code. This can be accomplished using a collection of techniques called algorithm abstraction.

Algorithm abstraction is the process of taking a potentially complicated algorithm and reducing it to its components. These components can then be reused and reassembled to create a new combination that was not originally anticipated. In many ways, algorithm abstraction is like data abstraction. Data abstraction is the process of moving common elements to a base class so they can be used by many derived classes. Algorithm abstraction is the process of moving common operations or tasks into a single generic algorithm so they can be used to serve new purposes.

There are three main algorithm abstraction design patterns:

Algorithm abstraction is the process of taking a potentially complicated algorithm and reducing it to its components

Design Pattern	Description
Strategy	There are multiple variations of an algorithm from which the client can choose between
Template Method	Define common steps in an algorithm and allow the subclasses to decide what the steps mean
Decorator	Add tasks to an algorithm dynamically, so they can be applied in any order or combination

It is possible to implement each of these design patterns without using OO tools. Additionally, the entire purpose of these techniques is to make algorithms more adaptable and powerful. For these purposes, one could argue that these belong in Unit 0: Algorithm Design. However, inheritance and composition are two effective tools enabling us to achieve these purposes more elegantly and flexibly. For this reason, algorithm abstraction techniques are class relation design patterns.

Strategy

The strategy design pattern involves identifying collections of related algorithms, each of which honors the same contract. These algorithms are placed in classes in a single inheritance hierarchy. The client is then able to choose and select the variation that is needed for the application. There are several components to the strategy pattern:

1. **Abstract Strategy.** Also known as the algorithm interface or simply as the strategy class, the abstract strategy class is the abstract base class from which all concrete strategies are derived. The abstract strategy class defines the interface or contract for the algorithm to be abstracted. Usually, the abstract strategy class has no attributes and only a single method.
2. **Concrete Strategy.** There are several concrete strategy classes, each of which is derived from the abstract strategy base class. Each concrete strategy represents one variation of the algorithm being abstracted.
3. **Context.** Also known as the client, the context is the class that uses the algorithm defined in a concrete strategy class. This class contains the algorithm that is to be abstracted.

The context class contains one instance of an abstract strategy object. The abstract strategy class is an abstract class with several derived classes.

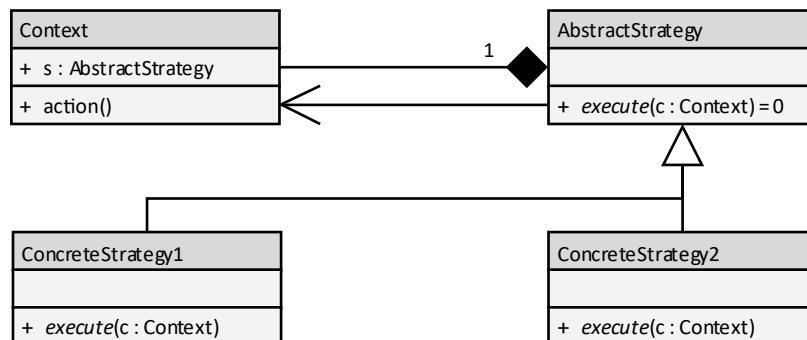


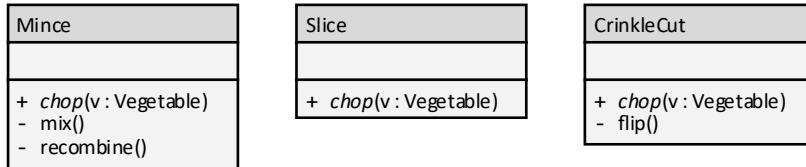
Figure 38.1:
Class diagram of the
strategy pattern

Notice that the **Context** class contains an instance of an **AbstractStrategy** object through composition. This is represented both with an **AbstractStrategy** member variable in the **Context** class and with the filled diamond. This member variable is used to access the `execute()` method belonging to the **AbstractStrategy** class. Finally, observe that **Context** has an `action()` method. This is the algorithm that is being abstracted.

The **AbstractStrategy** class is abstract because its `execute()` method is a pure virtual function (as indicated with the “= 0” clause on the `execute()` method). Because `execute()` accepts a **Context** object as a parameter, it is related to **Context** through association. The open arrow endcap indicates this relationship.

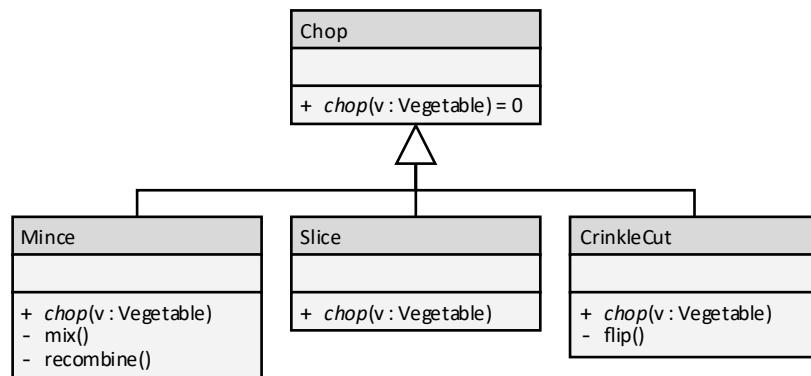
An **AbstractStrategy** class has several **ConcreteStrategy** derived classes (**ConcreteStrategy1** and **ConcreteStrategy2**). These derived classes implement the `execute()` method differently. The inheritance relation is represented with an empty, outlined triangle as the endcap.

To see how this works, imagine a cooking program that represents several ways to chop vegetables. These include mincing (chopping finely), slicing (a collection of parallel cuts to the food), crinkle-cutting (slicing that leaves a corrugated surface), and many others. Each of these algorithms is represented in its own concrete strategy class. Note that we give each derived class a verb name because the class only contains a single public method and methods are usually given verb names.



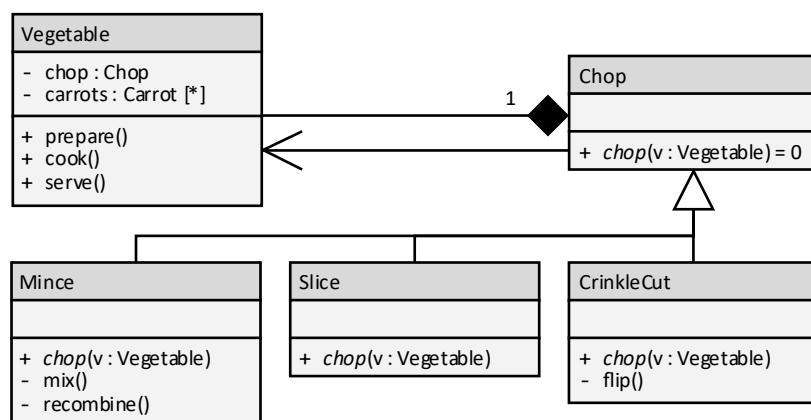
*Figure 38.2:
Class diagram of several
chopping varieties*

Each derived class is a polymorphic sibling to a single base class: **Chop**. **Chop** is our abstract strategy class and serves no purpose other than connecting the three derived classes.



*Figure 38.3:
Class diagram of Chop
and its derived classes*

The final part to the strategy design pattern is to define the context class **Vegetable**. This class will contain an instance of **Chop** through composition or through association. Composition is used when the abstract algorithm needs to be utilized several times during the lifetime of a **Vegetable** object. Association is used when the **Chop** object is only utilized in a single method.



*Figure 38.4:
Class diagram of the
strategy pattern*

Notice that the **Vegetable** class does many things—it prepares the vegetables, cooks them, and serves them. One aspect of the **prepare()** method is abstracted, the chopping component. We may wish to do the same for the wash and serve aspects as well.

The **Vegetable** class must instantiate the appropriate **Chop** variation. This happens before the **chop()** method can be invoked. Though there are many ways to instantiate this class, we will initially use the **new** keyword. This is accomplished when the context class is instantiated.

Pseudocode

```
Vegetable.initialize()
... code removed for brevity ...

chop ← new Mince
... code removed for brevity ...
```

Figure 38.5:
Pseudocode of selecting a
chop variety

Notice that the context (the **Vegetable** class) needs to know about every type of concrete strategy (**Mince**, **Slice**, and **CrinkleCut**). This makes the coupling between the classes tighter than necessary.

Best Practice 38.1 Use a factory to instantiate a concrete strategy to reduce coupling

The tight coupling between the context and the strategy classes can be reduced through the use of a factory method (See Chapter 37 Strategy: Object Creation for details on the factory design pattern).

Pseudocode

```
static Chop.factory(chopType)
SWITCH chopType
CASE mince
    RETURN new Mince
CASE slice
    RETURN new Slice
CASE crinkle
    RETURN new CrinkleCut
```

Figure 38.6:
Pseudocode of the factory
pattern

The factory function will then be the only function that is aware of the various ways of chopping vegetables.

Best Practice 38.2 Use the strategy design pattern when the algorithm variation is selected at runtime

The strategy design pattern combined with a factory design pattern makes it possible to select the algorithm variation at runtime. This is a unique feature of algorithm abstraction; most algorithms are compiled into the codebase. The specific algorithm can be selected by the user or even randomly, introducing a whole new level of malleability.

Pseudocode

```
Vegetable.initialize()
... code removed for brevity ...

chop ← Chop.factory(RANDOM)
... code removed for brevity ...
```

Figure 38.7:
Pseudocode of a
random chopper

Template Method

The template method is a design pattern where the components of an algorithm are separated into steps, each of which is defined in a derived class. These steps are then executed from a single method. There are several components to the template method:

1. **Template Method.** This is the algorithm which is to be abstracted. This algorithm is subdivided into several steps.
2. **Step.** The steps are the components of the template algorithm. Each step can be defined in the base class or in a derived class.
3. **Template.** The template, also known as the abstract template, is the class which contains the template method. Since most template classes have at least one pure virtual step method, **Template** is usually an abstract class. There are usually several concrete templates which implement the step methods.
4. **Client.** The client is the application that uses the template method. The client also chooses which variation of the algorithm is to be used.

The template class has at least one derived class, usually called a concrete template. The client can reference the template in many ways, depending on the needs of the application.

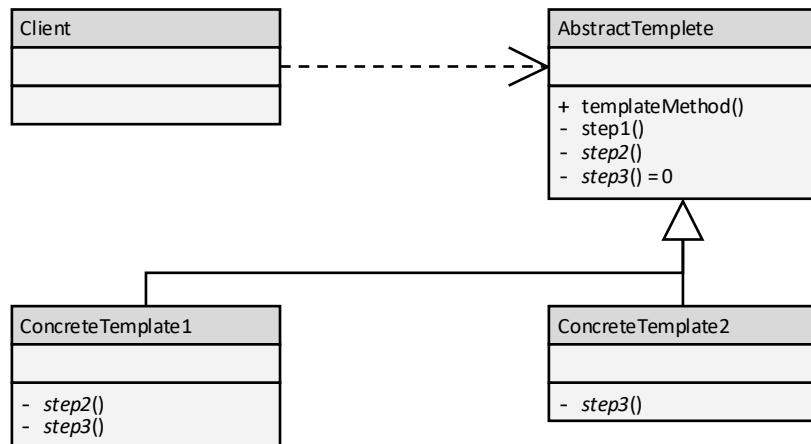


Figure 38.8:
Class diagram of the
template method

Client can invoke **AbstractTemplate.templateMethod()** in a variety of ways. Client may contain an instance of **AbstractTemplate**, or it could be that **AbstractTemplate** is a static class, so instantiating an **AbstractTemplate** object is unnecessary. The exact relationship of Client and **AbstractTemplate** is up to the needs of the client. For this reason, a simple dependency open arrow is used in the class diagram.

Observe how **AbstractTemplate** has only one public method—the **templateMethod()** itself. This is the algorithm to be abstracted. This method is not virtual because it contains the unchanging aspect of the algorithm. It does, however, call the various step methods. As seen in Figure 38.8, each step can be a concrete function (as is the case with **step1()**), a virtual function (see **step2()**), or a pure virtual function (**step3()**). Only virtual steps can be implemented in the concrete templates. Pure virtual steps *must* be implemented in a concrete template.

Back to our cooking example, you can bake a cake by first mixing the dry ingredients, then mixing the wet ingredients, then combining all the ingredients, and finally putting the cake in the oven. You can bake bread using the same general algorithm (dry, wet, combine, then bake). To make this algorithm abstract, the first step is to define the template method.

Pseudocode

```
Bake.bake()
    mixDryIngredients()

    mixWetIngredients()

    combineIngredients()
    WHILE not mixed
        mix()

    placeInOven()
    WHILE not finished()
        wait()
```

This algorithm has several components which are common to all baking scenarios: the process of mixing the dry ingredients, the process of mixing the wet ingredients, combining the two types of ingredients together, mixing them all together until everything is consistent, placing the mixture in the oven, and waiting for it to finish. There are important variations, however. The combination of dry ingredients varies according to what is being made. The same is true with the wet ingredients. Finally, the way one determines whether things are finished is quite different from one baked item to another. Based on this, we will have four concrete

Figure 38.9:
Pseudocode of a
template method

methods (`combineIngredients()`, `mix()`, `placeInOven()`, and `wait()`), and we will have three virtual methods (`mixDryIngredients()`, `mixWetIngredients()`, and `finished()`).

Three steps are abstract, requiring a derived class to define them

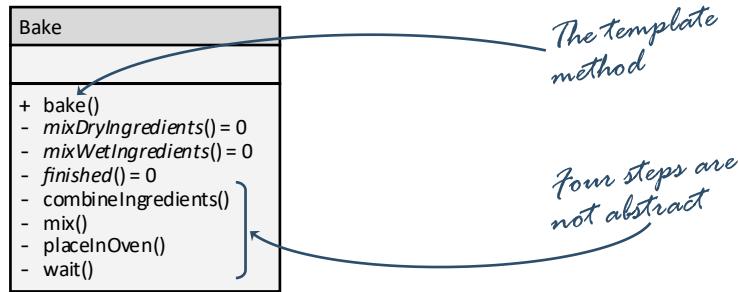


Figure 38.10:
Class diagram of an
abstract template

Our cooking program will bake three things: cakes, breads, and protein bars. We will create three derived classes from our `Bake` base class. Observe how the concrete classes only describe what is unique about them, allowing the `Bake` class to define the common operations as well as the overriding `bake()` algorithm.

Parts of the algorithm common to all are here

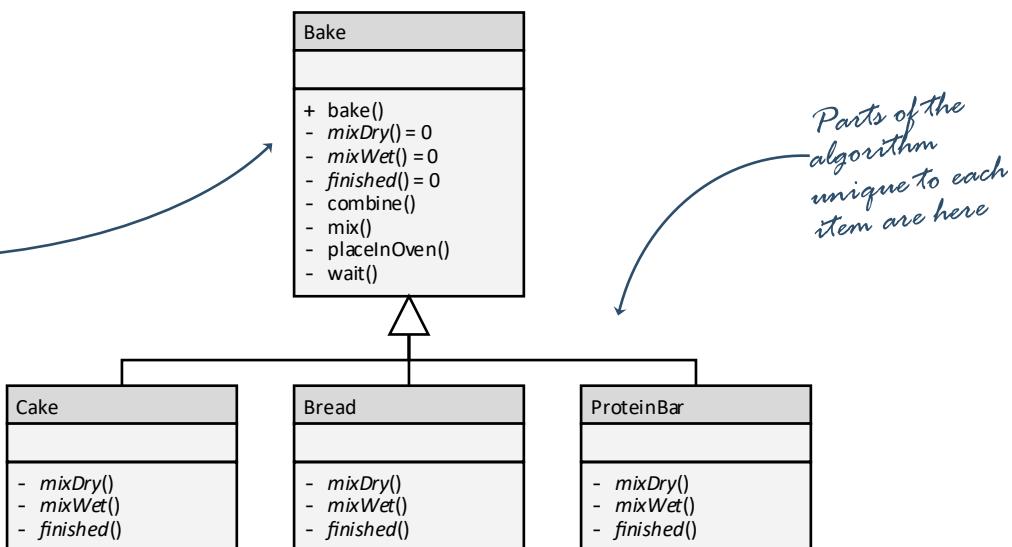


Figure 38.11:
Class diagram of abstract
and concrete templates

When the client chooses to make protein bars, it is only necessary to instantiate a `ProteinBar` class. Baking a protein bar is possible even though none of the `ProteinBar` methods are public to the client. The only public method is the template method `bake()`, which is defined in the `Bake` base class. This `Bake.bake()` method then calls the private methods in the `ProteinBar` class, which know the specifics of making a protein bar.

Figure 38.12:
Pseudocode of invoking a template method

```
Pseudocode
makeLunch()
getDrink()

IF proteinBars.count() = 0
    proteinBar = new ProteinBar
    proteinBar.bake()
    proteinBars.push_back(proteinBar)
    proteinBar[0].eat()
```

Observe how the client never uses the base class; the client simply instantiates the desired instance of the derived class and calls the template method. The client does not have to be aware that an algorithm abstraction design pattern is used, making the abstraction level of this design complete.

Best Practice 38.3 Use the template method pattern when an algorithm has several changeable steps

When an algorithm has just one component that varies, then the strategy design pattern is a better choice. However, when there are several steps which can be defined at various levels in an inheritance hierarchy, then the template method is a better fit.

Best Practice 38.4 Avoid redundancy by percolating common tasks higher in the inheritance hierarchy

Template methods often have deep inheritance hierarchies, allowing common steps to be used by many concrete classes. Back to our baking example, there are several flavors of protein bars. Of these, the nut variety have the same wet ingredients but have different dry ingredients.

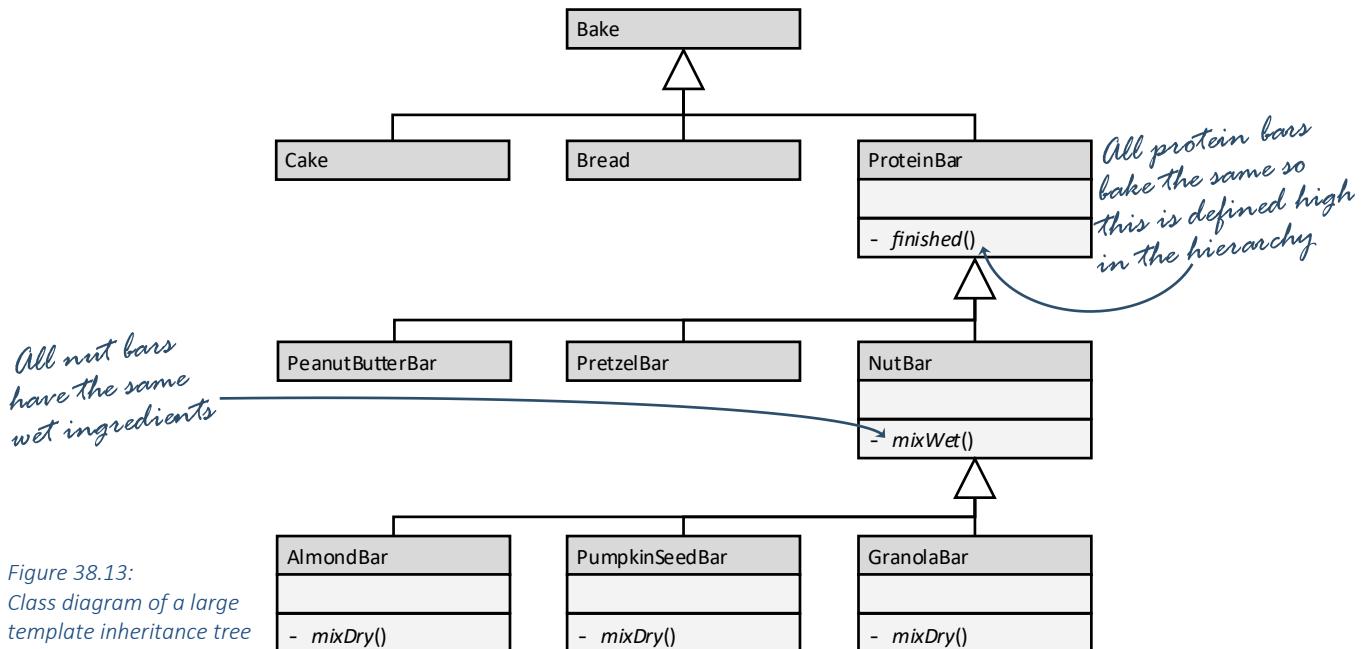


Figure 38.13:
Class diagram of a large template inheritance tree

Decorator

The decorator design pattern is a technique for adding a sequence of tasks to an algorithm so it can easily be extended. The client assembles the sequence, and when the algorithm is executed, each task in the sequence is executed in order.

The traditional way to implement the decorator pattern is recursively. There is a base task, which performs an action and ends when the action is completed. In addition to the base task, there are a collection of recursive tasks. These also perform an action but, when finished, invoke the next task in the sequence. The client may chain as many recursive tasks as needed, each one extending the original base task. There are several parts to this pattern:

1. **Task.** The algorithm that the client wishes to execute. This task is usually represented as a single method in a class.
2. **Abstract Component.** The class that contains the task method. In the abstract component, the task method is a pure virtual function, making the component class abstract.
3. **Concrete Component.** A class that derives from abstract component and contains a concrete implementation of task. This concrete component contains the base task, which is a task that performs an operation and then ends.
4. **Decorator.** The decorator is a polymorphic sibling of the concrete component, both of which derive from the abstract component. As with the concrete component, the decorator contains a task method which performs an operation. In addition, to that, the decorator contains a reference to a component. Once the decorator's task has completed, it then invokes the task method of the component to which it references. This is the recursive aspect of the decorator pattern.
5. **Concrete Decorator:** Usually the decorator design pattern contains several tasks which can be added to the parent task. Each of these tasks is implemented in its own concrete decorator, which serves as a single task in the chain to be executed by the client. Thus, the set of concrete decorators that derive off the **Decorator** class constitutes the menu of tasks that can be applied to the base task.

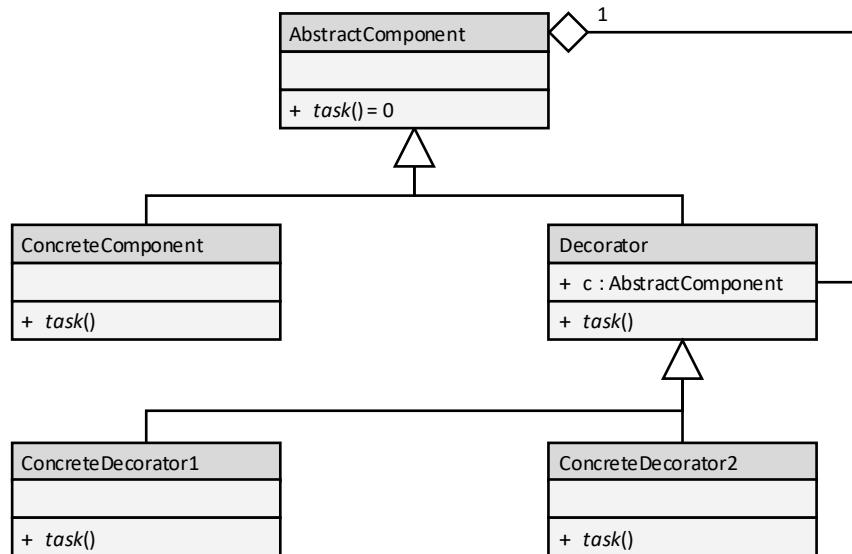


Figure 38.14:
Class diagram of the
decorator pattern

The important thing to notice about the decorator pattern is that the **Decorator** class contains a reference to its parent. This parent references functions much like a node pointer in a linked list, referencing the next node in the collection. If **ConcreteDecorator1** and **ConcreteDecorator2** were to be applied to the main task, then the resulting object relationship would be the following:

Figure 38.15:
Object diagram of
a decorator chain



Back to our cooking example, we have an algorithm which makes marinara sauce (also known as “red sauce” or occasionally “spaghetti sauce”) for Italian dishes. Though all these dishes have the same basic components (tomatoes, vegetables, and spices) there are many variations. Pizza sauce is thinner, so it spreads on a pizza crust more easily. Bolognese adds meats. Burro adds Parmesan cheese and butter. Arrabbiata is spicy and cooks for a very long time. To accommodate these variations, the decorator pattern will be applied to our marinara algorithm. The first step is to create a class which makes the basic “sugo semplice” (meaning “easy sauce” in Italian). This class contains a **prepare()** method which combines the vegetables and spices.

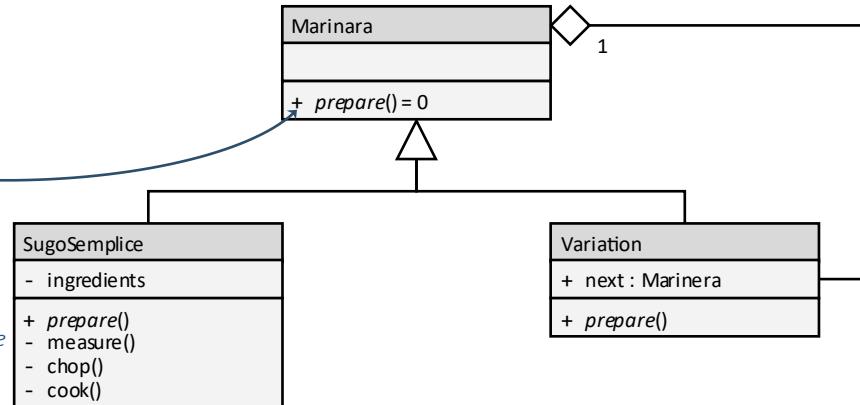
Figure 38.16:
Class diagram of a
concrete component



SugoSemplice serves as the concrete component in the design pattern because it does not refer to any other task. In other words, **SugoSemplice.prepare()** does not invoke methods from another class.

Next, we know that there are going to be many variations of our marinara sauce. To accomplish this, we will create a base class called **Marinara** and a derived class called **Variation**. **Variation** is the decorator class. By itself, it does nothing other than call the next component in the chain.

Figure 38.17:
Class diagram of a concrete
component, abstract
component, and decorator



Notice that **Marinara** is an abstract class. This serves as the abstract component in the decorator design pattern. **Variation** contains a reference to **Marinara** so it can call **Marinara.prepare()**. To complete the decorator pattern, we need concrete decorators which derive from the **Variation** class.

Initially, we will define three alterations to our sauce: **Pizza**, **Bolognese**, and **Burro**. Each alteration is derived from **Variation**.

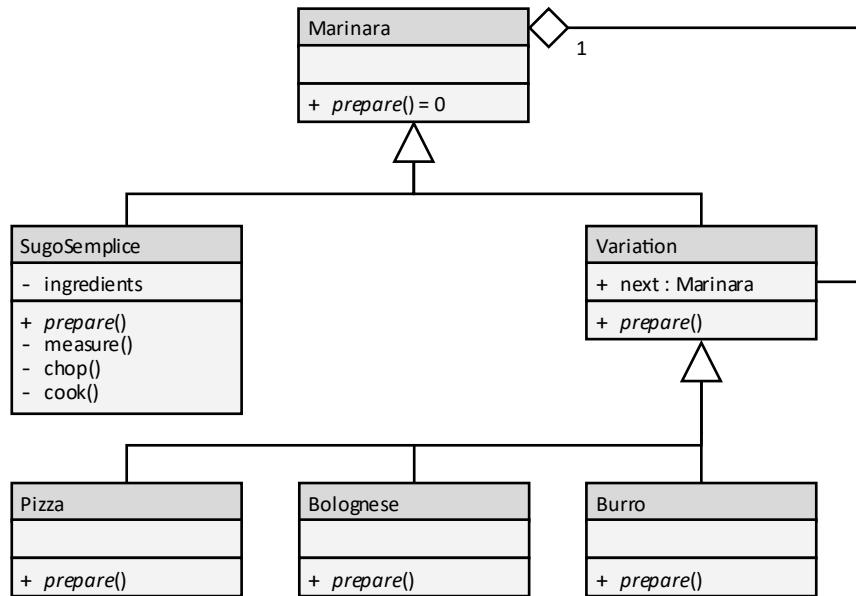


Figure 38.18:
Class diagram of the
decorator pattern

Each class that derives from **Variation** is a concrete decorator. These classes all implement **prepare()**. While the details of each implementation of **prepare()** vary according to how the variation is meant to alter the sauce, they all have the same component. They all call the **prepare()** method from the decorator base class. This ensures that the next task in the chain will be executed.

Pseudocode
<pre> Bolognese.prepare() Variation.prepare() measureMeat() cookMeat() addToSauce() </pre>

Figure 38.19:
Pseudocode of a concrete
component calling its
parent's task method

The decorator base class serves two purposes: it is a base class for all the decorator concrete classes so they have a shared interface, and it calls the next decorator in the chain. Recall that the decorator (**Variation** in this case) maintains a reference to the next element in the chain. This reference (called **next**) was made when the object was instantiated.

Pseudocode
<pre> Variation.prepare() next.prepare() </pre>

Figure 38.20:
Pseudocode of a
decorator's task method

Notice that only the **Variation** decorator class is aware of the chain of decorators to be executed. The concrete decorators (such as **Bolognese**) need to remember to call the decorator's **prepare()** method to ensure the chain is not broken.

The final step in the decorator design pattern is how the client specifies the variation of marinara sauce he or she would like to make. First, the concrete component (`SugoSemplice` in our marinara example) does not take a parameter. This is because we always start with a concrete component. If we wish to create a simple sauce, then we simply instantiate a `SugoSemplice` object.

Figure 38.21:
C++ code of creating a
simple object

```
C++  
// Easy sauce for those who don't like things fancy  
Marinara * pEasy = new SugoSemplice();
```

Next, we wish to add meat to our sauce, requiring the addition of the `Bolognese` variation. Note that the constructor of all decorators requires a decorator as a parameter. This is how the `next` member variable is initialized so the chain can be created.

Figure 38.22:
C++ code of creating an
object with one decorator

```
C++  
// Standard meat sauce  
Marinara * pMeat = new Bolognese(new SugoSemplice());
```

When finished, we will have created a two-task chain. It is easy to add new decorators to our chain. The only thing we need to remember is to pass the previous item onto the next one in the sequence.

Figure 38.23:
C++ code of two
decorators

```
C++  
// Spicy Cheese sauce  
Marinara * pBase      = new SugoSemplice();  
Marinara * pCheese    = new Burro(pBase);  
Marinara * pSpicyCheese = new Arrabiata(pCheese);
```

The client can also combine variations in ways that are difficult for the designer to anticipate. In this case, the client is ordering a triple meat, cheese, double spicy pizza sauce. While it may be complex to order, the decorator handles the variation with ease.

Figure 38.24:
C++ code of seven
decorators

```
C++  
// Triple meat, cheese, double spicy pizza sauce  
Marinara * pSauce = new SugoSemplice();  
pSauce = new Bolognese(new Bolognese(new Bolognese(pSauce)));  
pSauce = new Burro(pSauce);  
pSauce = new Arrabiata(new Arrabiata(pSauce));  
pSauce = new Pizza(pSauce);
```

This task chain is quite a bit longer than our previous one.

Best Practice 38.4 Verify your decorator implementation early and often

When it is possible to create long and complex task chains such as these, it is a good idea to write unit tests to verify that they work as they should. Task chain bugs can be difficult to find and even more difficult to fix. They are much easier to find and fix early in the development process.

Alternative Implementation

While the standard way to implement the decorator design pattern is with a chain of objects, there are other ways which many see as more straightforward. One popular variation is to create a list or array of objects which will be executed in sequence.

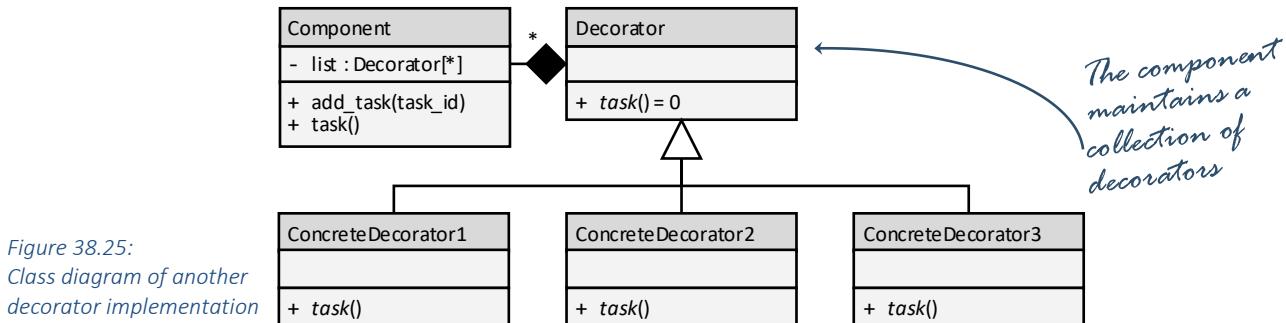


Figure 38.25:
Class diagram of another
decorator implementation

To illustrate how this works, we will return to our marinara example. Since all sauces start with the sugo semplice base, that functionality will be moved to the component class. The decorator will be vastly simplified, providing only an interface for the variations.

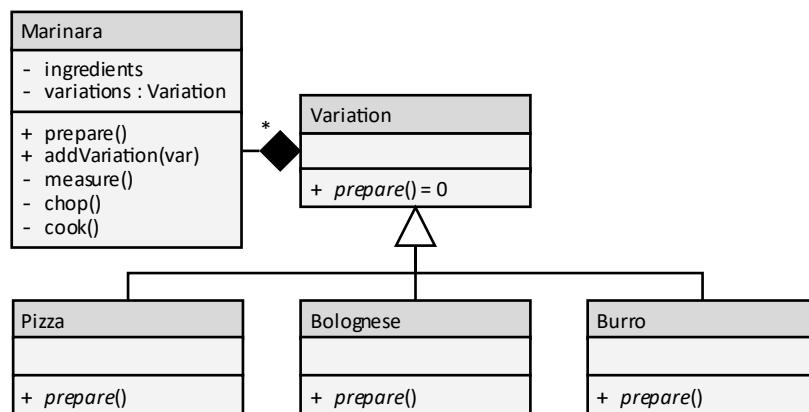


Figure 38.26:
Class diagram of another
decorator implementation

The final step is to add an `Marinara.addVariation()` method allowing the client to specify what is to be added to the client. To simplify this process, `addVariation()` is a factory method. This shields the client from needing to have direct access to the `Variation` class or having to know the various concrete decorator names.

Pseudocode
<pre> Marinara.addVariation(var) SWITCH var CASE pizza variations.push_back(new Pizza()) CASE bolognese variations.push_back(new Bolognese()) CASE burro variations.push_back(new Burro()) </pre>

Examples

Example 38.1: Strategy

This example will demonstrate the strategy design pattern.

Problem

Create a design to match the following problem definition:

A bank account may have one of a variety of fees associated with it. The set of fees include the following: a fixed annual fee, a percentage of transactions, and a fixed fee per transaction.

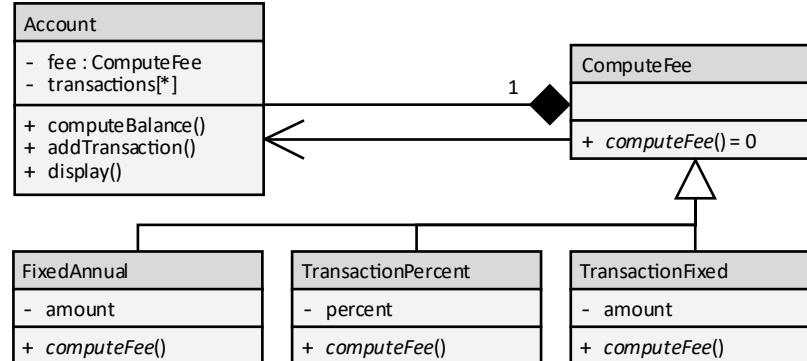
Solution

Observe how the only one aspect of the balance computation algorithm varies: the fee computation component. With one variation, the strategy design pattern is a good fit.

The context is the class that contains the bank account itself. This class will contain the list of transactions and provide a variety of useful services in addition to the balance computation algorithm.

The abstract strategy is the fee component containing a `computeFee()` method. Since abstract strategies are usually named after the method they contain, this class will be called `ComputeFee`.

The final aspect of the strategy design pattern is to define the various concrete strategies. These will be called `FixedAnnualFee`, `TransactionPercent`, and `TransactionFixed`.



Example 38.2: Template

This example will demonstrate the template method design pattern.

Problem

Create a design to match the following problem definition:

A bank account balance is updated in three steps: apply any transaction added since the last balance update event, compute interest, and apply any market corrections. The ways that transactions are added for cash accounts (credit, savings, checking) is quite different than how they are added for investment accounts (with stocks and shares). Interest computation also varies depending on account type. Finally, only some account types have market correction components (such as stocks, precious metals, and foreign currency).

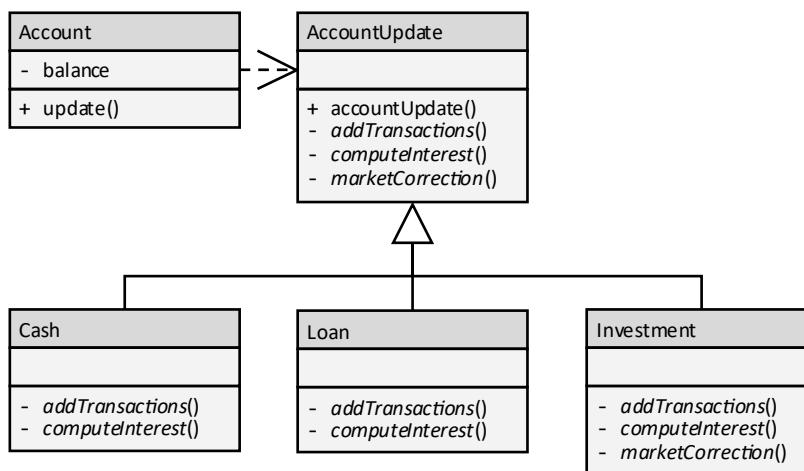
Solution

The balance update feature occurs in three distinct steps, each of which have various implementations. This is a good fit for the template method.

The client class will contain an account balance and an `update()` method. The `update()` method will call the `accountUpdate()` method in the `AccountUpdate` class.

The abstract template is the `AccountUpdate()` method which has one public method (`accountUpdate()`) and three virtual methods: `addTransactions()`, `computeInterest()`, and `marketCorrection()`.

Each account type will be a derived class from `AccountUpdate`. This includes `Cash` accounts (for savings and checking accounts), `Loan` (for credit cards, car loans, and mortgages), and `Investment` (for stocks, precious metals, and foreign currency). Observe that each account type implements `addTransactions()` and `computeInterest()`. However, note that only `Investment` implements `marketCorrection()`.



Example 38.3: Decorator

This example will demonstrate the decorator design pattern.

Problem

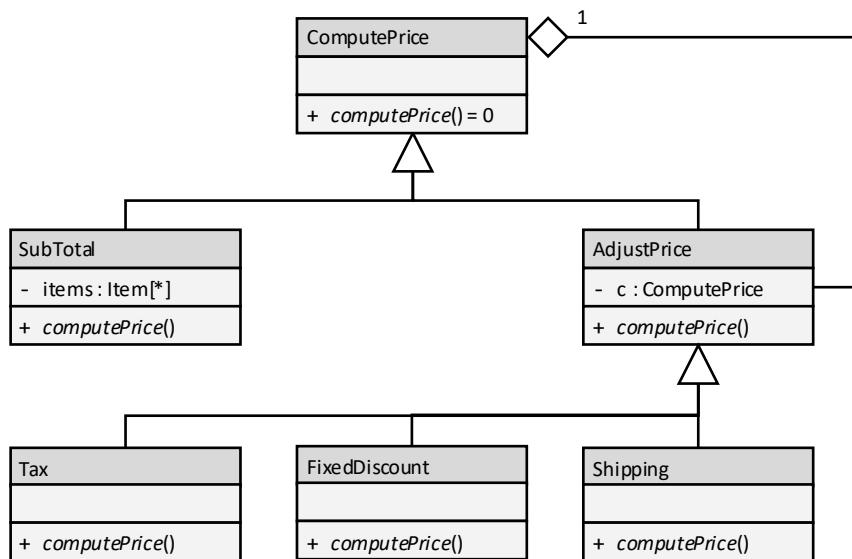
Create a design to match the following problem definition:

A store has a Point of Sale (PoS) system which computes the prices of a given purchase. This purchase includes the individual prices of the items, as well as a collection of special considerations. These considerations may or may not include tax (food items are not taxed), a fixed discount due to the application of one or more coupons, shipping fees, and/or any other special handling fees. These adjustments can occur in any combination.

Solution

The compute price algorithm consists of a fixed procedure (summing up the individual prices of the items) as well as a collection of procedures which may be applied. This makes the algorithm a good candidate for the decorator design pattern.

The task is the `computePrice()` method which determines the price or augments the price for the customer. This is the algorithm to be abstracted. The concrete component is the `SubTotal` class in the PoS system. This class contains the collection of items to be purchased and returns the total. When `SubTotal.computePrice()` is executed, the summation of all the item prices is returned. The abstract component is the `ComputePrice` class which is the base class for all classes which compute a price for the system. This class contains the pure virtual function `computePrice()`. The decorator is the `AdjustPrice` class, serving as the base class for all concrete decorators which will adjust the final price in some way. This class will contain the link to the next decorator in the chain. Finally, there are several concrete decorators. These serve to apply several adjustments to the client's bill. For now, we will include three: `Tax`, `FixedDiscount`, and `Shipping`.



Exercises

Exercise 38.1: Terms and Definitions

From memory, name the three algorithm abstraction design patterns. For each, describe its essential characteristic.

Design Pattern	Characteristic

Exercise 38.2: Components

The three algorithm abstraction design patterns consist of several component, each of which is represented as a class or as a method within a class. For each component, identify the corresponding design pattern and briefly describe the role it fulfills.

Component	Design Pattern	Role
Client		
Template Method		
Template		
Abstract Strategy		
Abstract Component		
Context		
Concrete Strategy		
Task		
Step		
Decorator		
Concrete Component		

Exercise 38.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
The client must select the concrete strategy at compile time.	
The template method can be thought of as a type of linked list.	
The decorator can be implemented as a list or array of tasks.	
A factory is a useful way to create a concrete strategy object.	
Use the decorator when the client has a fixed set of steps, each of which can have several variations.	

Exercise 38.4: Class Diagram

From memory, draw the class diagram associated with each of the following design patterns.

Design Pattern	Class Diagram
Strategy	
Template Method	
Decorator—standard implementation	
Decorator—alternative implementation	

Problems

Problem 38.1: Restaurant Bill

Identify an appropriate pattern from the following scenario:

A bill for a restaurant must factor in tax, extra charge for parties over six, tax, birthday discount, etc.

Draw a class diagram representing a design that will meet the needs of this scenario.

Problem 38.2: Video Game Thrust

Identify an appropriate pattern from the following scenario:

A spaceship in a video game can have one of a variety of thrusting models: inertial, non-inertial, and relative.

Draw a class diagram representing a design that will meet the needs of this scenario.

Problem 38.3: Store Bill

Identify an appropriate pattern from the following scenario:

The bill at a store is computed the same way each time: sum the items, apply tax, and print. However, the way items are summed are quite different if you are talking about labor or items for purchase. Also, there are five different tax algorithms depending on the type of service performed.

Draw a class diagram representing a design that will meet the needs of this scenario.

Problem 38.4: Video Game Position

Identify an appropriate pattern from the following scenario:

The position of an element in 3D video game is computed by several actions: inertia, gravity, attraction, thrust, and drag, to name a few. Not all elements honor all these actions.

Draw a class diagram representing a design that will meet the needs of this scenario.

Problem 38.5: To-Do

Identify an appropriate pattern from the following scenario:

A to-do application stores a collection of lists which are presented to the user according to his or her preferences. These lists can be displayed a variety of ways: filtered by name, filtered by difficulty, sorted by name, sorted by difficulty, or any combination thereof.

Draw a class diagram representing a design that will meet the needs of this scenario.

Problem 38.6: Financial Report

Identify an appropriate pattern from the following scenario:

A financial program displays a report of the state of the user's account. This involves the same steps: collect data, apply filter, compute summary, and display. Note that there are many variations of how to collect the data (which accounts?), to apply the filter (which types of records are included?), compute the summary (one total or by month?) and display the result (histogram, table, bar chart?).

Draw a class diagram representing a design that will meet the needs of this scenario.

Problem 38.7: Bar Chart

Identify an appropriate pattern from the following scenario:

A financial program has a bar-chart feature. This feature displays the provided data, but also has a collection of optional items to overlay: the title, the subtotals, account labels, and such.

Draw a class diagram representing a design that will meet the needs of this scenario.

Problem 38.8: Alternative Implementation

Use the alternative implementation of the decorator for the following scenario:

A store has a Point of Sale (PoS) system which computes the prices of a given purchase. This purchase includes the individual prices of the items, as well as a collection of special considerations. These considerations may or may not include tax (food items are not taxed), a fixed discount due to the application of one or more coupons, shipping fees, and/or any other special handling fees. These adjustments can occur in any combination.

Draw a class diagram and write pseudocode for two methods: `addAdjustment()` and `computePrice()`.

Challenges

Challenge 38.1: Decorator Implementations

In the programming language of your choice, implement both the linked list and the collection versions of the decorator design pattern. What are the pros and cons of each implementation?

Challenge 38.2: Simulator

An artillery simulator computes where a shell will land after being fired. This is done by computing the position of the shell every 1/10th of a second from the time it is fired. The simulator can model various forces on the shell: effects of aerodynamic drag on the shell, the force of wind, various gravity models, etc. Demonstrate the template method on this simulator.

Chapter 39 State

Many complex software problems can be simplified by turning them into state machines.

The chief technology officer (CTO) at a large bank has just hired you to write the software for the next generation of automated teller machines (ATMs). Security is an important requirement of this project; the CTO needs assurances that the software will not malfunction and mistakenly discharge all its cash due to a logic error. As you gather requirements, it becomes apparent that the system must maintain dozens of different states, each of which has a collection of allowable actions. How can anyone keep track of all of this, let alone design a system with the level of trust the CTO needs? Fortunately, there is a design strategy that facilitates representing, understanding, and implementing algorithms that maintain their state.

A state is a discrete configuration of a system which describes its behavior

A state is a discrete configuration of a system. The discrete part of the definition refers to the fact that states cannot be subdivided or blended. For example, a bank account can be open or closed; it

cannot be somewhere in between. Thus, the status of a bank account is discrete. On the other hand, the amount of money in an account can be subdivided. One could describe the amount variable as continuous. States are discrete, indivisible, and distinct. The configuration part of the state definition refers to the fact that a state represents a collection of settings that map to a status of the system that is meaningful in the context of the program. This part of the definition is difficult to identify unambiguously but can be recognized with practice. A car's transmission has five gears (reverse, neutral, park, drive, and first), each of which is a state. A transaction at your bank is in one of four states: cleared, voided, reconciled, or uncleared. A taxi is under hire, off duty, or available, each of which is a state. States represent a meaningful configuration of the system which describes its behavior.

Stateful systems usually have a set of transitions which govern how the system can move from one state to another. Each transition has a source state, a destination state, and usually a function governing whether the transition takes place.

The collection of all possible states and their transitions is called the state space. Though it is possible for a state space to be very large or even infinite, most systems utilizing state strategies have fewer than a few dozen well-defined states. It is not uncommon for a designer to initially identify a few states, only to discover that the state space is much larger once the problem is better understood.

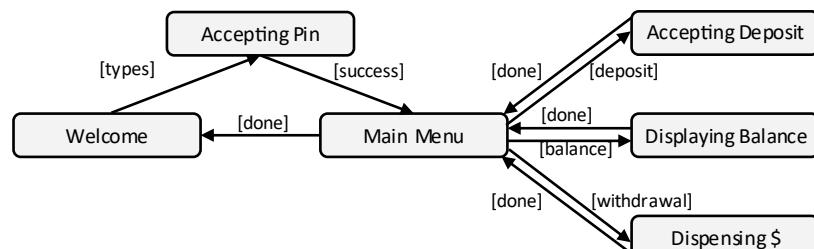


Figure 39.1:
State diagram of a simple
ATM system

This chapter will describe how to represent state with a state diagram, create designs that make effective use of states, and how to represent state algorithms in code.

State Diagram

The first software design tool ever utilized by programmers was the state diagram. Introduced by Claude Shannon and Marren Weiver in 1949, state diagrams are also known as state machines, state transition diagrams, state-chart diagrams, and state charts. They are used to represent systems that have a finite number of states and well-defined events that facilitate movement between states. A state diagram is a directed graph consisting of nodes (states), edges (state transitions), and groupings (collections of related states).

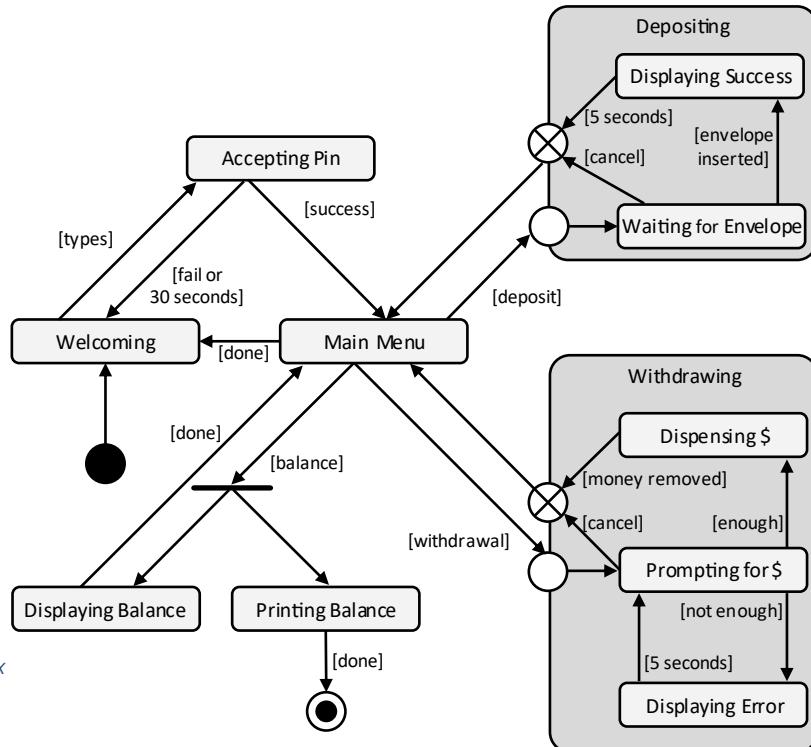


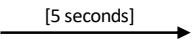
Figure 39.2:
State diagram of a complex
ATM system

State transition diagrams have the following properties:

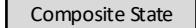
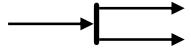
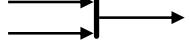
Property	Description
Use	Represent, design, and analyze the states of a system
Viewpoint	Process: State diagrams represent one aspect of algorithms
Strength	Easy to understand, easy to create
Weakness	Difficult to scale, only represents part of an algorithm

State Diagram Elements

A state diagram consists of two main elements and a collection of supporting elements. The main elements are states and transitions.

Symbol	Name and Description
	State: Represents one configuration or state of the system. Each state in a state diagram should have a label and represent a unique condition of the system.
	Transition: Describes how the system can move from one state to another. Each transition should have a condition represented in a label next to the arrow.

These two elements are enough for most state problems. State diagrams may also have a collection of other elements that help represent special design concerns. These include the initial state, the final state, forks, joins, composite states, entry points, and exit points.

Symbol	Name and Description
	Initial State: The initial state is a pseudo-state (not a real state), indicating where the program begins. The system immediately transitions to a regular state from here.
	Final State: A special state representing the last state the program is in. An arrow can lead into the final state but cannot originate from a final state.
	Composite State: A collection of related states, serving only to clarify complex state diagrams. They do not represent the actual program's state.
	Entry Point: A pseudo-state representing the entry point to a composite state. An entry point is the initial state of a composite state.
	Exit Point: A pseudo-state representing the exit point from a composite state. The exit point represents the final state of a composite state.
	Fork: A special transition indicating that two or more states will be simultaneously occupied next. Forks are only used in multi-process state machines.
	Join: A special transition indicating that several states in a concurrent state machine transition to a single state, moving from several processes to one.

State

A state is a discrete configuration of the system. In a state diagram, a state is represented with a rounded rectangle and a label. A state is like a vertex in a graph and the collection of all the states in the system is called the state space.

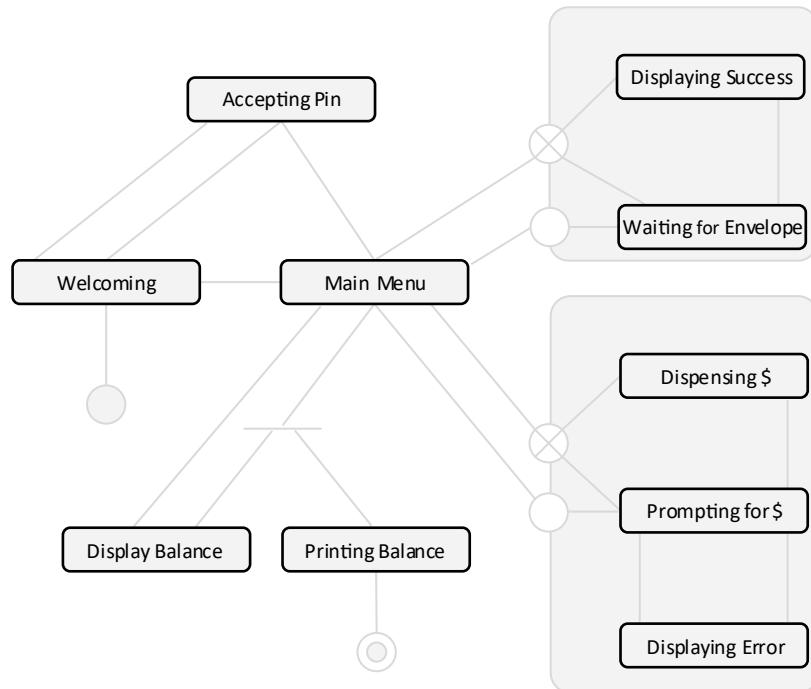


Figure 39.3:
State diagram illustrating
the states of an ATM

Each state in a state diagram should be given a label. This label serves two purposes. First, it allows the designer to assign meaning to the state. Poorly understood problems are characterized by missing or overly generic labels. Second, the label serves to uniquely identify each state. This makes it easier to refer to a given state when discussing the design with others, when ascribing properties to the state, or when implementing the state in the code.

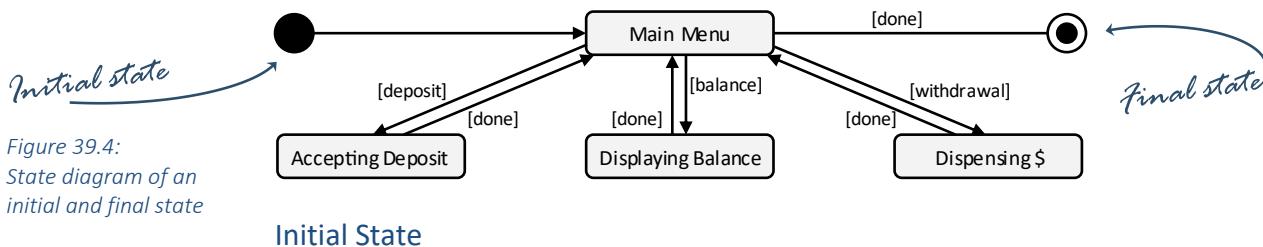
Best Practice 39.1 State labels are adjectives, nouns, or progressive verbs

Because a state can represent a configuration of the system, it can have an adjective label. Here, the label should refer to the name of the configuration. For example, a check may be reconciled. Here, reconciled is an adjective referring to the status of a check so “reconciled” is an appropriate label for a check state. A state can also represent a noun. A car can be in first gear, so “first” is an appropriate label for a transmission state. Finally, a state can represent a status of the system or what the system is doing. These are best captured with a progressive verb label. Thus, labels like “waiting” or “printing” or “accepting input” are common.

Best Practice 39.2 Do not label a state with an action verb

Be cautious when a state is given an action verb name. Action verbs usually refer to a process and are more appropriate for function or method names. When designers start using action verb labels, their state diagrams begin to resemble flowcharts or structure charts.

There are two special states in some state diagrams: the initial state and the final state.



Initial State

The initial state is the state of the application before execution begins. The initial state is not a true state in that the system immediately transitions to the next state. Therefore, the transition from the initial state usually has no label. The initial state is drawn with a filled black circle.

Figure 39.5: State diagram of an initial state

Though many state diagrams have initial state symbols, it is not required in the state diagram syntax and is not needed in all scenarios.

Best Practice 39.3 Use an initial state symbol only when the system starts in a given state

When an initial state has a label, it is usually because it indicates a connection with another state diagram, in other words, state diagram #1 may flow into state diagram #2. Here, state diagram #2's initial state will have a label so it can be connected to that of #1. Similarly, a state diagram may become large and complex. In situations like these, connecting State A with State B may involve a very long state transition arrow. This can be avoided by adding a labeled initial and final state symbol. In many ways, this works like a connector in a flowchart.

Figure 39.6: State diagram of connected initial and final states



Final State

The final state is the state of the application after execution ends. It is drawn with a filled black circle within a circle outline.

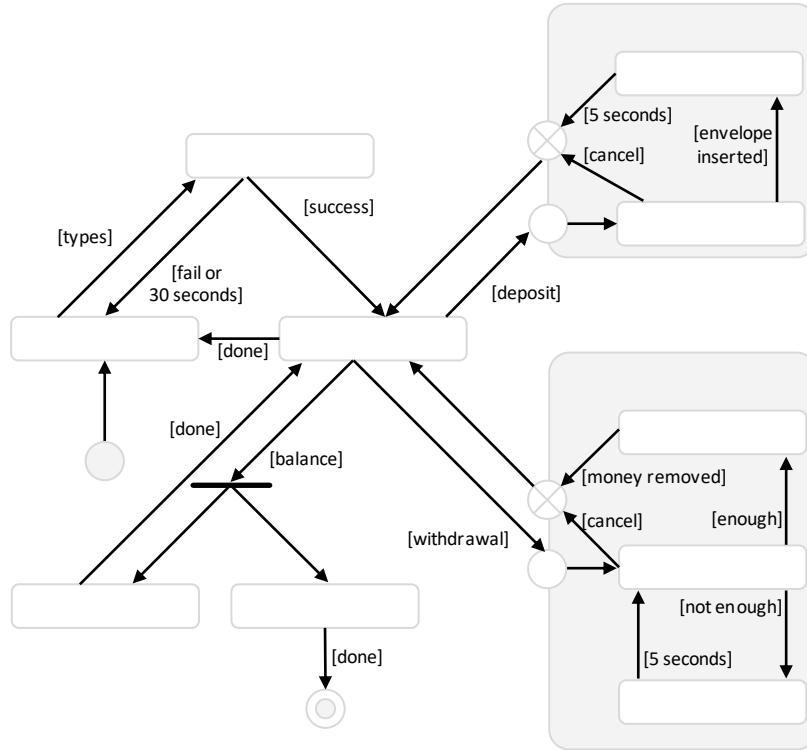
Figure 39.7: State diagram of a final state

As with the initial state, the final state is not a true state of the system because the running system cannot occupy it. In most cases, the transition into the final state has an event associated with it. This is different than that of the initial state.

The final state is typically not labeled. The exception is when the final state of one state diagram corresponds to the initial state of another. It is important when this occurs to verify that the label to the final state matches the label for the corresponding initial state.

Transition

A transition is the movement of the system from one state to another. This process is called a state transition. In a state diagram, a transition is represented with an arrow and a label. A transition is like an edge on a directed graph, the collection of which constitutes every way the system can move between states.



*Figure 39.8:
State diagram of
transitions for an ATM*

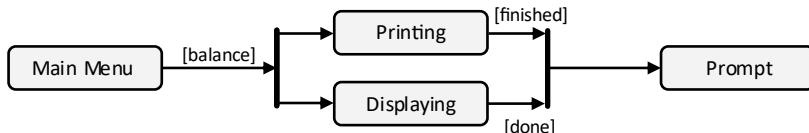
A transition consists of two parts: the transition arrow and the transition condition. The transition arrow connects two states. There is always exactly one source state and exactly one destination state. If it is possible for the system to move to several different states from a given initial state, then several transition arrows are required.

Transitions also contain a transition condition. This represents the event that indicates the transition will happen. There are three main types of transition conditions:

- **Function:** A transition condition can be the execution of a function which results in a state transition. This function is called a transfer function and they are represented with the function name in square brackets: `[login()]`.
 - **Event.** A transition condition can be represented as an event, the realization of which results in a state transition. These events can be the result of external stimuli (usually input from outside the system or subsystem), or from the passage of time. Such events are called triggers and they are represented with the event in square brackets: `[click]`.
 - **Condition.** A transition condition can be represented as a Boolean expression. When the expression evaluates to true, then the transition occurs. These are called guard conditions and are represented with the Boolean expression in square brackets: `[amount_requested >= amount_required]`.

There are two special types of transitions: a fork and a join.

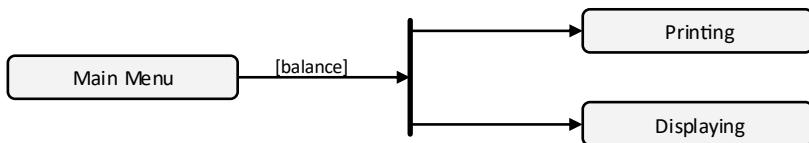
Figure 39.9:
State diagram of
a fork and a join



Fork

A fork is a transition between a single source state and several destination states. The important thing about a fork is that the destination states are occupied concurrently. In other words, this occurs when one process splits into two or more. A fork is drawn with a solid bar perpendicular to the dominant flow of the arrow.

Figure 39.10:
State diagram
of a fork

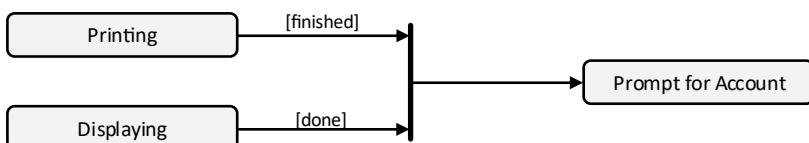


In figure 39.10, the transition condition of a fork is on the arrow leaving the **Main Menu** state. This is the condition that must be met for the fork to occur. There is no transition condition after the fork, meaning that control travels to both the **Printing** and the **Displaying** state unconditionally. Also notice that the ATM was in a single state before the balance option was selected – the **Main Menu** state. After the balance option was selected, then the ATM is in two states simultaneously: **Printing** and **Displaying**. This means that the ATM is both printing a receipt for the user and displaying the balance on the screen at the same time.

Join

A join is a transition between multiple source states and a single destination state. As with the fork, the requirement for the join is that the system occupies the source states concurrently. In other words, two or more processes combine into one. As with a fork, a join is drawn with a solid bar perpendicular to the dominant flow of the arrow.

Figure 39.11:
State diagram
of a join



In figure 39.11, the system is initially in both the **Printing** and the **Displaying** states simultaneously. When the printing process is finished (as is indicated by the **[finished]** transition condition), control moves to the join and pauses. When the balance displaying process is finished (as is indicated by the **[done]** transition condition, indicating that the user has selected the **[done]** option on the screen), control from this process also moves to the join and pauses. Only after both the **[finished]** condition and the **[done]** condition are met does the machine move to the **Prompt for Account** state. Notice that there are no additional transition conditions after the join symbol.

Composite State

A composite state, also known as a composite, is a collection of related states. In a state diagram, a composite state is represented with a rounded rectangle with a label on top (much like a state) but with one or more states contained therein. These nested states are called substates. In many ways, they are like system boundaries in a DFD (see Chapter 11 Tool: Data Flow Diagram) or a nested component in a component diagram (see Chapter 40 Tool: Component Diagram).

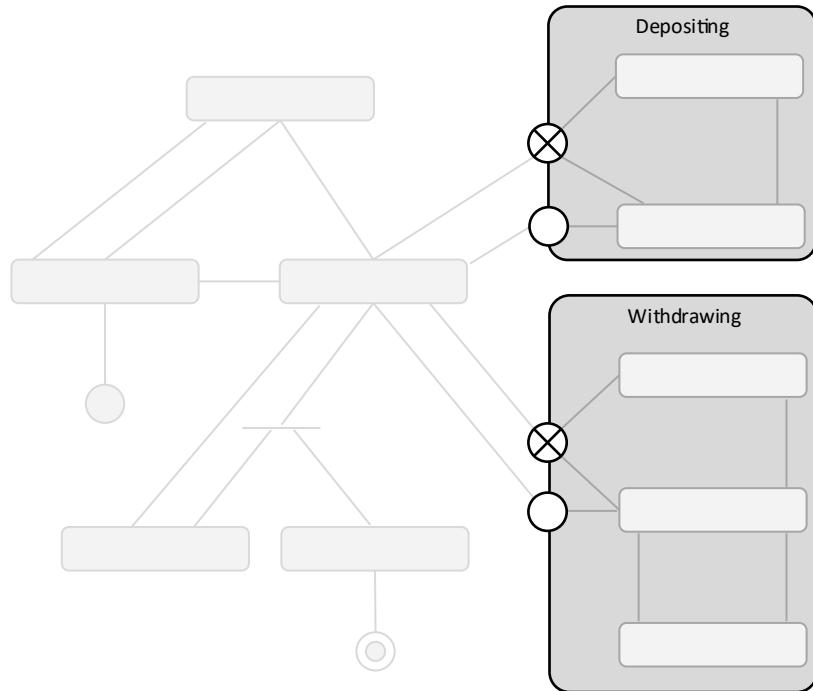


Figure 39.12:
State diagram
of a composite state

A state diagram can have many composite states or none; they are not a required part of a state diagram.

Best Practice 39.4 Use composite states to add clarity

Unlike states and transitions, composite states do not provide any descriptive power to a state diagram. In other words, every state machine represented with composite states can be equally represented without them. Their sole purpose is to add clarity to a potentially complicated diagram.

A composite state should be used to group a collection of states when the group has semantic meaning in the context of the application domain. In our ATM example, there are many states involved in the process of withdrawing or depositing money. By grouping these states into composite states, it becomes clearer what these substates represent and how they are related.

Best Practice 39.5 Composite states can be used to subdivide large state machines into multiple state diagrams

Though most state diagrams consist of less than a dozen states, there are some that may contain several dozen. These quickly become too busy to be easily understood. Composite states can address this problem by allowing the designer to describe subsystems in separate state diagrams. In our ATM example, the entire depositing subsystem can be moved to another state diagram and then the **Depositing** composite state simply refers to the other diagram.

There are times when there is an important distinction between substates in a composite and states outside a composite. When this is the case, it is helpful to distinguish transitions between substates (called local transitions) and those that cross a composite boundary (called external transitions). To make this distinction obvious in a state diagram, an entry point and an exit point may be used.

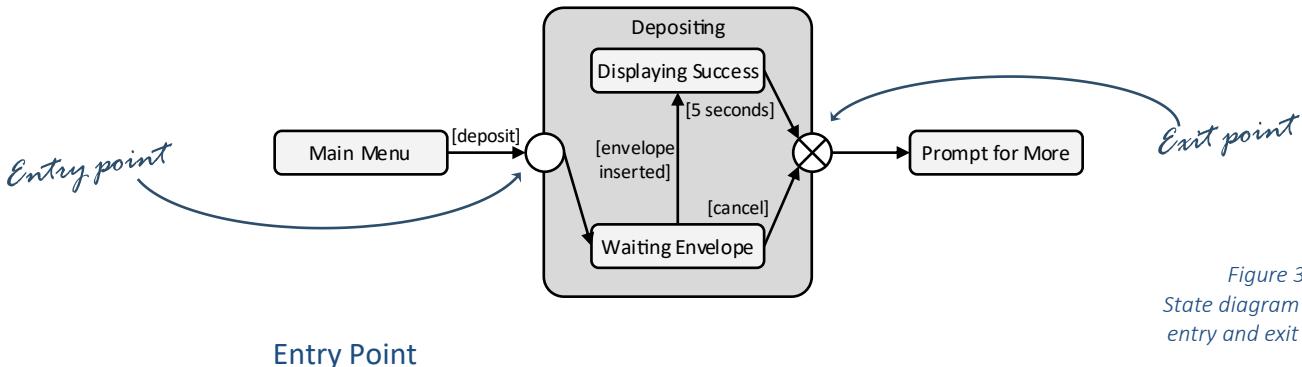


Figure 39.13:
State diagram of an entry and exit point

An entry point to a composite state is like the initial state in an overall state diagram. It is not a true state in that the application never occupies this state, but rather indicates which state the subsystem occupies first. An entry point is represented with a circle outline.

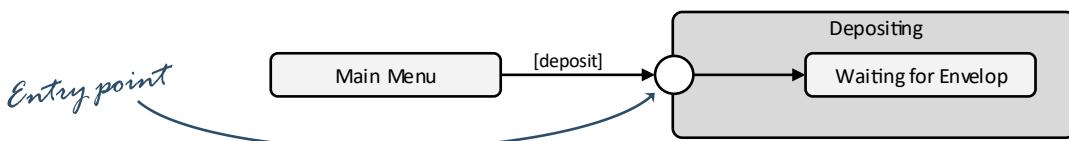


Figure 39.14:
State diagram of an entry point

Notice that there is a transition condition (the **[deposit]** condition) on the arrow leading into the entry point (from **Main Menu** to the entry point), but there is not one on the arrow leaving the transition (from the entry point to **Waiting for Envelope**). The transition condition leading into the entry point indicates the condition that must be met for the state to move into the composite state. However, since the entry point is not a true state, control moves directly to the next state indicated by the next transition. Thus, the system transitions directly from the source state outside the composite to the first substate in the composite state without pausing at the entry point.

Exit Point

An exit point to a composite state is analogous to the final state in an overall state diagram. As with the entry point and the final state, the program never actually occupies this state. It is just an indication that the state of the system has left the composite state. An exit point is represented with a circle outline and an X in the middle.

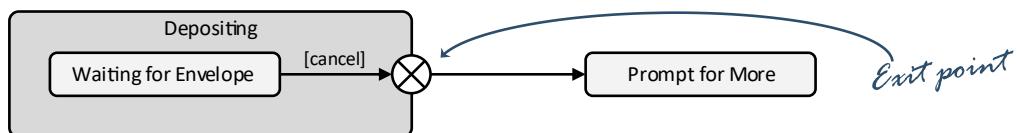


Figure 39.15:
State diagram of an exit point

As with the entry point, there is a transition condition on the transition leading into an exit point (the **[cancel]** condition in the transition leading from **Waiting for Envelope** to the exit point), but there is not a condition on the transition leading out of an exit point (the transition leading from the exit point to **Prompt for More**).

Designing State Machines

A state diagram is useful for modeling discrete behavior through finite state-transition systems. These state-transition systems are also called finite state machines (FSMs). Each can be mathematically modeled as directed graphs where the states are the nodes and the transitions are edges. Because they are graphs, we can draw conclusions about state designs that help optimize them and find bugs.

Identifying States

There is no simple way to identify states in an application domain. However, states do have certain characteristics which can make them easy to spot if you know what to look for.

Best Practice 39.6 States tend to be located where the system pauses

Though it is possible for a meaningful state to exist where the system remains for only an instant, most states capture a condition in which the system dwells. The reason for the pause can be diverse. It may be that the system is waiting for user input. It may be that the system requires another process to finish, that a certain amount of time must pass, or that a required piece of information must be present. In each case, the system situation can be modeled with a state.

Best Practice 39.7 States should represent unique conditions of the system

When a system has multiple conditions under which the system behaves essentially the same, then one state is probably better to model the conditions than several. To illustrate this point, consider a six shooter. This handgun can hold up to six bullets. One may be tempted to model this gun with seven states (zero bullets, one bullet, etc.). However, the gun behaves essentially the same if it has one bullet or six. Thus, a two-state design (empty and loaded) would be better than a seven-state design.

Best Practice 39.8 Avoid combining unrelated state machines

If you notice that two sides of a state diagram look very similar, then there might be some redundancy. This is typically because the design would be better served with two independent state machines rather than one. For example, consider a gun able to hold six bullets and having a trigger safety. The state diagram may be the following:

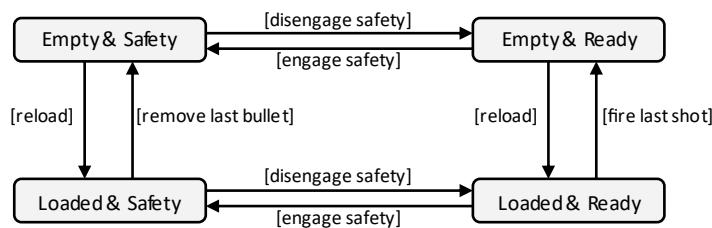


Figure 39.16:
One state diagram that
should be two

The large amount of redundancy can be removed when we realize that the safety state and the loaded state are mostly independent. We need two state diagrams here, not one.

Shortest Path

A path is an ordered collection of nodes that must be traversed to get between two locations on a graph. It is often the case that one would like to see the shortest path between two states in a state diagram. This can be useful when making performance determinations—transitioning between two states must execute all the state transition functions for every transition on the path. Identifying the shortest path also helps the designer discover if unnecessary steps exist in what should be a streamlined process.

Edsger Dijkstra, an influential computer scientist from the 1950s, conceived the shortest path first (SPF) algorithm. This algorithm performs a breadth-first search through a directed graph. To see how this works, consider the following simplified state diagram:

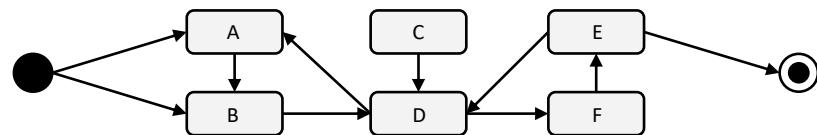


Figure 39.17:
State diagram before we
find the shortest path

The SPF algorithm starts at the source node and, from there, identifies nodes directly reachable from the start (nodes A and B). These are all given the value 1 because they are one transition from the start.

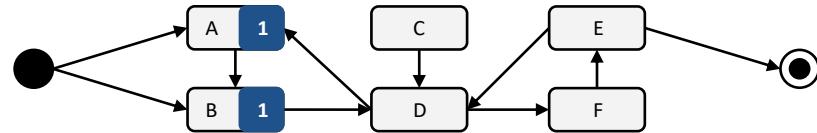


Figure 39.18:
SPF algorithm after
one round

Next, for each node that has the value 1, all the nodes that are reachable from there are given the value 2 (nodes B and D). Notice that a node already containing a number retains its previous number. For this reason, node B stays at distance 1.

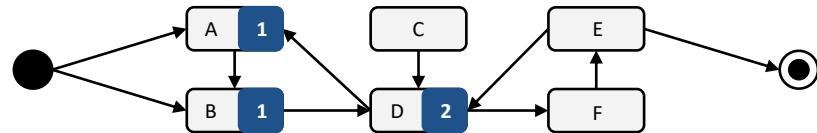


Figure 39.19:
SPF algorithm after
two rounds

This process continues until all the reachable nodes have a value associated with them or the destination node is found.

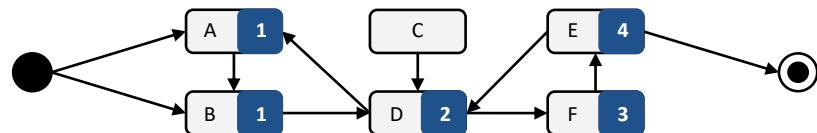


Figure 39.20:
SPF algorithm after
four rounds

When finished, we know the shortest distance from the start to the end node. We can also determine the shortest path between the two nodes if we also keep track of how we got to a given node. From this path, we can understand the transition conditions that must occur to move from the start node to the end node.

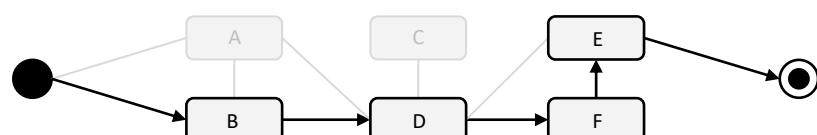


Figure 39.21:
The shortest path between
the initial and final state

Reachability

Reachability is the condition of one state in a state diagram being reachable from another. In other words, does there exist a set of transitions and states that can lead the program from a start node to an end node? If no such set of transitions exists, the two nodes are called orthogonal. This also means that any node reachable from the start node forms a distinct region from the set of nodes that can lead to an end node. These two regions are called orthogonal regions.

Orthogonal regions are sets of states between which no transitions exist

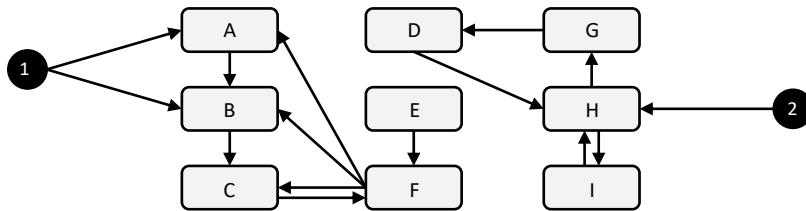


Figure 39.22:
A state diagram which
may contain an
orthogonal region

Reachability can be determined using the same algorithm we used to determine the shortest path between two states. If the shortest path is undefined, then the nodes are orthogonal. If we start at initial state (1), the following is the result of the SPF algorithm:

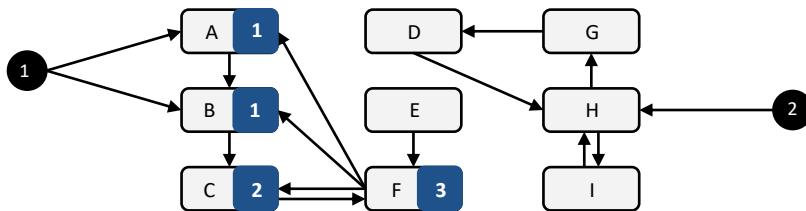


Figure 39.23:
A state diagram with the
SPF path length added

Notice that it is impossible to reach state D, E, G, H, or I from the start. This means that nodes A, B, C, and F form an orthogonal region to D, E, G, H, and I.

Best Practice 39.9 Delineate orthogonal regions with composite states

It can be the case that the existence of orthogonal regions is a desirable characteristic of a state diagram. This occurs when there should be no overlap between subsystems. In cases such as these, it is usually a good idea to encapsulate the subsystems in a composite state.

Best Practice 39.10 Verify that orthogonal regions are desirable properties of the state design

Usually, the existence of orthogonal regions is an indication of a defect in the design. If a program has a set of states which cannot be reached from the initial state, then why are these states in the design at all? Computing the SPF distance for all nodes in the state diagram will quickly identify such defects.

Implementing State Designs

There are two main ways to implement state designs. The first is most appropriate for simple designs utilizing a small number of states. The second scales well to extremely large and complex problems.

Integer States

Because there are a finite number of states in a state diagram, the simplest way to represent state is with an integer. Using this method, each state in the state diagram is assigned an integer value. Only one variable is then needed to represent the state of the system.

Best Practice 39.11 Use an enumeration to name your states

If your programming language supports enumerations, use them to name your states. This makes the code easier to read and makes it more difficult to confuse states. Going back to our ATM example, each state is given a name matching the label in the state diagram.

```
C#  
enum States { Welcome,  
    Accepting_Pin,  
    Main_Menu,  
    Accepting_Deposit,  
    Displaying_Balance,  
    Depositing_Money};
```

Figure 39.24:
C# with states represented
as an enumeration

The next step is to represent the transitions. If the program were to automatically move between two states (from `Displaying_Balance` to `Main_Menu` for example), then we represent it as the following:

```
C#  
public void update(Context context)  
{  
    switch (state)  
    {  
        case Displaying_Balance:  
            state = Main_Menu;  
            break;  
        case Depositing_Money :  
            ...  
            break;  
        default:  
            ...  
            break;  
    }  
}
```

Figure 39.25:
C# of a simple state
transition

Best Practice 39.12 Use a SWITCH/CASE statement to handle the state transition events with integer states

SWITCH/CASE statements make it easy to implement state logic and result in easily understood code.

Most state transitions include a transition condition. These conditions govern whether a given transition is allowable. For example, consider the following state diagram representing the ATM system where the current state is **Main Menu**:

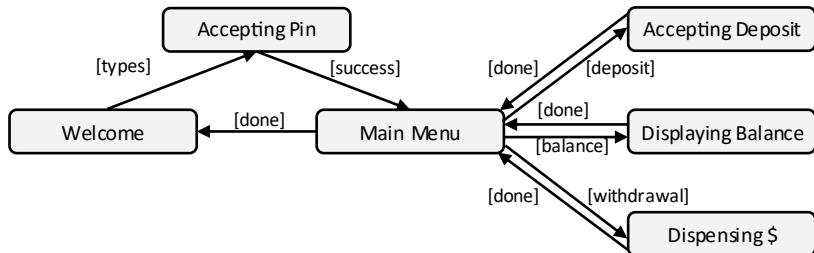


Figure 39.26:
A state diagram where
the main menu is
the current state

Notice that we can transition to four states from **Main Menu**: **Welcome**, **Accepting Deposit**, **Displaying Balance**, and **Dispensing \$**. The selected event depends on which button the user selected from the ATM user interface. These transition events can be handled with a query to the context which knows whether the user has selected a given button.

```

C#
public void update(Context context)
{
    switch (state)
    {
        case Main_Menu:
            if (context.done_selected())
                state = Welcome;
            else if (context.deposit_selected())
                state = Accepting_Deposit;
            else if (context.balance_selected())
                state = Displaying_Balance;
            else if (context.dispensing_selected())
                state = Dispensing_Balance;
            break;
        case Depositing_Money:
            ...
            break;
        default:
            ...
            break;
    }
}
  
```

Figure 39.27:
C# state update method
with state transition events

There are several advantages to this approach. First, all the state transition logic is represented in a single function, making it easy to verify that the code honors the logic described in the state diagram. Second, the design is very efficient. Very little code needs to be executed when the program moves from one state to another. Finally, the code is very understandable. The entire state design is implemented with programming primitives which are well understood by all programmers.

There are two big drawbacks to the integer state design. The first is that there is not a single program entity which represents a state. Instead, the state code can be in several functions and variables. Modifying a state or adding a new state can be much more difficult than necessary. The second and most important limitation is that of scale. When the state design contains a hundred states and a thousand transitions, then the state function becomes immensely complicated and difficult to verify. Clearly, a more sophisticated solution is needed.

State Design Pattern

The state design pattern is an OO strategy developed by the original “gang of four” (GoF): Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The core idea is to represent each state as a class, which also contains the logic of the transition conditions. There are several components to the state pattern:

1. **Context.** The client which holds the state variable. It also has a `setState()` method which allows the state to be changed.
2. **State.** The state is an abstract class from which all concrete state classes are derived. When an event occurs, then the `handle()` method gets called which determines if state is to be changed.
3. **Concrete State.** Every state in the state diagram is represented as a concrete state whose `handle()` method contains all the transition conditions out of the state. If a context event requires a state change, then the context’s `setState()` method gets called.

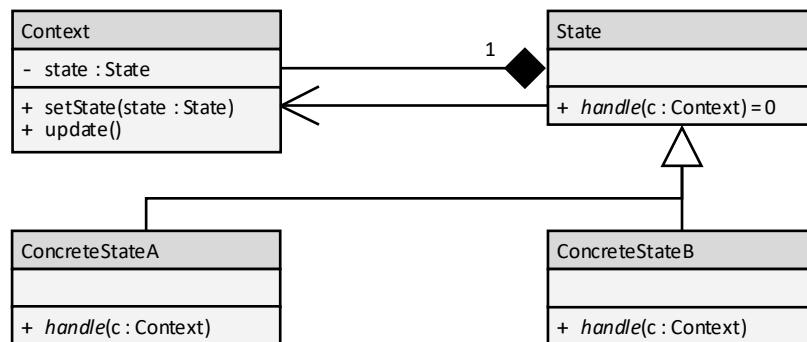


Figure 39.28:
Class diagram of the
state design pattern

The **Context** class has a **State** member variable. Every time its `update()` method is called, the state of **Context** can be changed. Though all these state changes are initiated by one of the concrete states, the `update()` method asks the current state if the state needs to be changed:

Pseudocode

```
Context.update()
    state.handle(this)
```

Figure 39.29:
Pseudocode of the
context's update method

The **State** abstract class has only a single method (called `handle()`), and that method is a pure virtual function. This method is called by **Context** when a state may be changed. Each concrete context implements `handle()` which takes a **Context** as a parameter. It is through this context that the various concrete state classes can determine whether a state change is necessary. When this happens, then the concrete state specifies the new state by calling the client’s `setState()` method. In the following example, **ConcreteStateA** will always transition immediately to **ConcreteStateB**.

Pseudocode

```
ConcreteStateA.handle(context)
    context.setState(new ConcreteStateB)
```

Figure 39.30:
Pseudocode of a trivial
state transition

To see how this works, we will return to our ATM example.

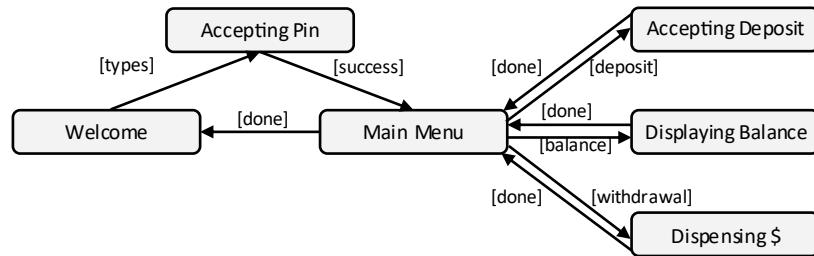


Figure 39.31:
State diagram of the
ATM system

The context will interface with the rest of the ATM system. In this case, it will have a collection of methods indicating the status of the various buttons and devices.

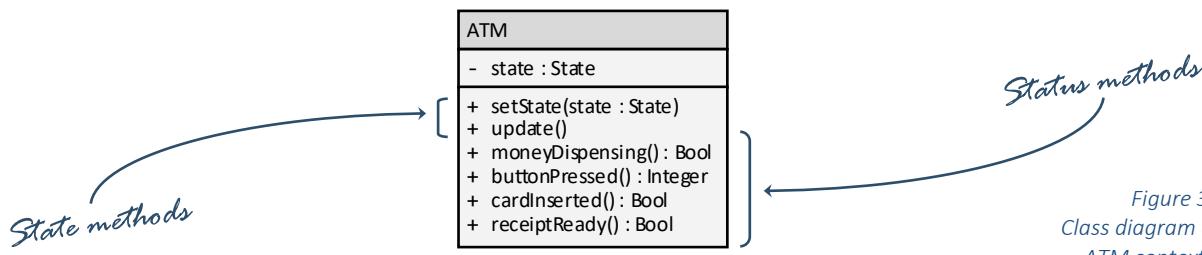


Figure 39.32:
Class diagram of the
ATM context class

Note the two types of methods in the context `ATM` class. The first are those required for the state design pattern: `setState()` and `update()`. The second are needed for the concrete classes to inquire about the status of the machine for the purpose of determining whether a state transition should be made. These are the status methods: `moneyDispensing()`, `buttonPressed()`, `cardInserted()`, and `receiptReady()`.

There are several concrete classes. At this point, we will just focus on `MainMenu`. The key feature to this class is the `handle()` method which determines how states will transition.

Pseudocode
<pre> MainMenu.handle(atm) IF atm.buttonPressed() = DONE OR atm.timeout() atm.setState(new Welcome) ELSE IF atm.buttonPressed() = DEPOSIT atm.setState(new AcceptingDeposit) ELSE IF atm.buttonPressed() = BALANCE atm.setState(new DisplayingBalance) ELSE IF atm.buttonPressed() = WITHDRAWAL atm.setState(new dispensingMoney) </pre>

Figure 39.33:
Pseudocode of a complex
state transition function

Notice that `handle()` calls `setState()` to handle a state transition. It also asks the context class (`atm` in this case) about the status of the system so a state transition decision can be made. A final feature of this design is that every transition leading out of a given state is fully described in the concrete class associated with that state. In other words, every state is completely self-contained. All questions about a single state are present in that state's concrete class.

Examples

Example 39.1: Simple State Diagram

This example will demonstrate how to create a state diagram matching a problem definition.

Problem

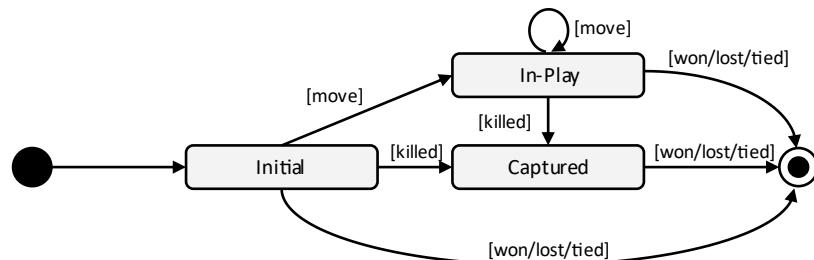
Consider the following scenario:

A rook (otherwise known as a castle) is a chess piece that can move horizontally or vertically in a straight line. The rook, like all the other pieces except the king, can be captured. This means that it is removed from the game. Also, the rook can be involved in a castling move with the king. For this to happen, the rook must not have moved yet in the game.

Create a state diagram describing the various states of the rook.

Solution

From the problem definition, there are three states the rook can be in: initial (meaning it has not yet moved), in-play, and captured. The rook always begins in the initial state. The game can end with the rook in either of the three states.



From this state diagram, we can see that the game can end if the rook had not been moved, if it had been moved and not captured, or if it had been captured. We can see that the rook can only move when it is in the initial state or the in-play state; it cannot move from the captured state. Finally, we can see that the rook can only be killed in the initial state and in the in-play state; it cannot be killed in the captured state.

Example 39.2: Complex State Diagram

This example will demonstrate how to create a state diagram matching a problem definition.

Problem

Consider the following scenario:

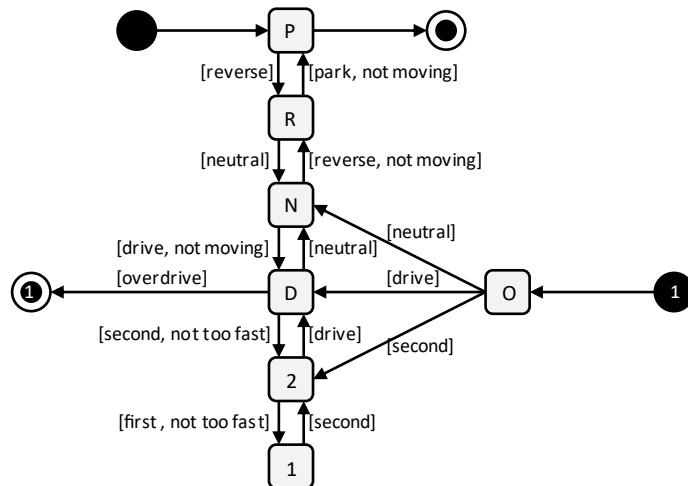
The automatic transmission on a car has several gears: park (P), reverse (R), neutral (N), drive (D) with an associated overdrive (O), second (2), and first (1). There are several rules:

1. The car always starts and stops in park.
2. You can shift into reverse from park and into park from reverse when not moving.
3. You can shift into neutral from reverse and from neutral back to reverse only when you are not moving.
4. You can shift into drive from neutral when not moving and from drive back to neutral even when moving.
5. You can shift into second from drive if you are not going too fast but can always shift from second back to drive.
6. You can shift into first from drive if you are not going too fast but can always shift from first back to drive.
7. You can shift from drive to overdrive. You can shift from overdrive into drive, second, or neutral.

Create a state diagram describing the various states of the transmission.

Solution

The state diagram for the automatic transmission is the following:



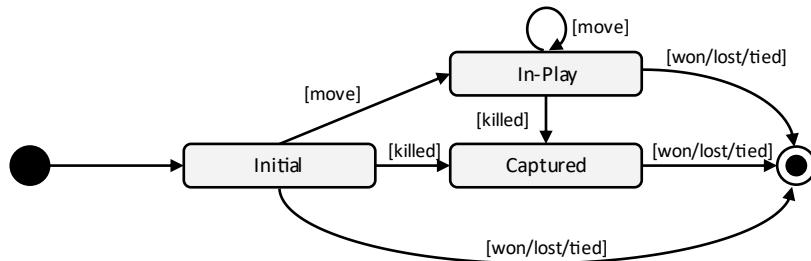
Notice that the labeled initial and final states are used to declutter the state diagram because the arrows were too tight around the overdrive state.

Example 39.3: Integer States

This example will demonstrate how to implement a simple state diagram using integer states.

Problem

Implement the following state diagram in pseudocode:



Solution

The first step is to name the states. This will be done with the enumeration construct.

Pseudocode

```
ENUMERATE States as Initial, InPlay, Captured
```

The next step is to implement the state change function called `update()`. This is accomplished with a switch/case statement. Each transition condition is accomplished by calling the appropriate function in the `context` object.

Pseudocode

```
update(context)
SWITCH state
CASE Initial
    IF context.move()
        state ← InPlay
    ELSE IF context.killed()
        state ← Captured
CASE InPlay
    IF context.killed()
        state ← Captured
```

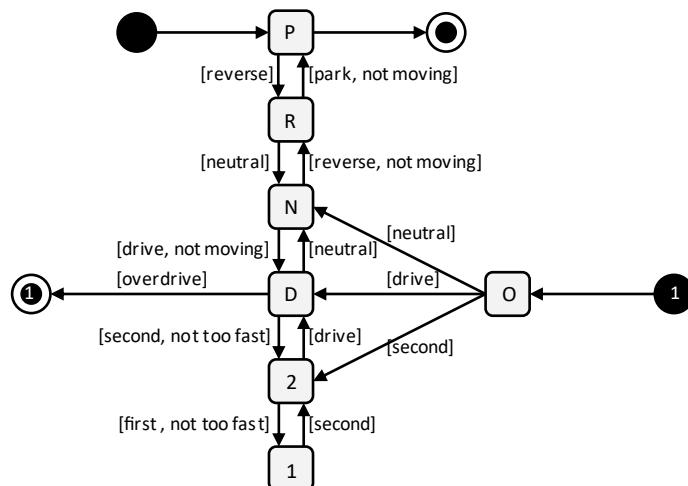
Notice that we do not need to implement the transition from `InPlay` to `InPlay`; leaving the `state` variable unchanged has the same effect. Also notice that we do not need to implement the state transition to the final state. Since the game is over, we can leave the final state as a pseudo-state.

Example 39.4: State Design Pattern

This example will demonstrate how to implement a simple state diagram the state design pattern.

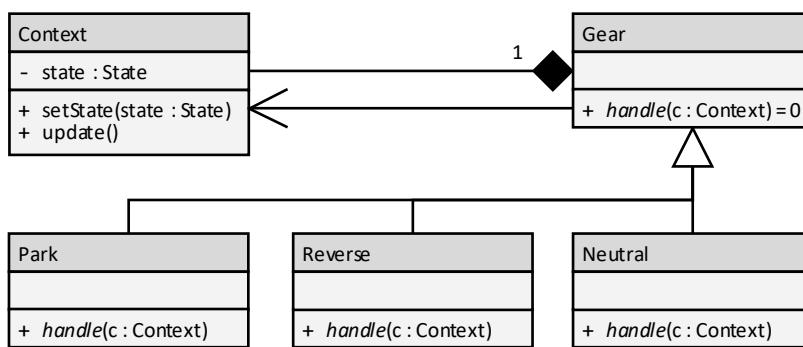
Problem

Implement the following state diagram:



Solution

The state diagram will include only three gears, but the other four look the same.



The handle() function in each gear concrete class handles the events.

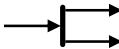
Pseudocode

```
Reverse.handle(context)
  IF context.selected() = NEUTRAL
    context.setState(new Neutral)
  ELSE IF context.selected() = PARK and context.notMoving()
    context.setState(new Park)
```

Exercises

Exercise 30.1: Name the Symbol

Name the state diagram symbol.

Symbol	Symbol Name
	
	
	
	
	
	
	

Exercise 39.2: Terms and Definitions

From memory, define each of the following terms.

Term	Definition
Local transition	
State	
Substate	
Transition arrow	
FSM	
Orthogonal region	
State diagram	
Transition condition	
External transition	
SPF	

Exercise 39.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
All state diagrams need an initial symbol.	
Large state diagrams can be subdivided into several smaller ones.	
One condition of the system should map to exactly one state.	
Integer state implementations should not name the states with enumerations.	
State labels should be action verbs.	
Composite states add clutter and confusion.	
Orthogonal regions are good candidates for composite states.	
State labels are often adjectives and nouns.	
States are often where the system pauses.	

Problems

Problem 39.1: Robotron: 2084

Consider the following scenario:

Robotron: 2084 was an arcade video game released in 1982. This game consists of 255 levels, the first nine being: basic wave, spheroid wave, crowd wave, wave 4, brain wave, wave 6, tank wave, hulk wave, and grunt wave. If the player has cleared all the enemies from the screen, then the game advances to the next wave in the sequence. If the player gets killed and runs out of extra lives, then the game ends.

Create a state diagram describing the waves of *Robotron: 2084*.

Problem 39.2: Movies

Consider the following scenario:

A film goes through two main phases: production and distribution. In the production phase, there are six stages: development, pre-production, production, principal photography, wrap, and post-production. It is possible for a film to retreat in these stages or even be canceled. Every movie, including animated movies and indies, goes through these six development stages. Distribution, on the other hand, can go through a wide variety of stages. It can go directly to cinema, can go directly to a streaming platform such as Netflix, can go straight to video, or just about any other sequence. The only exception is that films that were streamed or sold to video do not return to the cinema.

Create a state diagram describing the lifecycle of films.

Problem 39.3: Chess Pawn

Consider the following scenario:

A pawn is a chess piece that can advance one square in its file if no piece stands in its way. However, if the pawn has not yet moved, it can advance two squares provided neither of the two squares are occupied. If an enemy piece is diagonally forward one square to the pawn, then the pawn has the opportunity to capture that piece. If an enemy pawn has just advanced two squares and sits adjacent to a pawn, then that pawn can capture the enemy using an en passant capture. The pawn can be captured by any enemy piece. Finally, if the pawn makes it to the final rank of the board, then it can be promoted to any other piece except for a king.

Create a state diagram the different states of a pawn.

Problem 39.4: USPS

Consider the following scenario:

The United States Postal Service (USPS) maintains status information for every package they deliver. This list of statuses include delivered, notice left, delivery status not updated, receptacle blocked, no access, in transit, alert, and lost.

Research the meaning of the various USPS packages status codes and what type of transitions occur for a package to move from one state to another. Create a state diagram representing your findings.

Problem 39.5: Hired

For an employer to hire a new employee, the candidate's application moves through several states. Research how five companies hire new employees. Represent all these processes in a single state diagram.

Problem 39.6: US Navy

Consider the following scenario:

The United States Navy has nine enlisted ranks (Seaman Recruit through Master Chief Petty Officer of the Navy), one warrant officer ranks (Chief Warrant Officer) and eleven officer ranks (Ensign through Fleet Admiral).

Research the requirements for promotion for each rank and what type of transitions may occur. For example, is it possible to go directly from a Seaman Recruit to a Fleet Admiral? What about the other way around? Create a state diagram representing your findings.

Problem 39.7: Monopoly

Consider the following scenario:

Monopoly is a board game originally published in 1903 but still popular today. This game consists of several properties. Each property can have a variety of statuses, from available to purchase to mortgage to improved with houses or hotels.

Research the rules of monopoly and create a state diagram describing all the different states that a single property can be in.

Problem 39.8: Airline Flight

Consider the following scenario:

When one goes to an airport, the first order of business is to check the flight status chart. This chart describes the status of each flight that one may wish to board. Some of the statuses are loading, delayed, at gate, etc.

Create a state diagram the different states of an airline flight. Create the pseudocode of the `update()` function that handles the state transitions using the integer states implementation.

Problem 39.9: Robotron Implementation

If you have created a state diagram for Problem 39.1, create the pseudocode of the `update()` function that handles the state transitions using the integer states implementation.

Problem 39.10: Pawn Implementation

If you have created a state diagram for Problem 39.3, create the pseudocode of the `update()` function that handles the state transitions using the integer states implementation.

Problem 39.11: USPS Implementation

If you have created a state diagram for Problem 39.4, create a class diagram describing an implementation of your design using the state design pattern. Additionally, provide the pseudocode for the `handle()` method for one of the states.

Problem 39.12: Hired Implementation

If you have created a state diagram for Problem 39.5, create a class diagram describing an implementation of your design using the state design pattern. Additionally, provide the pseudocode for the `handle()` method for one of the states.

Challenges

Challenge 39.1: Science Fiction Spaceship

Consider the spaceship in the science fiction movie of your choice. This could be the Millennium Falcon from the Star Wars series, it could be the Starship Enterprise from Star Trek. It could be the Rocinante from the Expanse. In this movie, the ship has a variety of different statuses. For example, there might be shields that can only be up when the ship has a certain amount of power. There could be mechanical or maintenance problems. It could be low on fuel or ammunition. Through careful “research,” identify all the states of this spaceship and transitions. Represent this in a state diagram.

Challenge 39.2: Video Game Rooms

There is a style of video game where the player moves from room to room. Each room typically has puzzles to solve, obstacles to overcome, or prizes to collect. Some of these games allow the player to move between the rooms in a non-linear fashion, making this style of game distinct from that of a level-based game. Examples of games of this style include the Atari game E.T., the Dungeons & Dragons board game, choose your own adventure books, and many more.

Find such a game with a dozen or two rooms. Describe the rooms in a state diagram and implement it using the state design pattern in your language of choice.

Challenge 39.3: Visa

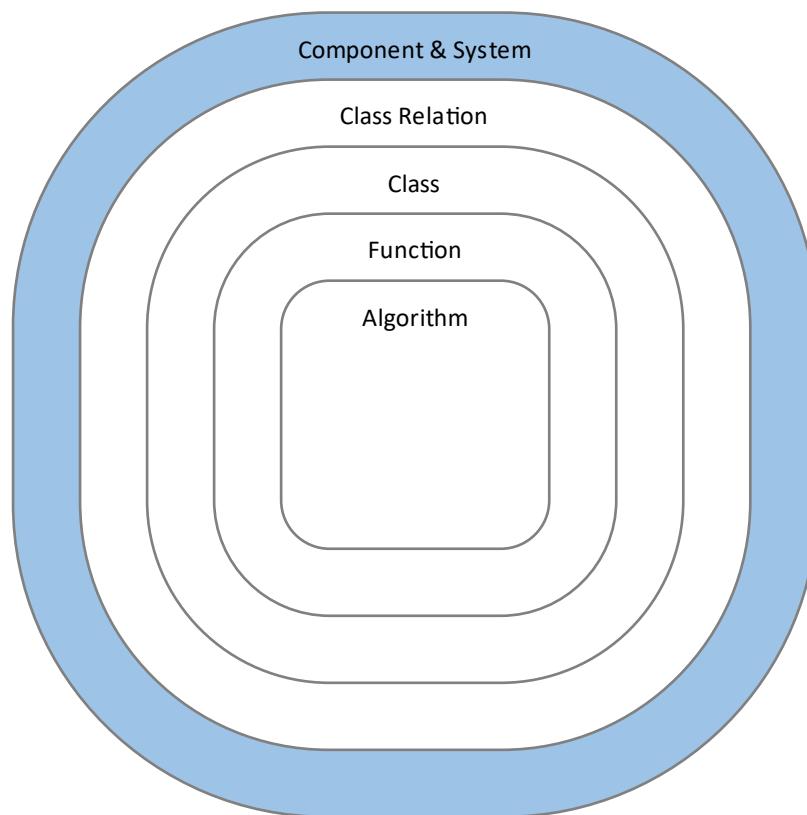
Non-native individuals wishing to spend time in the United States of America are required to have immigration clearance. This clearance can take a variety of forms depending on the duration of the planned stay and the types of things that the individual will be doing while in the country. This clearance is called a visa.

Research the various types of visas, the steps one must go through to obtain a visa, and the conditions that may occur to cause one to lose a visa. Create a state diagram representing the US visa immigration process.

System Design

Unit 4

A system is a holistic view of the entire application or software package under development. Often a system is large enough that it makes sense to subdivide it into several semiautonomous subsystems called components. In the past, systems and components were developed from functions or even algorithms. Modern software systems have significant parts of the design built from classes. Thus, unlike inner layers of software design, component and system design does not necessarily build from all the inner layers.



Component Diagram

Component diagrams are design tools allowing for the description of software systems and interfaces of any size or complexity.

The stated purpose of all software visualization tools, be that the flowchart or the class diagram, is to free engineers from the minutiae of detail that programming languages demand so they can focus on the big picture. Despite this goal, every single tool presented thus far forces the designer to identify individual variables, parameters, or functions. The missing piece is something that allows designers to work above that level. This is where component diagrams come in.

Component diagrams are design tools allowing for the description of software systems and interfaces of any size or complexity.

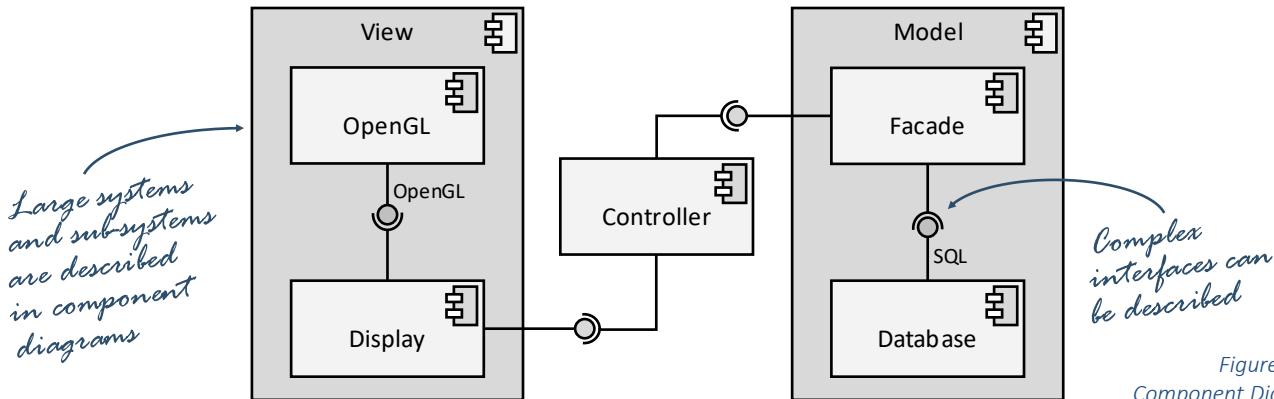


Figure 40.1:
Component Diagram

The heart of component diagrams is the notation of components: modular units designed to be upgradable and replaceable. Each component is connected to the other components of the system through well-defined interfaces. These interfaces can be simple function calls or complex message passing involving many complicated file formats. The more software designers think in terms of components and interfaces, the easier it will be to understand the resulting system, add new functionality, or swap in new technology.

Component diagrams have the following properties:

Property	Description
Use	Describe the organization of system components
Viewpoint	Development: showing the organization of the code
Strength	Provides an easy-to-understand roadmap of the system
Weakness	Few details of the components or interfaces are present

Viewpoints

Component diagrams represent subsystems of arbitrary size and complexity

A component diagram is a structural viewpoint like a structure chart. The only substantial difference between a structure chart and a component diagram is the size of the things they contain. Structure charts fundamentally are about functions; every circle in a structure chart represents a function and every line represents how one function calls another. Component diagrams, on the other hand, represent subsystems of arbitrary size and complexity. It could be something as small as a function or a class. It could be something as large as a web server.

The big question is this: When do you use a component diagram, as opposed to the other design tools? The short answer is “in the beginning.” Of course, the long answer is more complex.

Component diagrams are usually the tool of choice early in the design process for large systems. Several rounds or layers of design are typically conducted with component diagrams before more detailed viewpoints are employed.

Component diagrams are the tool of choice early in the design process for large systems

The following decision tree provides a more comprehensive view on how the various design tools are used. There is one important thing to note: Both the DFD and the component diagram are suitable for large system design. The DFD focuses on the flow of information whereas the component diagram focuses on the organization of the various components.

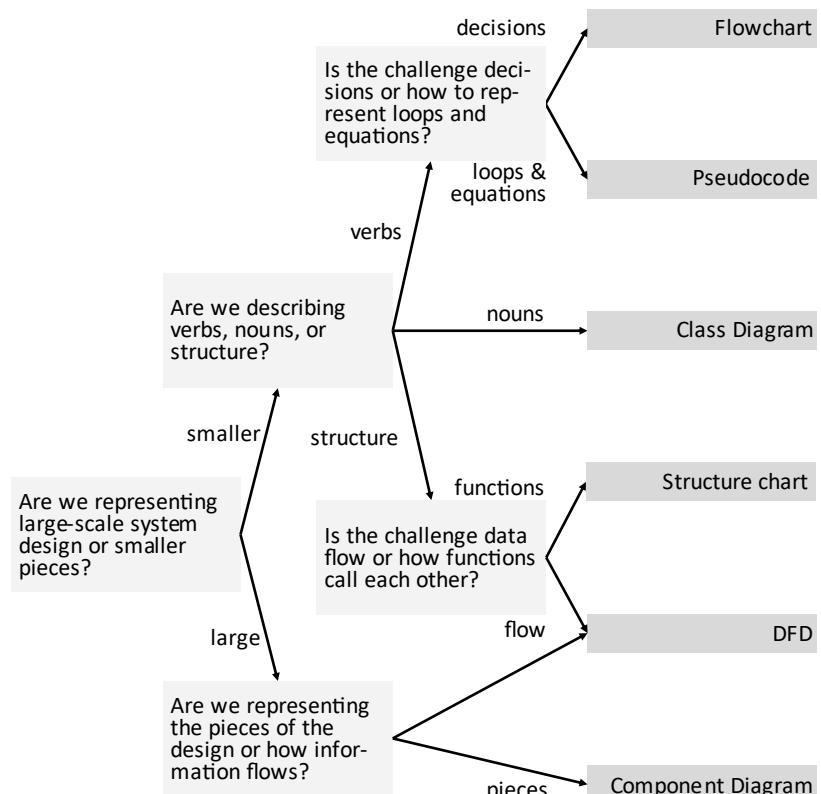


Figure 40.2:
Decision Diagram for
choosing a design tool

Component Diagram Elements

A component diagram consists of three elements: components, interfaces, and ports.

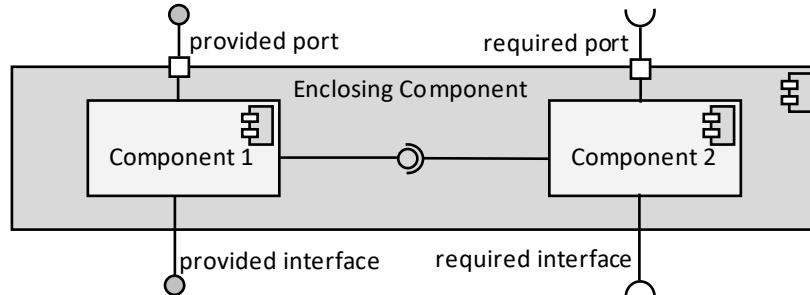


Figure 40.3:
*Component diagram
elements*

There are several things to notice about component diagrams. First is that there is no particular order or hierarchy. With a flowchart, structure chart, or class diagram, the top and bottom of the chart have special meaning. There is no such orientation with component diagrams.

Second, the lines do not imply the direction of data flow as they do with structure charts or data flow diagrams. They do not imply levels of abstraction as they do with class diagrams. In fact, interface lines have no implied direction of any kind. A provided interface, for example, could primarily give or receive information.

Finally, component diagrams do not imply how large or complex a component or interface may be. The same component box can be used to represent a single function or a subsystem consisting of ten million lines of code.

The various symbols used to represent component diagram elements are the following:

Symbol	Meaning
	A component, representing a system, subsystem, services, library, class, or other part of the system
	A required interface, meaning something needed by the component for it to function properly
	A provided interface, meaning a service offered to another component
	An assembly connector, representing the mating of a required and a provided interface
	A port, which is an explicit and often formal interface through which all communication travels

Component

A component is a self-contained and substitutable unit of a system designed to fulfill a well-defined function. There are several parts of this definition. First, the *self-contained* part of the definition refers to the need to have the smallest number of dependencies. Ideally, a component would have no dependencies on other parts of the system and a minimal number of dependencies with external entities. Second, a component is designed to be *substitutable*, meaning one should be able to swap-out one component with another as long as the two implement the same interfaces. Finally, a component performs a well-defined function, related to our cohesion property of modularization and fidelity property of encapsulation.

A component is represented as a rectangle (called a classifier rectangle) with the component name in the center. This rectangle also has a component icon in the upper-right corner, which serves no purpose other than to make it visually distinct from other software design tools.

A component can also contain other components. Here, the enclosing component has external interfaces and ports whereas the interior components only connect to each other. There can be any number of components inside a given enclosing component, but too many make the component diagram needlessly complex.

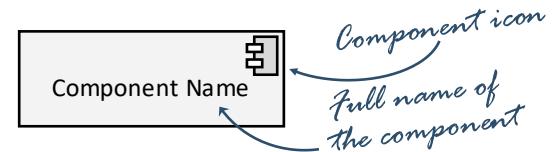


Figure 40.4:
Simple component

One component
can contain
several
subcomponents

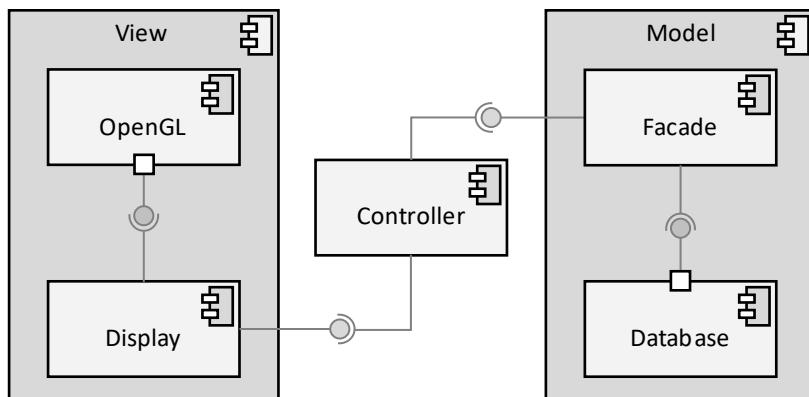


Figure 40.5:
Enclosed components

When the component diagram becomes too complicated, then it often becomes necessary to break it up into sub diagrams. To do this, no special notation is needed. If, for example, Diagram #1 contains a component called “Database Façade,” then Diagram #2 can be called “Database Façade” and show all the subcomponents contained therein. In other words, as long as the same interfaces are presented in Diagram #1 and Diagram #2, the two diagrams are considered consistent.

Interfaces

Components are connected by interfaces. These can be simple function calls or complex session-based interactions. Any time data are shared between components regardless of the technology used to manage that interaction, an interface is used to depict it. There are two broad classifications of interfaces in component diagrams: provided and required interfaces.

Provided

In their simplest manifestation, provided interfaces generate output that is consumed by required interfaces. However, most scenarios are more complex than this. The provided interface responds to requests made by required interfaces. In component diagrams, a provided interface is the ball in a ball-and-socket connection (often called the “lollipop”). It is a line with a circle outline at the end. Usually provided interfaces also include a label describing the name of the utilized protocol if there is one.

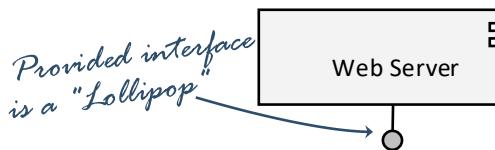


Figure 40.6:
Provided interface

For example, consider a web server. This component responds to web page requests by sending the contents of the desired web page. Notice that this is a “provided” interface because the web server does nothing until a request is made. Now, since we know nothing about who is making this request (it could be a browser or a web crawler), we leave the interface as an open ball connection.

Notice that the web server fulfills the contract that the interface specifies. Presumably, any other component that can fulfill the same contract regardless of the underlying technology would be an acceptable implementation of this component.

Required

A required interface is where a component makes a request of another component. This is often described as a “using” interface. A required interface is drawn with a half-circle (often called a “socket”).

For example, consider a web browser consuming HTML data provided by an external source. When the user clicks on a link or enters a URL into an address bar, an interface is created with a web server. Because the browser is initiating the request, the interface is required.

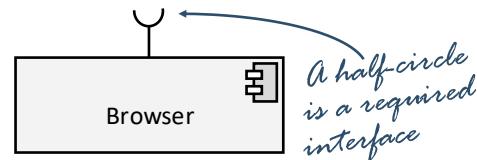


Figure 40.7:
Required interface

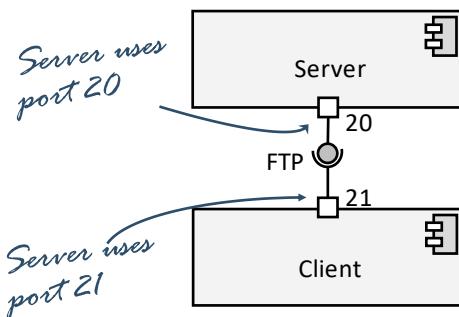


Figure 40.8:
Assembly connection

When a provided interface is mated with a required interface, an assembly connection is created. The exact nature of this connection is not always known at system design time. For example, the engineer building the web server does not know which browser will issue the request, and the engineer building the browser does not know which web server will respond to the request. In both cases, the ball-and-socket connection is left opened. However, when the nature of the connection is known, then both are drawn in the component diagram as an assembly connection.

Port

A port is a special type of interface distinguished by the degree of formality by which it is defined. There is no hard-and-fast rule as to when an interface qualifies as a port, falling more into the realm of “you will know it when you see it.” For example, most would consider SQL to be a port to a database. It is a standard and well-documented interface through which all database communication occurs. Another example would be a plug-in for your browser allowing it to understand a different language or protocol. This plug-in occurs because the browser created a special application program interface (API) through which third parties can write their own extensions. Since this interface is explicit, formal, and well documented, most would consider it to be a port.



Ports are represented as a small box at the base of an interface. The port is often labeled when the interface is named. For example, an internet-connected computer will make web-based requests through the file transfer protocol (FTP). The FTP server's port number is 20 whereas the client's port number is usually 21. Since the client is the one making the request, then the required interface is on the client side and the provided interface is on the server side.

Figure 40.9:
Ports

Single ports can also conduct multiple protocols. For example, a list management program may contain some synchronization functionality. In this case, the client component may have a subcomponent responsible for syncing with the server and the server component may have a subcomponent responsible with syncing with the client. Since either the client or the server could initiate a sync, then both must contain both provided and required interfaces.

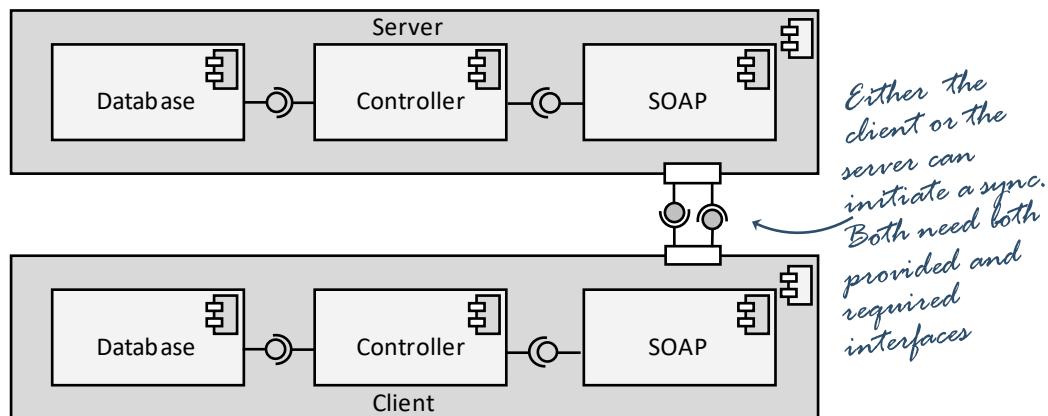


Figure 40.10:
Ports with multiple
interfaces

Variations

The most used component diagram format is the one presented, with the component name in the box connected with ball-and-socket joints. There are two popular variations: “fleshed-out” component diagrams, and UML 1.0 component diagrams.

Fleshed-Out Component Diagrams

Some component diagrams use stereotypes (labels in chevrons) explaining their purpose. For example, a typical component may include the keyword «component» on the top. This could be replaced with «subsystem» or even «class» depending on how that component is used. The following components are equivalent:

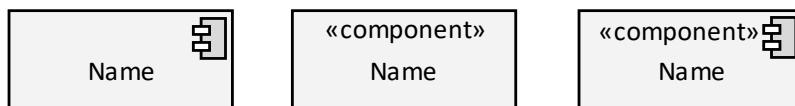


Figure 40.11:
Different ways to
represent a single
component

Another common variation is to include a second rectangle to enumerate the provided and required interfaces. This can be useful if the number and complexity of the interfaces are large. For simpler interfaces, however, the connectors are enough.

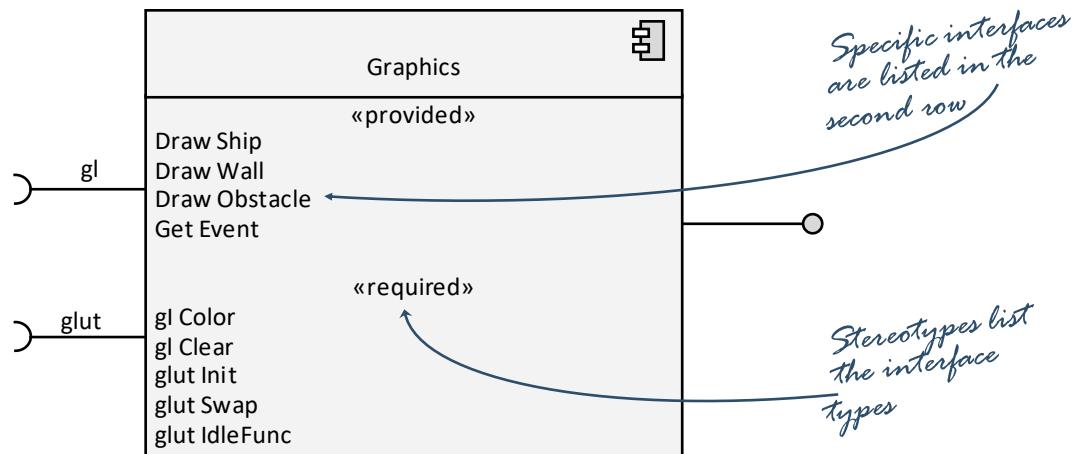


Figure 40.12:
Interfaces listed in the
second row

Finally, the component diagram notation allows to specify complex dependency relations. Here, *has-a* arrows from class diagrams are used to explain how one controller component manages a dozen web interface components.

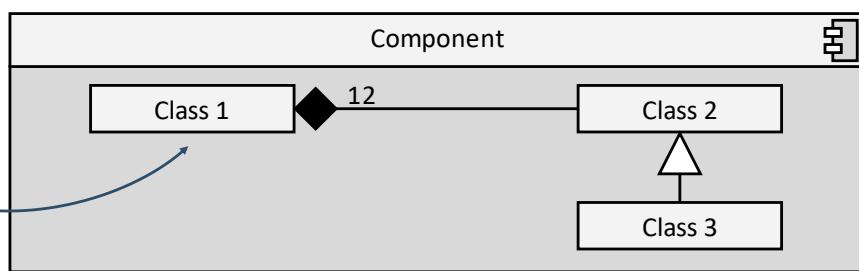


Figure 40.13:
Details inside a
component

UML Component Diagram 1.0

The standards organization that governs component diagrams is the UML group. While UML Component Diagrams 2.0 are the most used format today, it is not uncommon to see UML Component Diagrams 1.0 utilized. There are three main differences between the two versions:

- With UML 1.0 component diagrams, all interfaces are represented with white rectangles coming out the side.
- The Required interfaces are represented with dashed arrows in 1.0, whereas they are represented with lines and half-circles with 2.0.
- UML 2.0 uses a “component icon” in the upper-right-hand corner whereas there is no icon in 1.0.

To illustrate the differences, we will return to our model from Figure 40.1.

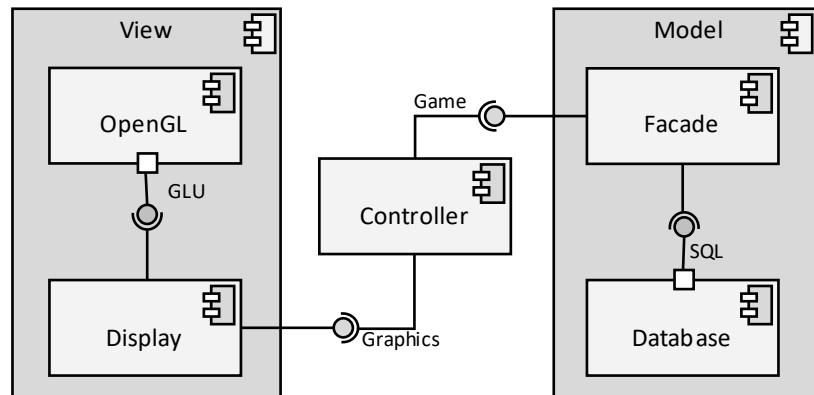


Figure 40.14:
Component Diagram 2.0

The following is UML Component Diagram 1.0 notation of the same design.

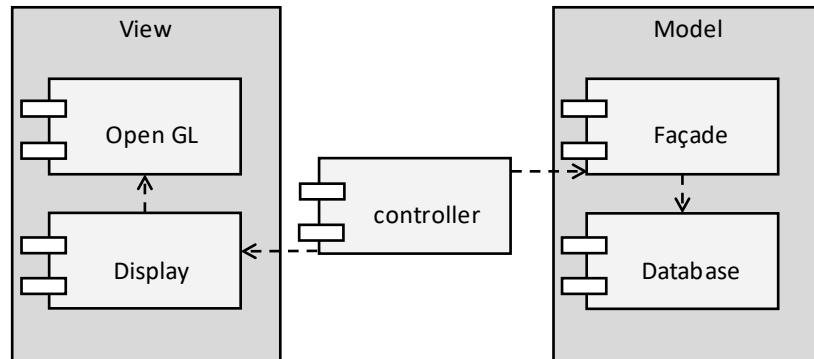


Figure 40.15:
Component Diagram 1.0

Key differences include:

- The interfaces are represented as dashed arrows in 1.0 and lollipops in 2.0.
- Ports are not present in 1.0, they are small squares in 2.0.
- There is no component icon in 1.0, but they are needed in 2.0.
- There are two rectangles sticking out the right side of a 1.0 component, there are no such items in 2.0 components.

Designing with Component Diagrams

There are several guidelines one should consider when developing a software system with a component diagram. Each of these are neither process steps nor rules. Instead, they are considerations that one should consider at the early steps of the design process.

Best Practice 40.1 Maximize reusability

Design components to be as useful as possible to other parts of the system. This may mean making interfaces more complete or generic than is strictly necessary for the system we are building.

Best Practice 40.2 Make components replaceable

Remember, components are able to be reused and replaced. You must ask yourself, “if someone were to swap out this component with another, how difficult would that be?”

Best Practice 40.3 Maximize cohesion

Strive to make every component do one thing and one thing only. This relates directly to the concept of cohesion for modularization and fidelity for encapsulation. As with functions and classes, if you can't come up with a good name for your component, then you probably have a cohesion problem.

Best Practice 40.4 Minimize coupling

Strive to make the interfaces as simple as possible between components and with the outside world. A well-defined interface does not betray any implementation details of the component, is easy for another component to honor, and is easy to extend.

Best Practice 40.5 Enclose appropriately

If several components have somewhat tight coupling between themselves and reveal more about the inner workings of the system than you would prefer, consider enclosing them in a larger component. The goal here, of course, is to minimize system complexity, not to add a new level of complexity to the system.

Best Practice 40.6 Consider compatibility

All changes to interfaces must be backwards compatible. This means that old components should work with new versions of the interface. Most changes to interfaces should be forward compatible as well. This means they should allow for extension and elaboration to accommodate future functionality that one may wish to add to the system. This greatly decreases the maintenance costs of the system.

Examples

Examples 40.1: Client-Server

This first example will demonstrate a common web interface.

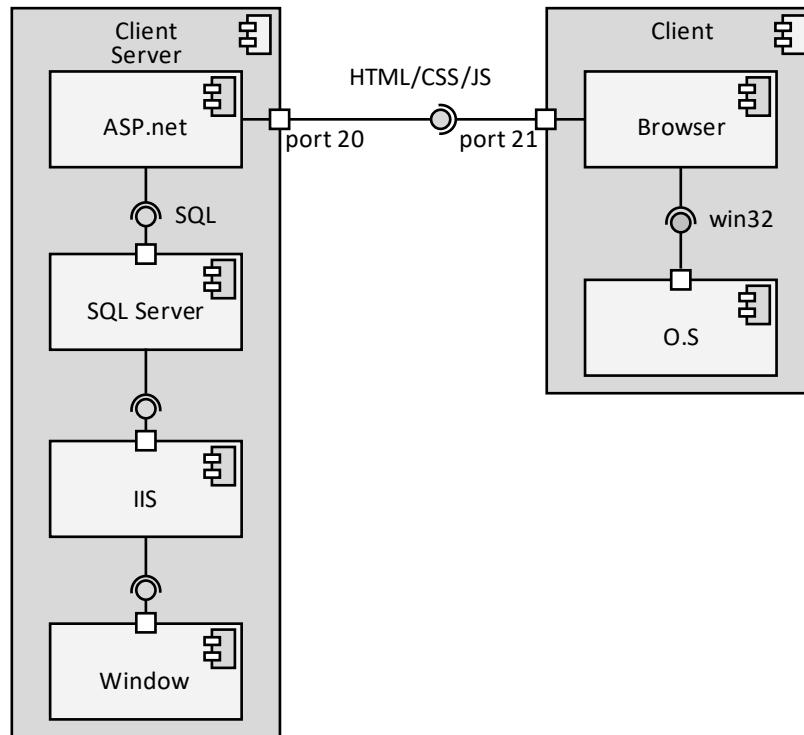
Problem

Create a component diagram describing the following scenario:

A simple web server hosts the results of a simple database query. This page is displayed on a client's browser

Solution

Two popular server stacks are LAMP (Linux, Apache, MySQL, and PhP) and WISA (Windows, IIS, SQL Server, and ASP.net). We will use WISA.



Examples 40.2: Peer-to-Peer

This example will demonstrate how one plays music through his or her car audio system.

Problem

Create a component diagram describing the following scenario:

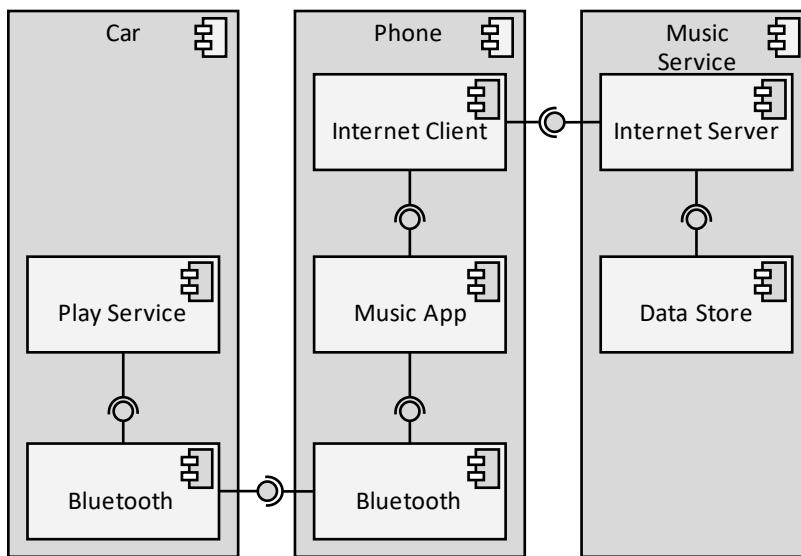
A user is streaming music to her phone. The phone then connects to the car's audio through a Bluetooth connection

Solution

There are three main elements to this design: the car, phone, and music service. Each of which has several subcomponents. The car has the Bluetooth component, able to work both with the music playback feature and with a variety of other services.

The phone also has multiple subcomponents, including a Bluetooth component and an internet client. Notice that each application on the phone is a separate component.

Finally, the music service itself can have several components, including the interface with the internet as well as the data store keeping all the songs.



Exercises

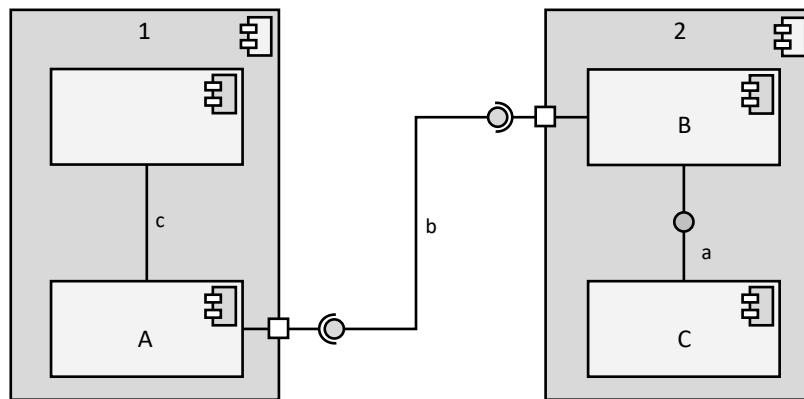
Exercise 40.1: Parts of a Component Diagram

Name the component diagram symbol.

Symbol	Symbol Name
—○—	
□—	
—○—	
■■■	
—○—	
□	

Exercises 40.2: Component Diagram Errors

Identify the errors in the following component diagram:



Exercises 40.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
A required interface is a service offered to another component	
Component diagrams are most similar to structure charts	
Component diagrams represent how data is stored in the system	
A component can contain subcomponents	
Component diagrams are only for high-level system design	
An API is a port	

Exercises 40.4: Scenarios and Tools

For each of the following scenarios, select the best viewpoint for the job.

Scenario	Tool
Represent functions and how they call each other.	
Represent a complex decision tree.	
Represent the attributes and operations of a single class.	
Represent how data flows through a system.	
Represent an algorithm including loops and equations.	
Represent the pieces of a large system design.	
Represent how many classes relate to each other.	
Describe the states in a system.	

Problems

Problem 40.1: LAMP

Consider the following scenario:

A small business has recently decided to get on the web and host a website. To accomplish this, they are using the LAMP architecture.

Research the LAMP (Linux, Apache, MySQL, PHP) web services stack. Represent it with a component diagram.

Problem 40.2: CORBA

Consider the following scenario:

A large automotive company has decided to create a persistent link between their new cars and the company server. To accomplish this, they are using CORBA.

Research CORBA (Common Object Request Broker Architecture) defined by the Object Management Group (OMG). Represent it with a component diagram.

Problem 40.3: MEAN

Consider the following scenario:

A newspaper has decided to modernize their website with a collection of interactive features. To accomplish this, they have decided to use the MEAN stack.

Research the MEAN (MongoDB, Express.js, Angular JS, and Node.js) web client stack. Represent it with a component diagram.

Problem 40.4: GLUT

Consider the following scenario:

I have decided to write a simple game for my young nephew. I will start with the classic arcade game Pong and then modify it to make it more enjoyable for him. To do this, I will use the GLUT graphics library on the Macintosh system.

Research the GLUT library. Represent it with a component diagram.

Problem 40.5: ATM

Create a component diagram describing the following scenario:

An ATM interfaces with a bank. The ATM system needs to interface with the money dispensing subsystem, the user interface, and the logging subsystem. The bank has a web portal, an account management subsystem, a database containing all the account transactions, and a logging subsystem.

Problem 40.6: Ticket Sales

Create a component diagram describing the following scenario:

I would like to create a service to sell tickets for the local high school. My system will have two user interfaces. The first is a website based on HTML, CSS, JavaScript, and Node.JS. The second is an iOS mobile application built on SpotLight, SpringBoard, Cocoa touch, and the Core Services. The system will also have a server using Python and MySQL.

Problem 40.7: Bike Shop

Create a component diagram describing the following scenario:

I own a small bike shop and would like to sell my products online. To do this, I need an accounting system to handle money, a warehouse system to handle inventory, a web store, an authentication system, and finally a user interface. The accounting system needs to keep track of the orders, interface with the bank, and create logs for a variety of purposes. The warehouse system needs to maintain a database of the inventory. The web store needs to interface with the warehouse to present to the user what is available, maintain the shopping cart, interface with the authentication system, and work with the accounting system to handle money. Finally, the authentication system needs to manage the customer list and give special privileges to the administrators.

Challenges

Challenge 40.1: Video Game

Create a component diagram of the 3D video game we have been using as an example this entire textbook. A few design considerations you may need to work out are these:

1. What graphics library are you going to use?
2. How will you separate the game engine from the logic or rules of the game so variations of the game can be easily made?
3. How will levels, ships, and other game entities be represented in the system in such a way that new versions can be made with minimal change to the system?

Challenge 40.2: List Application

Create a component diagram of the distributed list mobile application that we have been using as an example this entire textbook. A few design considerations you may need to work out are these:

1. How will one user share lists across multiple devices?
2. How will multiple users share lists?
3. What technology can you build this on that has the smallest amount of dependencies with the system on which it runs?

Challenge 40.3: Personal Finance Software

Create a component diagram of the personal finance software package that we have been using as an example this entire textbook. A few design considerations you may need to work out are these:

1. How will the user back up important financial data?
2. How will the user communicate with his or her financial institutions?
3. How will the user view his or her data across multiple devices?

Design Description

A design description is a document describing in detail the design of a software system, subsystem, or feature.

Front Matter	1
Status	1
Scope	1
Authorship	1
References.....	1
Change History.....	1
Stakeholders.....	2
Viewpoints	2
Design Views	3
System Overview	4
Class Structure	7
Game Class.....	10
Position and Velocity Classes.....	13
Score Class	15
Inertia Class.....	17
Ship Class.....	20
Saucer Class	23
Spaceship Class	25
Rock Class	27
Big Rock Class.....	30
Medium Rock Class	32
Small Rock Class.....	34
Bullet Class.....	36
DFD	39
Game Engine Structure Chart.....	43
Hit Detection	45
Rock Splitting	46
Rendering.....	47
Back Matter	50
Glossary	51
Abbreviations.....	60
Tractability Matrix	65

Figure 4.1.1:
Composition of an SDD

describe various parts of the software development process. Winston Royce, an early pioneer into the discipline of software engineering, was the first to identify these:

Document	Purpose
Requirements (SRS)	The client's needs, what the software needs to do to be considered successful.
Interface Design	Who is to use the system, how to process the input, and how all the output features are to behave?
Design Description (SDD)	How the software will be built, including overall system design, data storage strategies, and algorithms.
Test Plan	All the test cases and how the tests will be conducted (discussed in Chapter 42 Quality: V-Model).
Operating Instructions	Instructions to the user on how to use the system. This is the only document the user sees.

These documents work together to describe the system from a variety of perspectives. Though the architect, software developer, and programmer will work with versions of all these documents on a typical project, it is the SDD that occupies the majority of his or her attention.

In the early days of software development, it was common to require a system to be completely designed before any code was written. These huge and vastly complex documents were called Software Design Descriptions (SDD). Many a stout-hearted developer would cower in fear when an update to the SDD was announced!

With the advent of Agile development methodologies and similar practices, it is increasingly rare to see clients require complete SDDs. However, the need to be able to document or design large systems remains.

The SDD is a large document consisting of a collection of views describing the system from a variety of perspectives. If the details are sufficiently worked out and the views are well described, then any reasonably competent development team should be able to turn the SDD into working code. Well, this is the goal anyway!

The SDD is one of a collection of documents designed to

Design Description Sections

The purpose of the SDD is to accurately and concisely communicate design ideas

The purpose of the SDD is to accurately and concisely communicate design ideas to the various project stakeholders. If any essential components are missing or if any extraneous content exists, then the risk exists that the SDD will not fulfill this purpose. It is therefore necessary that every part of the SDD serves a purpose and communicates the designer's intent.

The SDD consists of three sections: the front matter, the design views, and the back matter. The front matter is rarely more than a couple pages in length, the design views constitute the vast majority of the document, and the back matter is only utilized on a case-by-case basis.

Front Matter

The purpose of the front matter is to inform any reader what is to be built, who sponsored the project, how to know if the project is successful, and any related work that may impact the project. Perhaps it is best put this way: if an engineer finds a printed SDD in the bottom of a filing cabinet, she should be able to figure out what it is for based only on the front matter.

These are the major parts of the front matter and the questions they are designed to answer:

Part	Questions that are answered
Date and Status	Is this the latest copy? How close it is to being finished?
Scope	What exactly is "the system?" What is covered in the SDD and, more importantly, what is not?
Authorship	Who do I contact if I have questions?
References	What other documents will I need to read to understand this one? This usually includes requirement specifications (SRS), test documents, user interface, etc.
Change History	What has changed since the last copy that I read?
Stakeholders	Who should I contact if I would like to suggest a change to the design? What considerations should I take into account?
Viewpoints	What design languages will be used to describe the design? For example, if Entity Relationship Diagrams are used in the SDD, you will need to provide a link to a source describing this language so an unfamiliar reader can understand what is written.

Readers of the SDD are likely to ask each of these questions. It is important to not only fill these sections out, but more importantly, make sure that the answers to these questions are clearly stated in the front matter.

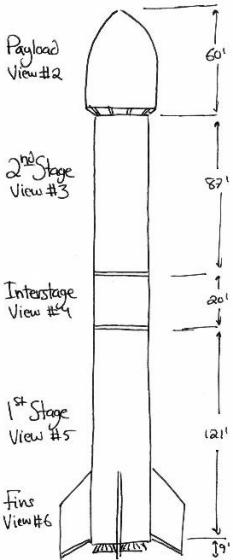


Figure 41.2:
View #1: Holistic overview

Design Views

The design views section represents the heart of the design description. This section demands the lion's share of the pages in the document and the lion's share of the effort to produce. The design views section is a collection of views, each of which is a perspective on the system. Perhaps this is best explained by analogy.

If one were to describe a design for a rocket ship, then the first view would be a drawing of the overall layout of the rocket. This view (we will call it View #1) is a schematic, including the dimensions of the rocket and the major components. None of these components would be described in detail in View #1, but each will be labeled with sub views (#2-6) providing more detail. The reader of this document will start with View #1 and, wanting to know more about the main engines, will then turn to View #5 representing the first stage.

View #5, covering the first stage, is also a schematic. It will not include anything about the rest of the design but will describe the first stage in much more detail than View #1. This view will show several subcomponents, such as the engines (View #21), the fuel tank (View #25), the oxygen tank (View #26), the landing legs (View #41), and the coupling mechanism attaching the first stage to the second stage (View #45). If the reader has any questions about the dimensions of the first stage, the components that go in the first stage, or how the components interface with each other, then View #5 should address all these questions. Our reader is interested in the first stage, but really wants to know more about the engines. Thus, he navigates to View #21.

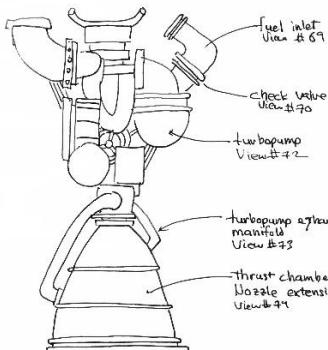


Figure 41.4:
View #21: Engine

View #21 illustrates the engine module of the first stage. It includes such details as the number and configuration of each engine (View #65), the plumbing configuration connecting the engines to the fuel and oxygen tanks (View #87), and the thrust vectoring subsystem (View #102). Of these, our reader is most interested in the plumbing configuration. For that, he navigates to View #87.

View #87 is different than View #1, #7, or #21. The first three were schematics, describing the layout of the various components. For plumbing, a different viewpoint is needed. Here, abstract symbols representing pumps, valves, pipes, and regulators are used. Those familiar with this notation will easily understand this view, but those that do not will have to reference the plumbing viewpoint document presented in the front matter.

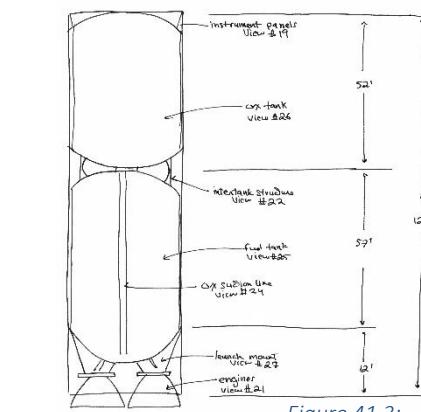


Figure 41.3:
View #5: First Stage

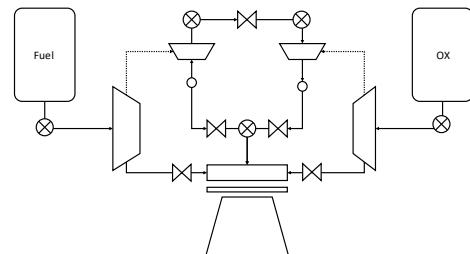


Figure 41.5:
View #87: Engine plumbing

Describing software systems is much like describing a physical rocket. You start at the broadest possible view of the system (usually a component diagram) and zoom in to the various subsystems using different views. Just like different viewpoints are required to describe a rocket (schematic, plumbing, wiring, etc.), it is often necessary to use different viewpoints to describe a software system. Each view in the design views section consists of a design representation and a design description.

Design Representation

The design representation component of a view is the actual diagram or figure needed to express a design idea. Examples of design representations include the ones discussed in this text: pseudocode, flowchart, structure chart, data flow diagram, class diagram, and component diagram. Aside from these, several others are often needed to express design ideas. The most common design representations include the following:

Representation	Design ideas
Flowchart	Algorithm design, specifically for complex decisions
Pseudocode	Algorithm design, specifically for loops and algorithms
Structure Chart	Modularization design, how functions call each other
Data Flow Diagram	The path information travels through the system
Class Diagram	Object-oriented design, the classes of the system
Component Diagram	The various subsystems and how they connect
Entity Relationship Diagram	How databases are configured in the system
State Diagram	The states of the system and how transitions are made
Sequence Diagram	Captures the order of interactions
Timing Diagram	Captures the timing of events
Circuit Diagram	Represents electrical circuits
Schema	Describes the layout of XML or JSON data
File Format	The layout of data in a file

The design representation is the actual diagram or figure needed to express a design idea

Each of the above design representation languages has an associated syntax or set of rules governing how it is used. This syntax could include the meaning of the symbols but may also describe how the symbols are used. The set of rules and syntax of a given design representation is called the viewpoint. Every design representation needs to honor the viewpoint to which it is ascribed.

Design Information

While the design representation commands most of the effort in creating a design view, more information is needed. This information is usually placed in a table immediately below the design view and helps explain various parts of it. The most common components to a design description include:

Part	Questions that are answered
View name and number	How can I unambiguously reference this view from anywhere in the SDD?
Design concern	What stakeholder need is met by this view? Usually this is completed with a list of requirements or user stories that this view is meant to address.
Elements	If any component of the design representation is not completely obvious, then a detailed description of that design element should be presented here.
Viewpoint	What does this specific design representation symbol mean? The design description should include a link or reference to the viewpoint.
References	What other views reference this view? What else do I need to look at to understand this view?

How do you know if your design description is complete? The answer to this is another question: if a developer was directed to implement this view, would he or she be able to discover all the required knowledge to be able to build it? In other words, all the questions a developer may have about this view should be answered in the design description. It will either directly state the answer or direct the developer to another place in the SDD which contains the answer.

Back Matter

The end of the SDD often contains back matter. This includes appendices, a glossary, a list of abbreviations, or anything else that may be needed to explain the intent of the design. One common thing to add here is a requirement traceability matrix. This is a table mapping the requirements of the system with the design views.

A requirement traceability matrix is shown as a grid. The columns represent requirements and the rows represent design views. Handwritten annotations provide context for the matrix:

- An arrow points to the column header '3.4.1 Perf.' with the text "Requirements are listed on the column header".
- An arrow points to the row header '#1 System View' with the text "Views are listed on the row header".
- An annotation on the right side states: "If a box is checked, then the view helps meet the requirement".

	3.4.1 Perf.						
	3.3.3 Errors						
	3.3.2 Tutorial						
	3.3.1 HRE						
	3.2.5 Role						
	3.2.4 User						
	3.2.3 Reject						
	3.2.2 Logout						
	3.2.1 Login						
	3.1.5 2-way						
	3.1.4 Lab Duty						
	3.1.3 Lab Assis.						
	3.1.2 HRE						
	3.1.1 Status						
#1 System View							
#2 Server							
#3 Feedback Class							
#4 Assignment Class							
#5 User Class							
#6 Database Server							

Figure 41.6:
Requirements traceability
matrix

Creating a Design Description

For some, creating a design description (often constituting hundreds of pages) seems like a daunting task. Questions such as “where does one begin?” and “why can I not just write the code?” come to mind. Others see the design description as an invaluable design tool, guiding the developer through the design process. How can we see this as a benefit rather than a hinderance in doing our job? Perhaps this is best described by example. Imagine one were to build an application to store, manage, and display lists (such as shopping lists, to-do lists, and even recipes).

Front Matter

Best Practice 41.1 Start with the front matter

Before one begins any project (large or small), one needs to understand the scope and what is needed. Many skip this step, thinking they know the project well enough, only later to discover that there is a miscommunication or misunderstanding. Taking a few minutes to write down your understanding of the project can clarify things and reduce the opportunity for mistakes. A few key questions include:

Key Questions

What is to be built? This is for the scope part of the front matter.

Who is on the design team? This will be the authorship part.

What are the other related technologies or projects? This is for references.

Who are the stakeholders?

Initially one does not have definitive answers to any of these questions. One starts by recording what is known and, as more knowledge is gathered, it is added to the front matter. In this way, the front matter is not written so much as it is collected over the course of the project.

Best Practice 41.2 With every major alteration to the front matter, review the entire design description

As often happens on medium- to large-scale projects, we discover that mistakes were made. Sometimes we realize that our understanding of the scope of the project is different than the client’s understanding of the scope. As one updates the front matter to reflect the latest understanding of the project, it is a best practice to review the rest of the document to identify places where these misconceptions may be manifest. Back to our list application, our first draft of the front matter is the following:

Section	Response
Scope	Mobile application, server, and web portal
Authorship	Our team, a group of six developers and one designer
References	Hosted on a web service. Link to the documentation
Stakeholders	Our company and two business partners

While writing down what is known about our project, we are prompted to carefully and individually consider each of these questions. From this, we discover that we are not only building a mobile application but also the server that sits behind it. We also realize that there are technology and business dependencies that should be consulted with each major design decision. Thinking about these things early in the design process rather than discovering them late will likely save us quite a bit of heartache.

View #1 Design Representation

Initially, design descriptions are created using the top-down design strategy (for more details, see Chapter 17 Strategy: Top-Down). One starts with the most general view of the entire system and gradually works to the more detailed and specific ones.

Best Practice 41.3 Involve the entire team in the first few views of the design

The first few views in the design description are usually created collaboratively. The manager or head designer calls a meeting with the entire team and different ideas are hashed out on a white board. Here it is important to encourage the team members to share their ideas and concerns. It might take an hour or two to work through all the options and arrive at a final decision. However, in the end, only one design is settled upon.

Best Practice 41.4 The entire team should fully understand and have buy-in on the first several views

After the team has debated many possible design possibilities, it is important that everyone is unified behind a single design. This means that many members of the team might need to forget about discarded design alternatives—even the ones that had desirable properties. It is more important that everyone is on the same page than that the team is following the best possible course. Managers and team leads often do a “road show,” visiting each member of the team individually to communicate the design, answer questions, and learn about concerns. It is worth repeating: at the end of this process, everyone must fully understand and have buy-in on the first several views.

It is more important that everyone is on the same page than that the team follows the best possible course

Best Practice 41.5 Use a component diagram or a DFD for View #1

The first view of a design description needs to be the most general view, capable of capturing the overall system architecture. In most cases, this is a component diagram (see Chapter 40 Tool: Component Diagram), though some designs are best described with a DFD (see Chapter 11 Tool: Data Flow Diagram). Here, the design team takes special care to identify the parts of the system which are mostly independent of each other. These are often called components. The design team also carefully identifies how the components talk to each other. These are called interfaces.

Best Practice 41.6 Carefully negotiate interfaces between components

If the interfaces between components are well understood, then the components themselves can be designed and built independent of the rest of the system. This serves to reduce the complexity of the overall system and simplify testing. Unfortunately, identifying and negotiating interfaces can be a very challenging task. Back to our list application, View #1 is a component diagram with a simple client-server architecture.

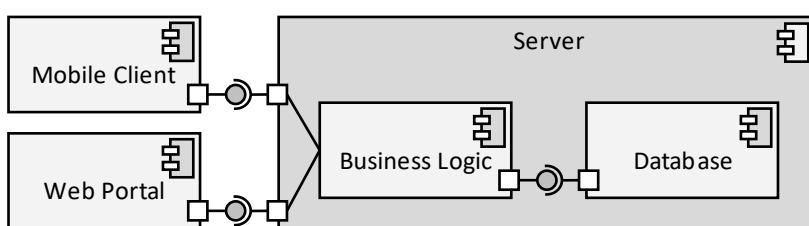


Figure 41.7:
View #1 of the list
application, a
component diagram

View #1 Design Information

Once the first view is drawn, the next step is to fill out the table associated with the view. This table, called the design information, represents all that is known about the view. The most important part of this table is the list of elements. Back to our component diagram of our list application:

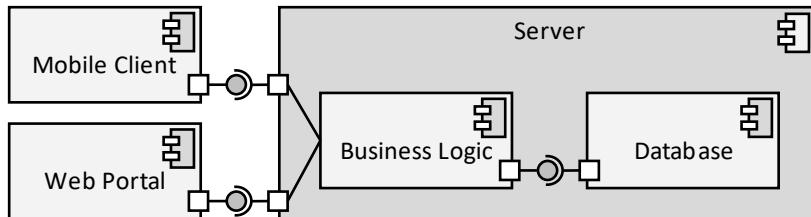


Figure 41.8:
Design representation
for View #1: a
component diagram

This component diagram does not provide enough detail for a team of software developers to implement the system as the designer may envision it. The design information table fulfills this need.

Best Practice 41.7 Use the design information table to record everything that is known about a given view

The most important part of the design information table is the enumeration of design elements described in the design representation. At all points in the design process, the design team records all that is known about each element in the elements row of the table. On the surface, it appears that there are four elements: the mobile client, the web portal, the business logic, and the database. As it turns out, there are eight elements.

With this design information table, View #1 is complete but not sufficient. All the required information is present, but we do not have enough information to build this system. We therefore need to dive deeper into each element to describe it more completely.

Name	# 1 System View
Purpose	Describe overall system view
Description	Basic client-server architecture where there are two flavors of clients: mobile and web portal
Requirements	1.0-2.9
Elements	Mobile Client: user-facing view that displays server information, accepts input to send to server, and caches data. Web Portal: administrator view allowing for tuning of the database and adjusting business logic Mobile-Business: HTTPS-JSON interchange Web-Business: HTTPS-JSON interchange Server: All hosted services hosted on AWS Business Logic: Access control, account management, etc. Database: SQL database containing all client data Business-SQL: SQL interface on secure network
Referenced By	
Viewpoint	Component Diagram

All of the elements
in the design
representation are
described to the best
of our knowledge

Figure 41.9:
Design information for
View #1

Subsequent Views

In our list mobile application example, the business logic component is briefly described but leaves much to the imagination. To clarify things, we will replace the brief description of “access control, account management, etc.” with a view number:

Name	# 1 System View
Purpose	Describe overall system view
Description	Basic client-server architecture where there are two flavors of clients: mobile and web portal
Requirements	1.0-2.9
Elements	Mobile Client: user-facing view that displays server information, accepts input to send to server, and caches data. Web Portal: administrator view allowing for tuning of the database and adjusting business logic Mobile-Business: HTTPS-JSON interchange Web-Business: HTTPS-JSON interchange Server: All hosted services hosted on AWS Business Logic: #7 Database: SQL database containing all client data Business-SQL: SQL interface on secure network
Referenced By	
Viewpoint	Component Diagram

Name	# 7 Business Logic
Purpose	Contain all the business logic of the application
Description	Access control, account management, etc.
Requirements	
Elements	
Referenced By	#1
Viewpoint	

Figure 41.10:
One view becomes two

At any point in the design process, we can elaborate on an aspect of the design by turning an element in one view to a view of its own. As this is done, it is necessary to choose an appropriate viewpoint to describe design ideas, work out different design alternatives, and explain the elements of the new view as clearly as possible.

One can think of a design description as a tree. At the root is View #1, representing the entirety of the design though at a very high level. Branching from View #1 are all the views that it directly represents. Thus, one can follow a series of branches from View #1 to every other view in the document. Note that the relationship between the views is not a true tree. One view (like the database component from View #1) may be referenced by several other views. Nevertheless, it is still instructive to see these views as related as a tree.

Large Design Documents

A small project consisting of a few thousand lines of code can be represented with a dozen views or less. These may consist of a structure chart, a few class diagrams, a half dozen flowcharts or pseudocode functions, and perhaps a DFD. A moderate project can easily consist of a couple hundred views whereas a large project may be numbered in the thousands. How does one manage a design description of that scope and size?

Best Practice 41.8 Use a multilevel numbering system to name views

When the number of views is greater than a couple dozen, it is useful to use a multilevel numbering system. Back to our list application, one may choose to use a system like the following:

Number	Meaning
1	Root view
2.x	Mobile client
3.x	Web portal
4.x	Server

From here, one can subdivide the numbering further:

Number	Meaning
2.0	Mobile client overview
2.1.x	Mobile authentication subsystem
2.2.x	Mobile list display engine
2.3.x	Mobile list-selection interface
2.4.x	Mobile list management subsystem

A numbering system like this ensures that related views are next to each other in the large document. Another benefit is that each component can add or remove views without impacting the number of other views in the document.

Best Practice 41.9 Place each view on its own page

When possible the design representation and the design information components of a single view should be on the same page in the design document. This increases readability because the reader of the document can view all the necessary information at a glance. Regardless of whether the design description is ever printed, the layout of the document should be done in such a way as to maximize readability for the intended audience.

Best Practice 41.10 Use headings and links intentionally

A design description is not read linearly as one would read a novel. Instead, one constantly jumps between views to get more information or to understand the context in which a design element resides. Most word processors will bookmark headings and create a document map to facilitate intra-document navigation. Leveraging these navigation tools is essential to making large documents such as a design description readable.

Using a Design Description

Even in this modern age of Agile software development, many teams find the need to use design descriptions. The most common reasons are these: tracking the status of a large project, effort estimation, single point of communication, and satisfying contractual or legal needs.

Status Reporting

Even with small projects involving a single developer, it is difficult to accurately ascertain the progress of a project. Most engineers tend to be optimistic, believing that everything is figured out and that all will go well. As most experienced engineers know, this is seldom the case!

The SDD enables reviews to find holes, inconsistencies, and problem areas

The process of capturing the design ideas from the entire development team facilitates determination of the status of the project. It enables the manager or a team of reviewers to systematically and thoroughly review the entire design for the purpose of finding holes (details which have yet to be worked out), inconsistencies (two or more parts of the design which make different assumptions or will otherwise not work together), and problem areas (areas of significant technical risk).

Even teams with significant technical expertise and experience often uncover surprises when the entire design is laid out in the SDD format.

Communication and Collaboration

In many software development scenarios, all the programmers are in the same physical location. If one programmer has a question, then a short walk yields the answer. Unfortunately, this is not always the case.

In an increasingly global economy, it is not uncommon to have to coordinate with other teams separated by vast distances and over significant periods of time. In scenarios such as this, many teams turn to the SDD as a collaboration tool. The SDD should represent the current thinking of the design as well as the system as it is currently built. Any member of the team should be able to look at the SDD to answer any design questions.

The SDD is the best place for members of the design team to find answers to all system design questions

Contractual or Legal Requirements

The final usage of the SDD in the software design process is to fulfill contractual or legal requirements. If a team were to outsource the creation of a mobile application to another entity halfway around the world, the SDD often serves as a central figure in the process.

Perhaps this is best explained by analogy. Though Apple designs their smart phones in California, they are built overseas. Apple sends the manufacturer a hardware equivalent of an SDD. As long as the manufacturer's resulting products match the SDD, then they will have fulfilled their contractual obligations. The same is true with software; when a part of a project is outsourced to a subsidiary, the statement of work is frequently in the form of an SDD.

Effort Estimation

Every software engineer will be asked to make estimates how long it will take to complete a given task. Sometimes these estimates are rough and off-the-cuff, used by managers and designers to weigh design decisions. Sometimes these estimates need to be precise and accurate—used for resource allocation and budgeting purposes. How does one make accurate estimations? The answer lies in the SDD.

If you are asked to estimate the time it will take to build a large project and you do not have the design worked out, then your estimation is really no better than a guess. You may have a general idea of what needs to be built, but nothing more than that. In effect, you are estimating View #1. Generally, the more detailed the design and the smaller the scope of the component you are estimating, the more accurate the resulting estimation will be.

The more detailed the design and the smaller the scope of the component you are estimating, the more accurate the resulting estimation will be

Effort estimation goes hand in hand with the design process. It is common to attach estimation numbers to individual views as the design process progresses. For example, a developer may put a 1,000-hour (half year) estimation on view #1. This view might reference several sub views. If the summation of the sub view estimates yields 800 hours and if it takes an additional 40 hours to integrate the sub views, then the estimation on View #1 would be refined to 840 hours. Each time the estimate on a view is refined, then the estimates on all the views referencing the newly estimated one needs to be refined as well. For example, consider the scenario where View #1 references View #7 which references #21 which references #87. If we add 20 hours to the estimate for #87, then the cost of #21 increases by 20 hours, as does #7, as does #1. Thus, View #1's estimation always represents the overall cost of the system.

Generally, the effort estimate for a single task should be no greater than four hours

How does one know when the estimation process is finished? The general rule is this: when a single task takes longer than half a day (4 hours) for a developer to complete, then it is too big. Thus, the cost of any single element in an SDD should be no larger than 4 hours. If it is, then that element should probably be subdivided into another more detailed view.

Examples

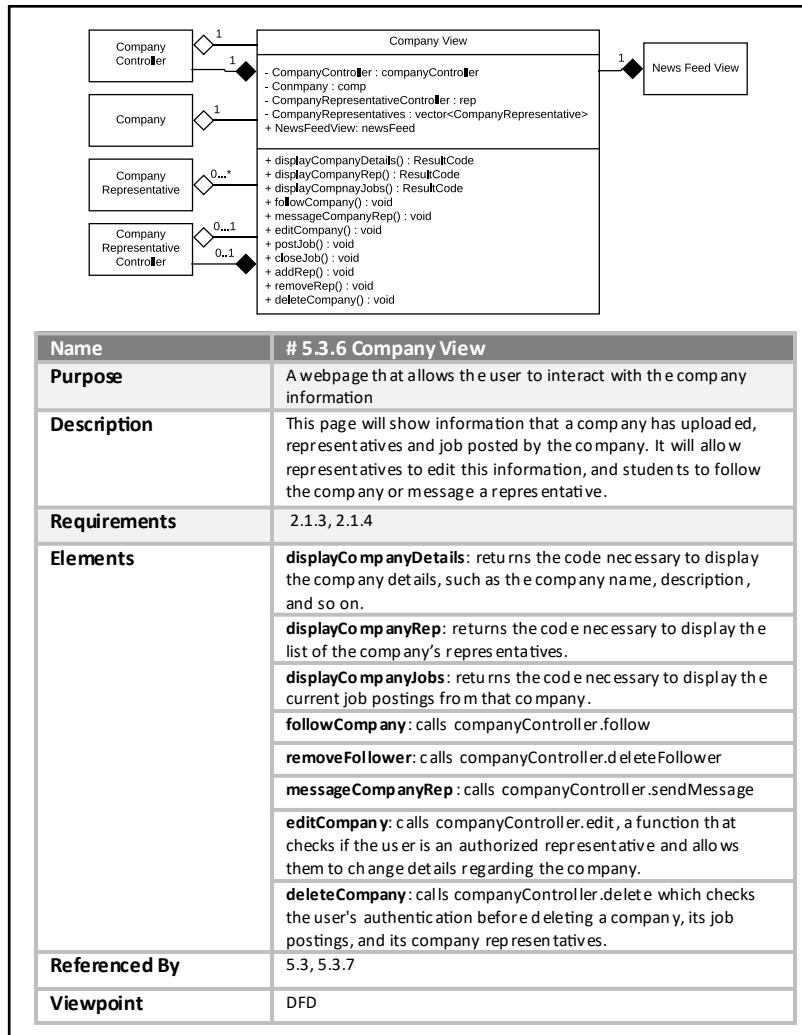
Examples 41.1: Class Diagram

This first example will demonstrate a class diagram viewpoint.

Problem

Create a view describing a class representing one company in a job-finding networking application.

Solution



Notice that the “referenced-by” component in the design description enables the reader to know which views utilize this one. Also, all the elements are fully described in this view; there are no child views from 5.3.6.

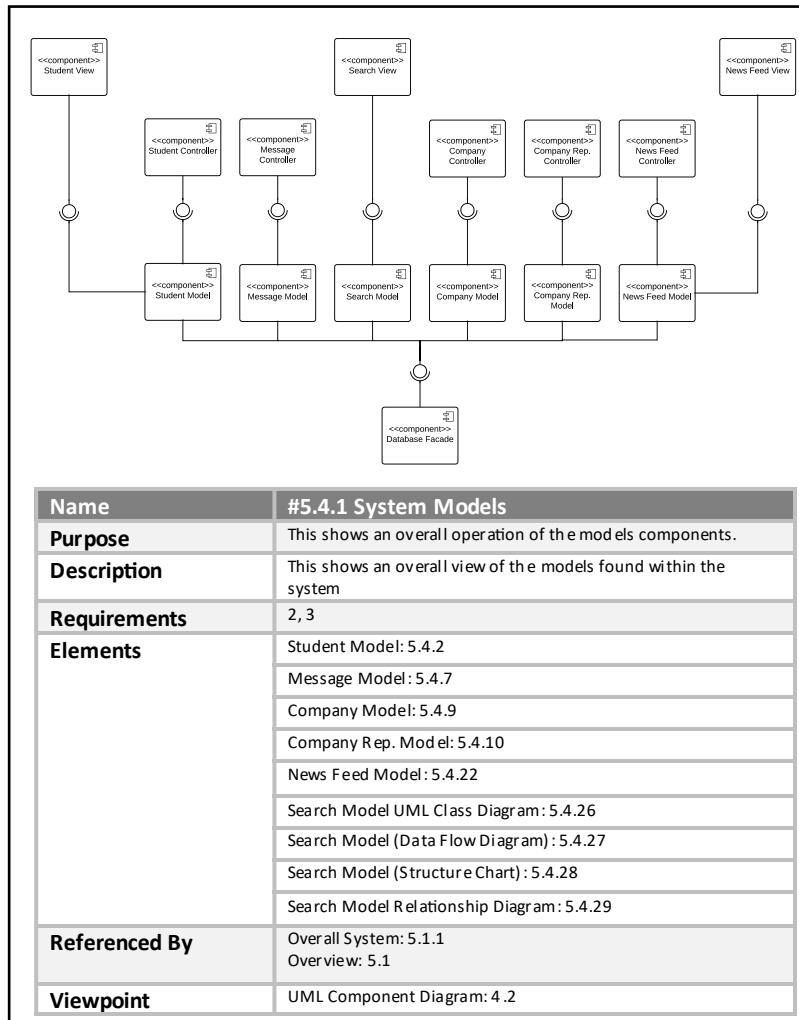
Examples 41.2: Component Diagram

This second example will demonstrate a component diagram view.

Problem

Create a view representing the model part of an MVC server for job-finding networking application.

Solution



None of the elements described in this high-level view are explained in any detail here. Instead, sub views contain all the details necessary to understand them.

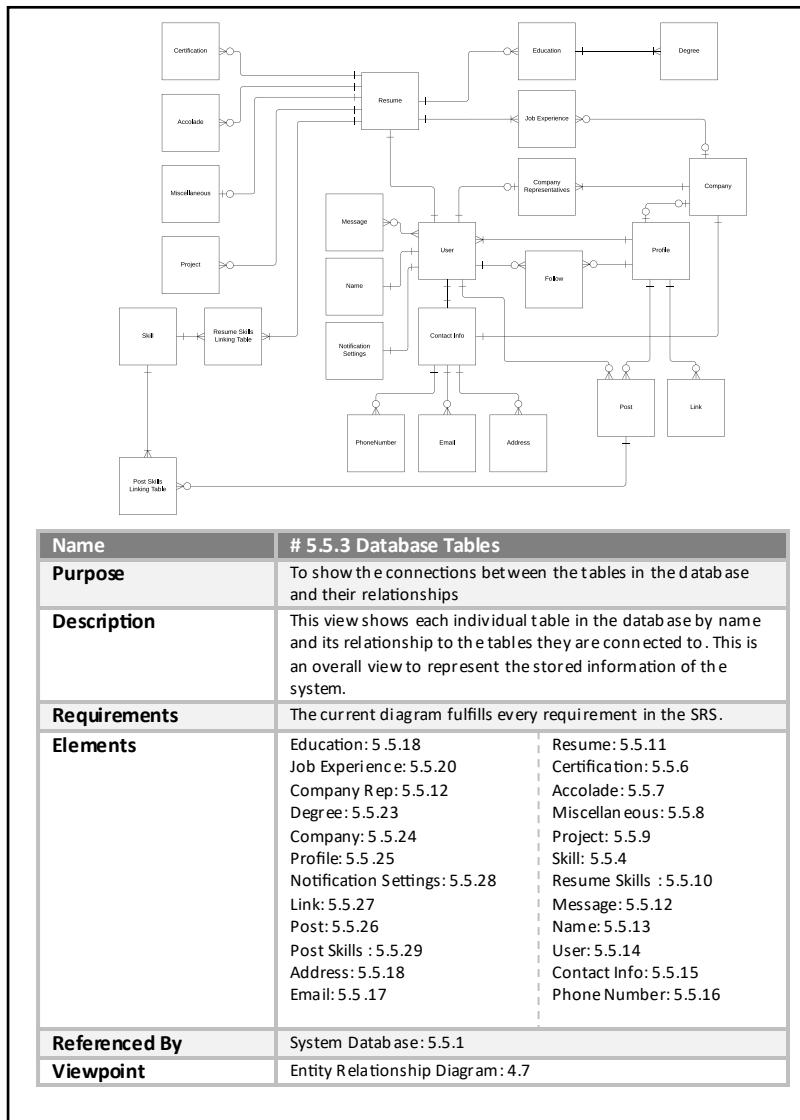
Examples 41.3: Entity Relationship Diagram

This third example will demonstrate an entity relationship diagram view.

Problem

Create a view representing all the databases involved in a job-finding networking application.

Solution



This high-level view serves as a roadmap to all the database tables used in the application. There are a total of 24 individual tables referenced here, each one in its own view.

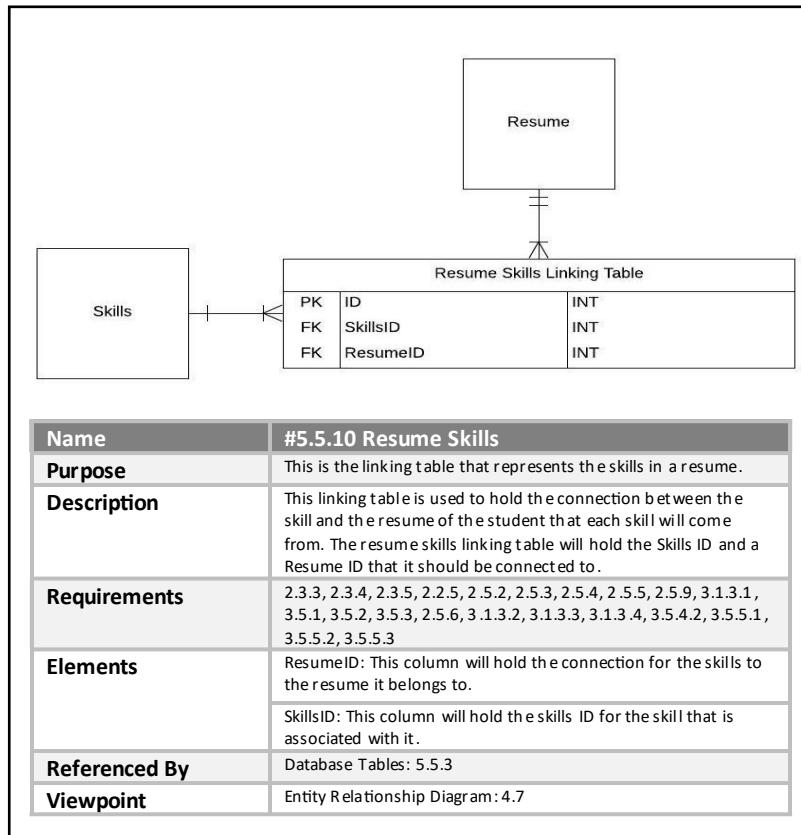
Examples 41.4: Entity Relationship Diagram

This forth example will demonstrate an entity relationship diagram view.

Problem

Create a view representing the `Resume Skills` linking table in a job-finding networking application.

Solution



This leaf-level view depicts only a single table. It also directs the reader to the holistic view depicting all the tables in the application (View #5.3.3) as well as the tables which are directly related to this table (`Skills` and `Resume`)

Exercises

Exercises 41.1: Part of the SDD

Identify the section of the SDD where one would look to get an answer to the following questions:

Question	Section
What has changed since the last copy I read?	
What exactly is “the system?”	
What other documents must I read to understand this SDD?	
Is this the latest copy of the SDD?	
What design concern is this particular view meant to address?	
What design languages will be used in the SDD?	
What other views reference this particular view?	
Who do I contact if I have any questions?	

Exercises 41.1: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
The first view (#1) is a holistic view of the system.	
The front matter is the biggest part of the SDD.	
The SDD is useful in the effort estimation process.	
The tractability matrix goes in the front matter.	
Any effort estimation greater than a half a day is too big.	

Exercises 41.3: Viewpoints

Select an appropriate viewpoint for each of the following scenarios:

Scenario	Viewpoint
Represent database tables	
Represent how data move through the system	
Represent a decision-heavy algorithm	
Represent how classes depend on each other	
Represent the states of a system and how transitions are made	
Represent a loop and algorithm-heavy function	
Represent an XML schema	
Represent how functions call each other	
Represent electrical circuits	
Represent the order of interactions and actions in a system	
Represent the overall architecture of a system	
Represent the file format of an application	
Represent the timing of events	
Represent the composition of a class	

Problems

Problem 41.1: Projectile View

Create a view to represent a bullet in a 3D video game. This view will be labeled “#47 Projectile” and will be a UML Class Diagram viewpoint. Use Example 30.4 as a starting point. Note that it will refer to several subsequent views. Assume the following views are already created:

- #31 Flying Object: The base class for **Projectile**
- #56 Bullet: A derived class from **Projectile**
- #57 Dart: A derived class from **Projectile**
- #58 Guided Missile: A derived class from **Projectile**

Problem 41.2: Checking Account View

Create a view to represent a checking account class. This view will be labeled “#21 Checking Account” and will be a UML Class Diagram viewpoint. Use Figure 30.17 as a starting point. Note that it will refer to several subsequent views:

- #15 Accounts: A broad description of all the accounts in the system.
- #24 Transaction: One transaction for a given account

Problem 41.3: Days In Month View

Create a view to represent a function **numDaysInMonth()** function which is part of the calculate budget component of a personal finance software package. This view will be labeled “#19 Days in Month.” Note that it will refer to several subsequent views:

- #21 **isLeapYear**: Pseudocode for the **isLeapYear()** function from Problem 01.1.
- #10 **calculateBudget**: A structure chart describing how to compute a budget that contains the **numDaysInMonth()** function.

Problem 41.4: Sort List View

Create a view to represent a function to sort a grocery list. This view will be labeled “#39 Sort List” and will be a structure chart viewpoint. It will accept user input, reorder the list, and present the results to the user. Note that it will refer to several subsequent views:

- #41 **Sort**: Pseudocode to sort a list using the merge sort
- #42 **AcceptInput**: Pseudocode to receive user input
- #43 **UpdateList**: Pseudocode to display the grocery list

Challenges

Challenge 41.1: Video Game

Create a Design Description of the 3D video game we have been using as an example this entire textbook.

Note: this will probably consist of about a hundred views. It will take 1 person 40 hours to complete this task, a team of 3–5 12–15 hours each to complete this task, and a team of 10–20 about 8 hours each to complete this task.

Challenge 41.2: List Application

Create a Design Description of the distributed list mobile application that we have been using as an example this entire textbook.

Note: this will probably consist of about a hundred views. It will take 1 person 40 hours to complete this task, a team of 3–5 12–15 hours each to complete this task, and a team of 10–20 about 8 hours each to complete this task.

Challenge 41.3: Personal Finance Software

Create a Design Description of the personal finance software package that we have been using as an example this entire textbook.

Note: this will probably consist of about a hundred views. It will take 1 person 80 hours to complete this task, a team of 3–5 12–15 hours each to complete this task, and a team of 10–20 about 16 hours each to complete this task.

V-Model

The V-Model is a software development life cycle model design to help teams build quality into a project.

The first widely distributed and adopted software development life cycle (SDLC) model was described by Winston Royce in 1970. This model, sometimes referred to as the waterfall model, described a multi-step approach to systematically developing a software project. In Royce's model, development would progress from high-level requirements gathering activities to coding. Only after coding is finished would testing begin.

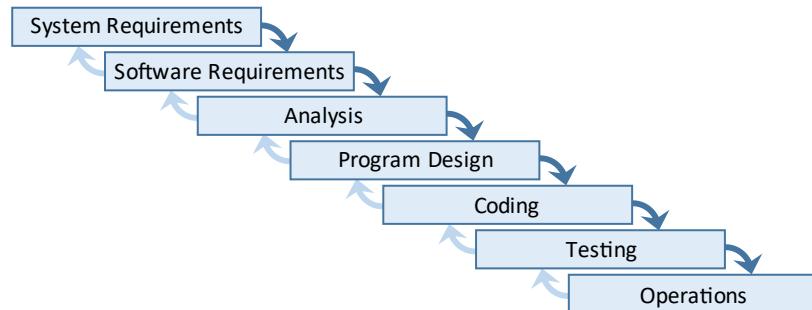


Figure 42.1:
Royce's SDLC model

Royce's model was often criticized for not adequately accounting for the difficulty of sufficiently testing complex software systems. To address this deficiency, the V-Model was developed. There is no consensus on when the V-Model was first described, though there is evidence that elements existed in the 1960's. The first formal documentation was written by Barry Boehm in 1979. The German ministry of defense independently developed the *V-Modell* where the V was not in reference to the shape of the model but rather to the German term *Vorgehensmodell* meaning "process model." Today, the V-Model is often called the systems engineering vee.

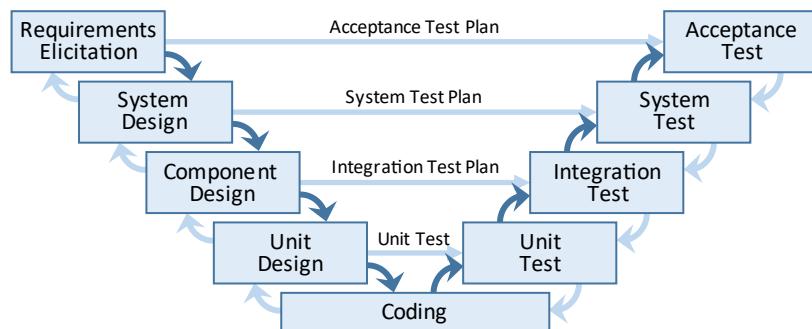


Figure 42.2:
Boehm's V-Model

The primary characteristic of the V-Model is that each development activity is paired with a testing activity. In other words, as a design feature is identified, the corresponding tests are also developed. The V-Model process flows from the upper-left corner of the V down to the bottom. Once coding is complete, then process flows upward towards the top-right corner. The process concludes when all the acceptance tests pass.

Steps of the V-Model

The V-Model is variously described as a 9-step model or a 7-step model. The main difference is the existence of a component design and an integration test layer. Large and complex systems need these layers where simple projects may not. These steps are requirement elicitation, system design, component design, unit design, coding, unit test, integration test, system test, and acceptance test.

Requirements Elicitation Step

The requirements elicitation process is the initial step of software development activity where the needs of the various stakeholders are elicited and documented. Requirements elicitation is an iterative and ongoing process that is often not finished until the project itself is complete. The requirements elicitation process has two outcomes: the system requirements and the acceptance test plan.

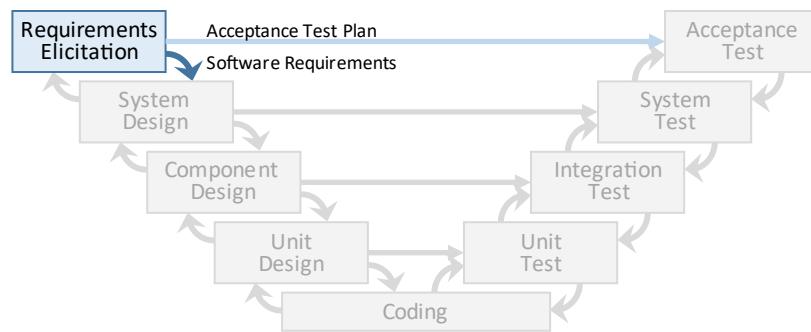


Figure 42.3:
V-Model with emphasis on
requirements elicitation

Software Requirements

Every project must have a high fidelity understanding of what the system is meant to do. There are many ways that the software requirements can be represented. Formal and contractual development environments may require a software requirements specification (SRS) to encode the requirements. Agile development methodologies use a product owner and a product backlog to represent the system requirements. Regardless of how they are represented, software requirements are an essential artifact of any software development methodology. This is because misunderstanding of the needs of the system will almost certainly result in a system that does not meet those needs.

Acceptance Test Plan

Recall from Chapter 25 Quality: Unit Test that acceptance tests, also known as customer tests, are quality checks designed to inform stakeholders whether the system is fit for use. The V-Model asks quality assurance (QA) engineers to create a suite of acceptance tests for each requirement so it can be known if the requirement is met. This is done long before any technology decisions are made.

It is possible to move on to the system design activity when the system requirements are sufficiently understood to make system-wide decisions. A perfect understanding of all the system requirements will not be known at this initial step in the development process—it is understood that new requirements will be discovered as the project progresses. When this happens, it is often necessary to reevaluate the system design to ensure the newly discovered requirements can be accommodated.

System Design Step

A software system is one or more software entities that fulfills the needs of the stakeholders. The system design step is where the overall system design is determined as well as where the plan to test the system is outlined.

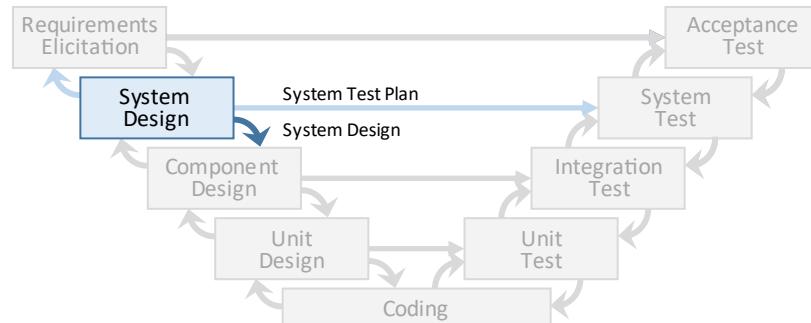


Figure 42.4:
V-Model with emphasis on
system design

System Design

The system design represents an outline of the overall system. The system is subdivided into components so individual teams can independently refine the design to higher levels of detail. Each of these components is named, the dataflow into and out of the components are identified, and the functionality of the components are described. The most common design tools used at this phase are component diagrams and DFDs. Protocols connecting components and technologies used in components are often identified in this phase. If the team utilizes a design document to record and communicate design decisions, then the system design represents the first few views at the beginning of the document. Please see Chapter 41 Tool: Design Description for more details about creating a design document.

System Test Plan

System tests are like acceptance tests with two important differences. First, acceptance tests are usually written from the user's perspective, describing how users interact with the system. System tests are written from a component's perspective, describing how one component interacts with another component. Second, the purpose of an acceptance test is to judge the overall quality of the project. The purpose of a system test is to find defects in the project. System tests often involve test harnesses, automation, and driver components designed specifically to fully exercise an individual component independently from the rest of the system. The system test plan is usually authored by a software development engineer in test (SDET), not a software developer.

From the system design step, the process can progress one of two directions: forward to the component design step or back to the requirements elicitation step. The process moves forward if the design team is sufficiently confident about the system design and that the components themselves can be designed. In other words, do the members of the development team believe that each component can be built in such a way that the system tests can pass? If the team cannot be assured that the design is tractable, the system design step may need to be continued or perhaps the requirements elicitation step may need to be repeated. This could be the result of several things: a member of the team may have discovered a previously unforeseen technical challenge, a requirement may not have been sufficiently understood, or the development cost of a component may be excessive. In almost all situations, it is better to catch problems early in the process rather than to procrastinate until a later date.

Component Design Step

A software component is a subdivision of a software system which is somewhat independent from the rest of the program. Frequently, components can be recompiled, upgraded, and replaced without affecting the rest of the system. The component design step, also known as the architecture design step, has two outputs: specifying the design of a component, and creating an integration test plan for a component.

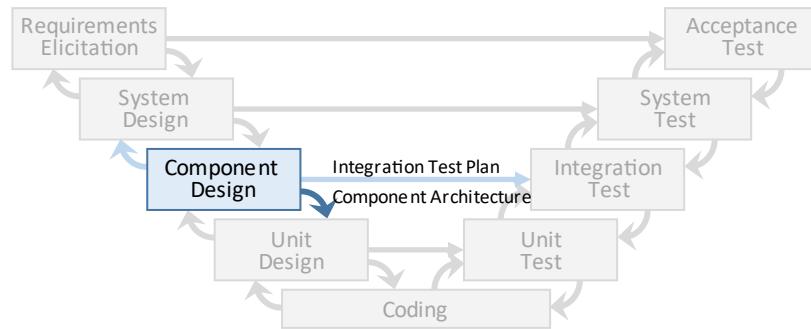


Figure 4.2.5:
V-Model with emphasis on
component design

Component Architecture Design

The component architecture design is the design of a single component in the system. This design considers how the component interacts with the rest of the system as specified in the system design, including how data flows into the component, how data leaves the component, and what type of processing the component itself performs. The component architecture design is usually expressed with class diagrams, data flow diagrams, and occasionally component diagrams. Design patterns are often used to solve problems at this level. If the team utilizes an design document to record and communicate design decisions, then the component architecture design represents most of the views in the design document.

Integration Test Plan

The integration test plan is the plan used to describe how to verify that the units within a component work together. It consists of a suite of tests for every pair of units that interact. These tests combine interacting units and carefully monitors the data transfer between them. In the following component, there are five units (A, B, C, D, and E). There are six interactions that need to be verified in the integration test plan: A-C, A-D, B-D, B-E, C-E, and D-E.

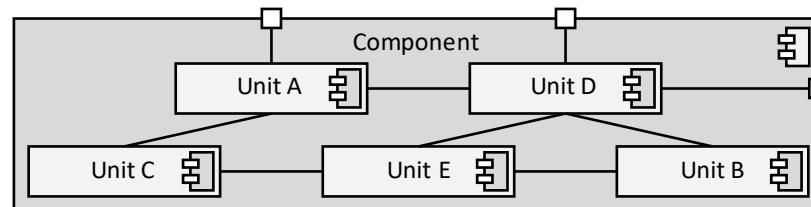


Figure 4.2.6:
Component diagram
showing six interactions to
be verified

When the component architecture design is sufficiently described, the development effort can move to the unit design step. This is the typical demarcation between a team-wide design effort and an individual development effort. However, the situation may arise when the architecture design is impossible, impractical, or difficult to verify. In those situations, the system design may need to be reconsidered.

Unit Design, Coding, and Unit Test Steps

A unit is the smallest testable part of any software system. In most cases, a unit is a single function, class, or inheritance tree. Unit design, coding, and unit tests are typically conducted by individual software developers under the direction of the component architecture design specified by a team lead.

The bottom three steps in the V-Model describe the process of designing, implementing, and verifying units. These three steps are presented together because they often happen in quick succession (often several times in a single day) and because the order in which they are completed can vary depending on the adopted development methodology.

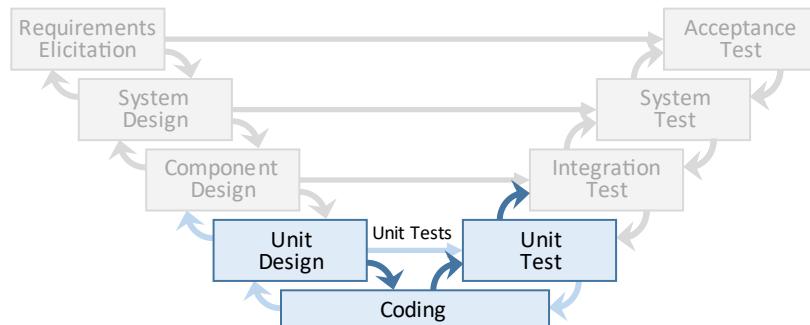


Figure 42.7:
V-Model with emphasis on
unit design and tests

In the early days of software development, the complete system, component, and unit designs would be described in their entirety before the coding phase began. Only after coding was finished would testing take place. This seldom happens in modern development teams today; the design at one level is developed only enough to work on the design at the following level. Also, testing is done as early as possible to inform the development team that the design will work. Thus, coding and testing occur early in a project, long before all the design details are fully understood.

Traditional and Test-Driven Development

The traditional way to implement a unit of code is to first design, then implement the design, and finally to run the tests. This means that unit tests are created and executed only after the code they are meant to verify is finished. This approach certainly works, but there are two problems that need to be overcome. First, it is often difficult to objectively write unit tests once the implementation has been finished. Second, the unit tests themselves are difficult to verify because they execute without error the first time they are run.

This order is reversed with the TDD methodology (please see Chapter 26 Quality: Test-Driven for more details). Here, unit tests are written and executed before coding begins. This way, the developer can verify that the unit test will catch errors as they are designed to do. TDD also encourages developers to write better unit tests because the tests are based on the functionality of the units rather than being based on their implementation.

There are two possible outcomes of the design-test-code steps of the V-Model: either a serious problem has been uncovered that cannot be handled at the unit design phase, or the unit tests pass. If a problem has been uncovered, it is often necessary to return to the component design phase to address it. This usually involves refactoring the component design so the unit design is more tractable. Of course, any significant change to the component design requires integration tests to be adjusted. When the unit tests pass, then the unit is considered done.

Integration Test Step

Integration testing begins once unit tests pass. Some integration tests can be built into the codebase like unit tests. Others are implemented in automation. Still others are to be executed manually. The goal of integration testing is to find defects for the purpose of improving the quality of a component.

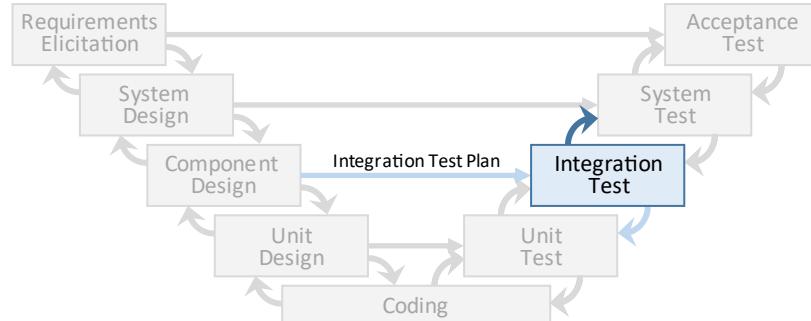


Figure 42.8:
V-Model with emphasis on
integration testing

As modules complete unit tests, they are integrated one by one. For example, consider five modules in a component (A, B, D, D, and E). The integration test plan consists of six sets of tests: A-C, A-D, B-D, B-E, C-E, and D-E.

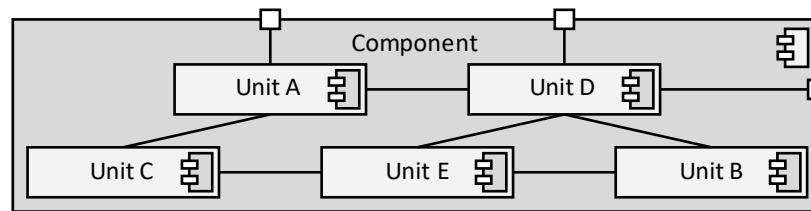


Figure 42.9:
Component diagram of a
component with five units

If these units are completed in the following order (A, C, B, D, and finally E), then the corresponding integration tests are executed in a similar order.

1. (A): If unit A is completed first, no integration testing can be done because the other modules are not yet ready.
2. (C): Once unit C passes her unit tests, then the interaction between A and C can be verified. The set of integration tests to be run is: {A-C}.
3. (B): Next B is completed. Since B does not interface directly with A or C, then no further integration testing can be done at this point. The set of possible integration tests to be run remains at {A-C}.
4. (D): Next, unit D is complete. Since D interfaces directly with A and B, then those tests can now be executed. A total of three integration tests can be run: {A-C, A-D, B-D}
5. (E): Finally, unit E is finished. Since E interfaces directly with C, B, and D, then all those integration tests can be conducted. At this point, the entire set of integration tests can be run: {A-C, A-D, B-D, B-E, C-E, D-E}.

Once a component's integration tests pass, then the component is ready for system testing. If the integration test fails, however, things can get complicated. It could be that the unit tests provided insufficient coverage and need to be augmented. It could be that the component design is insufficient and it needs to be reengineered. Regardless of the cause, it can be counterproductive to perform system tests until the component is considered finished.

System Test Step

Recall that unit tests are maintained and executed by developers. Integration tests can be maintained and executed by either developers or SDETs. System tests, on the other hand, are firmly in the domain of the SDETs. System testing focuses on the interoperability of the various components of the system. They are begun once integration testing is complete for a given component. In other words, it is not productive to determine how two components interact when the components themselves are not reliable.

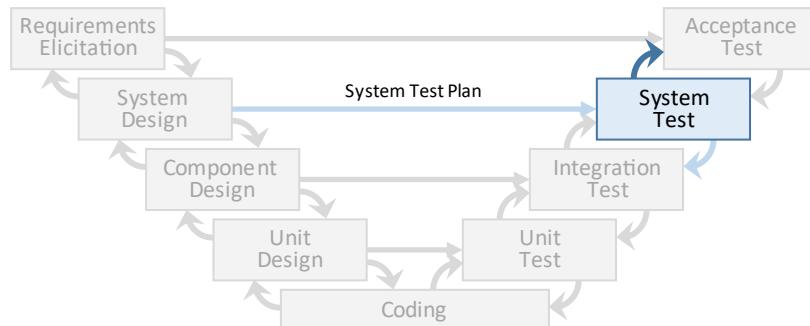


Figure 42.10:
V-Model with emphasis on
system testing

The purpose of system tests is to find defects in how the components interoperate. Because components can reside in different processes or on different physical machines, many system defects are related to timing dependencies or handling unreliable network connections. These are very difficult to detect and even more difficult to fix them.

Acceptance Test Step

The final step in the V-Model is acceptance testing. This is usually conducted by QA engineers with the goal of knowing if the system is fit for use. Acceptance testing often reveals code defects but cannot be relied upon to find all the defects in a system. Instead, acceptance testing is an indication of whether the stakeholders will find the system quality to be satisfactory.

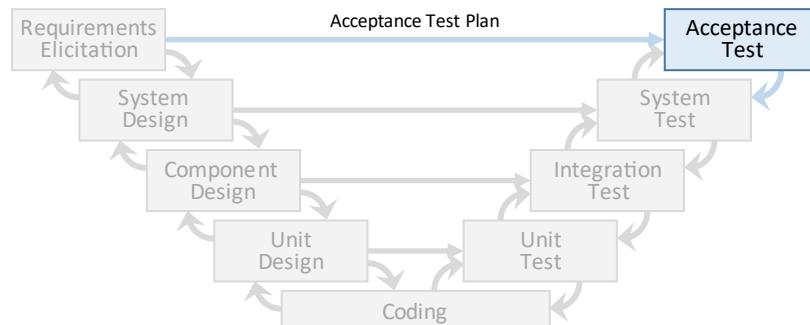


Figure 42.11:
V-Model with emphasis on
acceptance testing

The QA engineers spend considerable effort determining the quality level of the system before acceptance tests can be run. This is done through examining the partial working system, prototypes, and other tools. However, these techniques can only give the team a glimpse of the overall quality of the system. Assurances that the requirements are met cannot be made until the system tests pass and the acceptance tests can be run. For this reason, acceptance tests are the last step in the process and completion of the tests signals the completion of the project.

Creating a Test Plan

Just as many project teams consolidate their design ideas into a single document (called the design document or SDD), it is common to consolidate all testing plans into a single document. This is called a test plan.

Software developers have a different function in a team than QA engineers and SDETs. Since test plans are QA tools, why would a developer need to know how to create and use a test plan? The answer is this: each contributor on a software development team needs to independently function at a high level of proficiency and the various components need to work together seamlessly. If any role is not fulfilled, the entire project is in jeopardy. It is therefore the job of every member of the team to do their part, *and* to facilitate others to be successful as well. By considering the needs of SDETs and QA engineers, software developers can choose designs that are easier to verify. They can also know the types of information SDETs and QA engineers need to create comprehensive test plans. Thus, a developer with in-depth knowledge of the test plan creation process can be a better resource to the QA engineers and SDETs, helping them create a better test plan and resulting in a higher quality project.

Whether the test plan resides in a single document or if it refers to a collection of documents dispersed throughout the organization, a test plan consists of four components: the front matter, the acceptance test plan, the system test plan, and the integration test plan. Note that the unit tests are written by developers and are thus not typically part of the project's test plan.

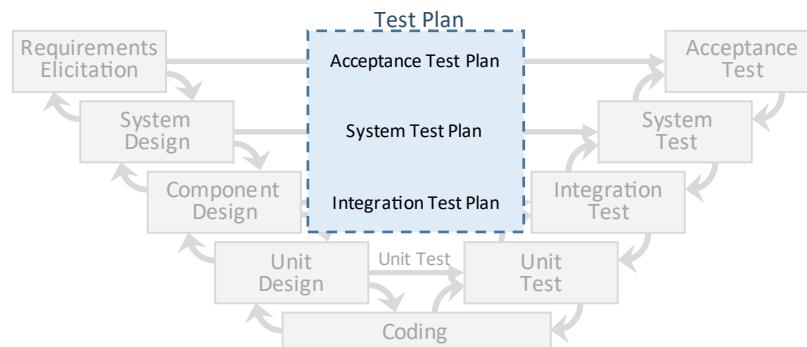


Figure 42.12:
V-Model with emphasis on
the test plan

Front Matter

As with the design document, The purpose of the front matter is to inform any reader about the system to be tested and where to go for more information. The questions the front matter is meant to address are the following:

- Is this the latest copy? How close is it to being finished?
- What is the scope? What is covered and not covered in this test plan?
- Who do I contact if I have questions?
- What other documents do I need to read to understand this plan?

Readers of the test plan are likely to ask each of these questions. It is important to not only fill out these sections, but more importantly, make sure that the answers to these questions are clearly stated in the front matter.

Acceptance Test Plan

The acceptance test plan describes how the team will determine whether the project meets the acceptance criteria. On most projects, the acceptance test plan is created, orchestrated, and executed by the QA engineer. There are four types of acceptance testing, each distinguished by the type of requirement it is meant to evaluate.

- **Contract Acceptance Test (CAT).** In many cases, software is developed by one organization for another. When this is the case, the software requirements are specified in a legally binding contract. This contract is often called the service level agreement (SLA). The typical SLA stipulates that payment for the project will only be made if all the requirements are fulfilled. Therefore, CAT plans are focused on ensuring that each requirement is completely and unambiguously satisfied.
- **Regulations/Compliance Acceptance Test (RAT).** Often there are rules and regulations governing how software is to perform. While this is especially true in the medical industry, it is surprising how many legal requirements govern various aspects of the software we use every day. Examples of regulations include accessibility (ensuring that the software satisfies the needs of differently abled users), privacy (ensuring that confidential information is not disclosed to unintended parties), protocol compliance (ensuring that file formats and communication protocols are correctly honored) and reliability (ensuring that services are available for mission critical applications).
- **Operational Acceptance Test (OAT).** This type of testing verifies if the software system can function on the hardware and software platform on which it is designed to run. Operational acceptance test criteria are non-functional requirements, describing performance parameters, version compatibility, and network utilization.
- **User Acceptance Test (UAT).** Does the product meet the needs of the end-users? UAT should be used when the requirements are written from the user's point of view. UAT can be the most difficult type of acceptance testing because one can never be sure all the user's needs are documented and because one can never be sure if the user is satisfied with a given design.

Most projects contain a collection of contract, regulation, operational, and user requirements. Because the nature of the requirements and the methods used to validate them can be different, it is often worthwhile to categorize the requirements list by their type. An example of a categorized list of requirements is the following:

Requirement	Type
The software shall be compliant with ADA section §36.303 with regards to large print material	Regulation
The software shall run on iOS 7.2	Operational
The mobile application shall receive weather data from regional weather software	User
The mobile application shall be able to upload photos from the camera roll	User

Care needs to be taken that the acceptance test plan utilizes the best combination of techniques so the tests can be adequately completed. Each acceptance test is performed in a black box scenario where the tester is assumed to have little or no knowledge of the inner workings of the system. Instead, only the outwardly facing interface and functionality are tested. There are two broad techniques used for acceptance testing: alpha testing and beta testing.

Alpha Testing

Alpha testing is conducted by specialized testers who systematically analyze the system. Care should be taken that the test plan clearly describes the necessary qualification of the alpha testers. For example, a server deployed on a web service may have complex operational requirements which require very specific expertise.

Alpha testers are specialists assigned to look at specific aspects of the system

Best Practice 42.1 Contract, regulation, and operational requirements should use alpha testing

Alpha testing is particularly effective for CAT because it often takes a trained QA engineer to decipher the language used to describe contractual requirements. Subtle differences in word selection can determine whether the contract is satisfied or whether it is not. Alpha testing is also effective in RAT because legally stipulated regulations can be complex and have subtle implications. Finally, alpha testing is effective in OAT. Many platforms are very complicated, requiring extensive domain knowledge and experience to understand their idiosyncrasies.

Best Practice 42.2 Acceptance testing of user requirements should have an alpha testing component

Alpha testing can and should be used for user requirements as well. The challenge with UAT alpha testing plans is to list the test cases necessary to adequately verify the system. User stories and use cases can be useful tools in this regard.

Beta Testing

Beta testing, also known as field testing, involves exposing the project to the real end-users in their environment. This has the potential of giving the development team the most accurate and trustworthy information regarding the acceptability of the product to the target audience. Though beta testing is particularly effective with UAT, it can also provide insight with OAT as well. For example, a well-designed beta test program can reveal user system configurations unanticipated by the development team. Beta test plans consist of two components: reporting and recruitment.

Beta testers are users recruited to provide feedback on the system

There needs to be a means for beta testers to report their impressions on the project. As a rule, the more convenient the reporting mechanism, the higher quality the feedback. For example, relying on the beta testers to e-mail feedback to the development team will likely result in uneven and terse data. A better approach is to instrument the project, so it automatically reports usage data back to the development team. This can necessitate a significant development effort.

Since beta testers are not employees, finding users representative of the target audience can be challenging. If the development team relies on volunteers for beta testers, then the testers are unlikely to be representative of the target audience. Also, some users may be hesitant to risk their workflow or home setup on software which is not ready for release. To address these concerns, the development team often needs to incentivize beta testers.

Best Practice 42.3 Building a network of beta testers should be an ongoing process

Because beta testers are such a valuable acceptance test asset, the QA team should constantly be working to fill the ranks of potential beta testers. Recruitment efforts should be made at each customer, user, and stakeholder interaction.

System Test Plan

A robust system test plan systematically verifies that the system functions as expected and that the various components interact as they are designed to do. Because of this dual nature of system testing, both black box and white box techniques are needed. The system functionality is verified through black box testing and the component interactions are validated using white box testing.

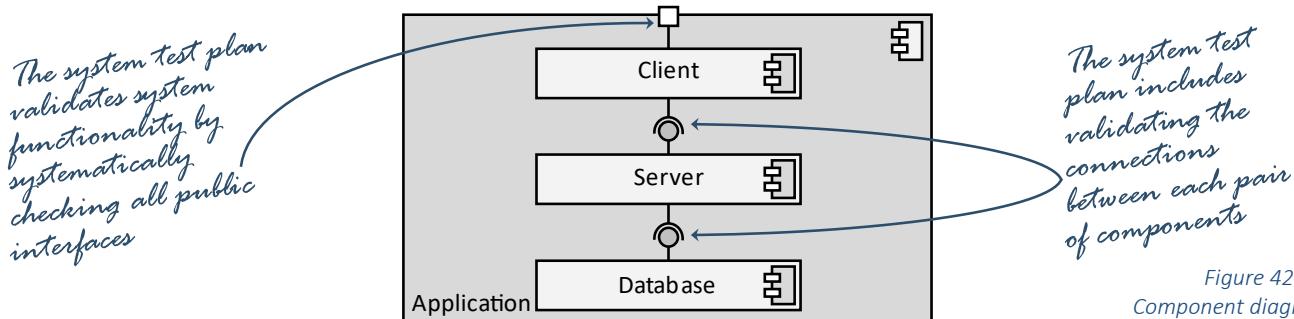


Figure 42.13:
Component diagram
illustrating the two parts of
a system test plan

System Functionality

The system test plan needs to verify that the system functionality is according to the specification. For this reason, it is often called specification-based testing. This is distinct from acceptance testing in two ways. First, the goal is finding defects rather than determine if the project is fit for use. Second, emphasis is made on boundary conditions rather than the most typical path of the user through the system. The system functionality part of the system test plan consists of many test cases, each of which are generated in a similar matter to those created for smaller units of software. Common techniques employed at this level include equivalence partitioning, boundary value analysis, file format and protocol exploration, and syntax testing. For more information how to generate test cases, please see Chapter 14: Test Cases. It is not uncommon for a QA engineer to author the system functionality component of the system test plan, but a SDET may manage this component as well.

Component Interactions

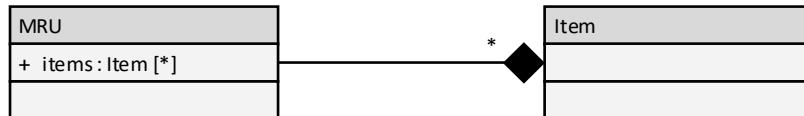
The system test plan needs to verify that the various components comprising the system interact the way they are designed. This is often called structure-based testing. Component interaction testing requires detailed knowledge of how the system is organized and how the individual components work. Surprisingly, the techniques used to test component interactions is like those used to test system functionality.

Consider each component in a system as a system itself. These components accept input, produce output, and have expected functionality. Thus, equivalence partitioning, boundary value analysis, file format and protocol exploration, and syntax testing techniques can be employed. The only difference between testing a complete component and testing a complete system is that the user does not have direct access to a component. Thus, component interactions can only be tested through writing specific test harness code to access them. These test harnesses are typically written and maintained by SDETs. Their specification, as well as the test cases that are exercised through them, constitute the component interaction part of the system test plan.

Integration Test Plan

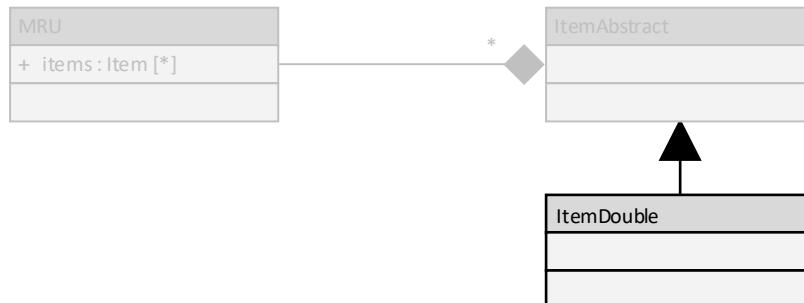
The integration test plan is solely concerned with finding defects in an individual component. Assuming that units are properly verified with unit tests, integration testing checks the interaction between the modules that constitutes a component. The integration component of the test plan is created and maintained by SDET and is usually the lowest-level testing that is described in the test plan. This testing is exclusively black box and debug code must be written to support it. To see how to write an integration test plan, consider two units which collaborate. A most recently used (**MRU**) class contains a collection of items (**Item**).

Figure 42.14:
Class diagram with two
units to be tested



Unit test will independently verify the units through the uses of dummies, fakes, stubs, spies, and mocks. These doubles stand in for the real components which the system utilizes. Two doubles will be created: one to test **MRU** with **ItemDouble**.

Figure 42.15:
Class diagram illustrating
how to test MRU with
ItemDouble



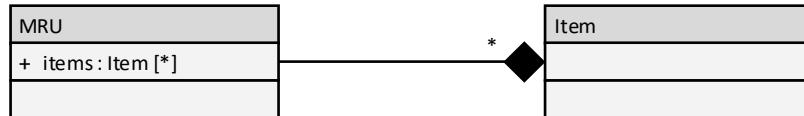
A second double will be needed to test **Item** with **MRUDouble**.

Figure 42.16:
Class diagram illustrating
how to test Item with
MRUDouble



The integration test will combine these units as they are intended and validate that they work together as expected.

Figure 42.17:
Class diagram illustrating
how MRU and Item are to
be tested together



Note that failing integration tests and passing unit tests can mean several things. It could mean that the doubles are not faithful representatives of the classes they are meant to stand in for. When this happens, the doubles should be strengthened. It could mean that the interaction between the collaborators is more complex than what the doubles anticipated. Again, better doubles would address this deficiency. Finally, it could mean that the component design does not fully address the functionality needed by the system. In this case, the component should be re-engineered, and the process begins anew.

Examples

Example 42.1: Acceptance Test Plan

This example will demonstrate how to create an acceptance test plan.

Problem

Describe how to create an acceptance test plan for the following requirements:

Requirements

1. The server shall run on Red Hat Enterprise Linux 8.3
2. The client shall run on Google Chrome 84.0.4147
3. The client shall conform to ADA section §36.303
4. A to-do list shall consist of a to-do name and a to-do collection
5. A to-do name shall consist of text between 1 and 256 characters in length
6. A to-do collection shall consist of zero or more to-do items
7. A to-do item shall consist of text between 1 and 256 characters in length
8. The system shall accept a to-do list from the user and store it on the server

Solution

Requirement #1 is an operational requirement. It will be verified using alpha testing through use of a SDET specializing in Red Hat Linux. The test plan will consist of an enumeration of Red Hat Linux 8.3 system parameters and an enumeration of the software requirements of the server components that will be installed on Red Hat. Each requirement will be cross referenced with system parameters and validated.

Requirement #2 is an operational requirement. It will be verified using alpha testing much like Requirement #1. Additionally, requirement #2 will use beta testing. The beta testing component will consist of allowing the 100 beta testers previously recruited to use the system for two weeks. After the two-week trial, each will individually be interviewed to see if there is a system compatibility problem with their browser.

Requirement #3 is a regulation requirement. It will be verified using alpha testing with a QA Engineer specializing in accessibility requirements. The QA Engineer will enumerate all the accessibility guidelines and form test cases for each. These will then be verified against the system.

Requirement #4-8 are user requirements. These will be verified using both alpha testing and beta testing. The alpha testing will involve a SDET enumerating test cases for each requirement with a special emphasis on use cases. The beta testing will involve allowing the 100 beta testers previously recruited to use a special version of the system that includes instrumentation. This instrumentation will record stability data (crashes), performance data (time to complete a task), and usage data (what types of features are utilized).

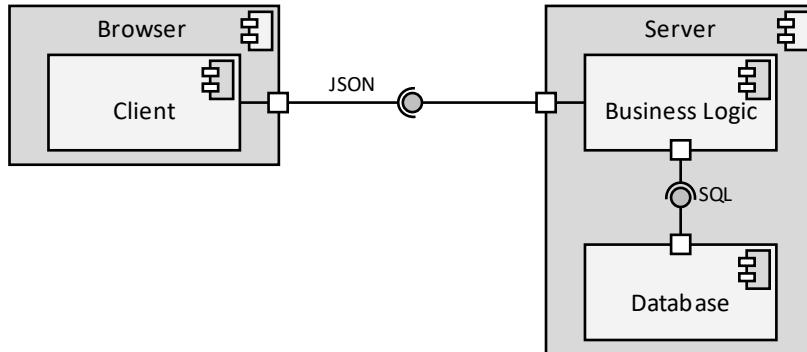
The entire acceptance test plan will be collected in a single document and reviewed by the QA engineer to ensure each requirement has sufficient coverage.

Example 42.2: System Test Plan

This example will demonstrate how to create a system test plan.

Problem

Describe how to create a system test plan for the following system design:



Solution

The system functionality part of the system test plan will be created by first identifying all the ways input can enter the system. This will be accomplished by both analyzing the requirements for user input and by inspecting the system for input ports. For each input avenue, boundary conditions will be identified, and test cases will be created to validate each.

The component interactions part of the system test plan will be created by identifying each component and verifying the ports connecting them. In the case of this system, there are two ports: the JSON port connecting the client and the business logic component, and the SQL port connecting the business logic component with the database.

The JSON port will be verified by identifying all the JSON formats connecting the two components. For each parameter, a suite of tests will be developed exhibiting both valid data and invalid data. These JSON files will be fed to both the client and the business logic component to validate that each handles the input as they should.

The SQL port will be verified by identifying all the SQL queries that the business logic component will send to the database. These queries will be categorized into types of queries, and, for each category, a suite of valid and invalid queries will be generated. This suite will be sent to the database and compared against expected results. The same process will be done for the table results sent from the database to the business logic.

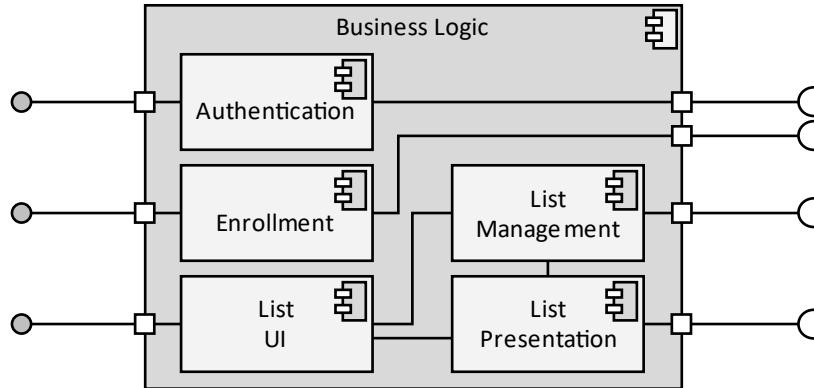
The complete set of test cases will be collected in the test plan. They will be reviewed by the QA engineer and the development team to ensure that adequate coverage is achieved.

Example 42.3: Integration Test Plan

This example will demonstrate how to create an integration test plan.

Problem

Describe how to create an integration test plan for the following component design:

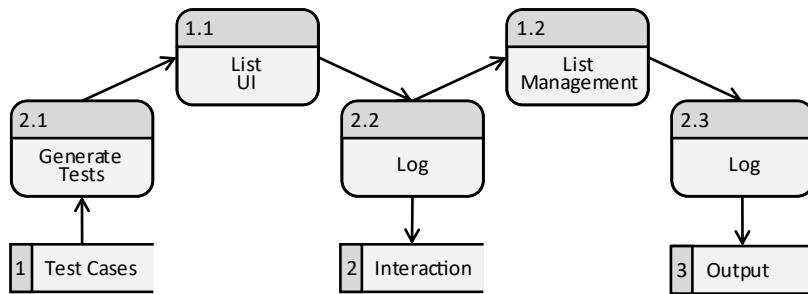


Solution

The business logic component consists of five sub-components, each of which needs to be integration tested after the unit tests are complete.

The authentication sub-component has no interrelationship between the other four components. Thus, the system tests should be sufficient for verifying the external interfaces and the unit tests should be sufficient for the authentication sub-component itself. No integration testing is required. The enrollment sub-component is the same as the authentication sub-component. No integration testing is required.

The List UI, List Management, and List Presentation sub-components are interrelated. A total of three suites of integration testing will be required here. First, the List Management to List UI interaction will be tested. This will be accomplished by writing code to generate tests (2.1) from test cases (1), by creating a logger (2.2) to intercept messages between List UI (1.1) and List Management (1.2), and by writing code to monitor the output from List Management (2.3).

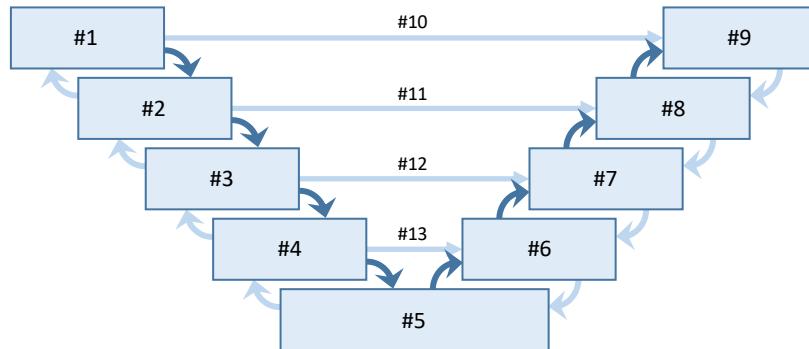


The same process will need to be repeated for data traveling the other way (from List Management to List UI), as well as between the other pairs of sub-components.

Exercises

Exercise 42.1: Name and Define

Name and define each part of the V-Model Process:



Name	Definition
#1	
#2	
#3	
#4	
#5	
#6	
#7	
#8	
#9	
#10	
#11	
#12	
#13	

Exercise 42.2: Activity and Step

For each of the following development, test, or quality assurance activity, identify the step in the V-Model process in which the activity occurs.

Activity	Step
Execute a unit test to verify that a unit of software is complete	
Create a design for the overall layout of the final system	
Implement a unit design	
Create an integration test plan for a single component	
Interview the stakeholders to discover and record what they need from the system	
Execute the part of a test plan that verifies the units within a component work together as they should	
Write code in a high-level language that will be part of the final system	
Create a plan for how the development team can know that the system is fit for use	
Perform beta testing and alpha testing to see if a system is fit for use	
Identify all the units in a single component and describe how they will interact	
Execute the part of a test plan that verifying the components within a system work together	
Create a plan for how the QA engineers will find defects in the overall system	

Exercise 42.3: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
The purpose of system testing is to determine if the system is fit for use	
A unit is smaller than a component	
SDETs create and maintain unit tests	
Alpha testers are often used for acceptance testing	
Acceptance tests are black box tests	
Beta testers are members of the development team	

Problems

Problem 42.1: Acceptance Test

Consider the following software requirements:

Requirements

1. The student user shall be able to filter search results based on company
2. The search feature shall display jobs matching the student user's selected filters
3. Data sent between the client and server shall be AES 256 encrypted
4. The system shall support modern web browsers
5. A student user shall be able to add, alter or remove information from their profile
6. A student profile shall contain a section for accolades

For each requirement, please do the following:

1. Categorize the requirement as contract, regulation, operational, or user
2. Describe the type of tester (alpha, beta, or both) to be used
3. Create one test case

Problem 42.2: Acceptance Test

Consider the following software requirements:

Requirements

1. All transactions shall be transmitted via a SSL connection utilizing HTTPS
2. The mobile application shall run on iOS 7 and later
3. The profile page shall contain content creator information
4. A memory cache will have an initial size of 20 megabytes
5. The profile page shall only be edited by the proprietary content creator
6. The client shall download content via asynchronous HTTPS transfer

For each requirement, please do the following:

1. Categorize the requirement as contract, regulation, operational, or user
2. Describe the type of tester (alpha, beta, or both) to be used
3. Create one test case

Problem 42.3: Acceptance Test

Consider the following software requirements:

Requirements

1. The System shall allow the Student to cancel their Help Request Entry
2. 90% of Transactions shall be completed in no more than 3 seconds
3. The System should acquire the User's name and classes from Jenzabar
4. The System shall have 99.999% of uptime
5. The System shall have a method for a Student to create a Help Request Entry
6. All Posts shall display the date the Post was created

For each requirement, please do the following:

1. Categorize the requirement as contract, regulation, operational, or user
2. Describe the type of tester (alpha, beta, or both) to be used
3. Create one test case

Challenges

Challenge 42.1:

Challenge 42.2:

Large Inheritance Trees

Special challenges arise when working with large inheritance trees. Though they have a reputation for being unmanageable, it is possible to effectively negotiate their pitfalls with a few simple strategies.

When working with small inheritance trees consisting of a dozen or less classes, it is easy to keep everything straight. Things change somewhat when there are hundreds of classes and the inheritance tree is deep. In the early days of object-oriented programming, huge unorganized inheritance trees were common, causing many a programmer to cast a wary eye and avoid such designs. Though no single strategy addresses all the challenges of working with large inheritance trees, there is a great model of how to do this well: the taxonomy of living things.

In the early days of OO design, large disorganized inheritance trees gave inheritance a bad name

In 1758, the Swedish botanist Carl Linnaeus devised a way to classify all living things in a single, all-encompassing taxonomy consisting of just seven layers. For example, a human is the following:

Animalia	Kingdom
Chordata	Phylum
Mammalia	Class
Primates	Order
Hominidae	Family
Homo	Genus
Sapiens	Species

Figure 43.1:
Humans in the
taxonomy of living things

This taxonomy is able to capture approximately 10 million living things (of which about 1.2 million have currently been described) with just seven layers. Each category in the taxonomy correlates with meaningful distinctions (alignment) and all animals sharing the same characteristic are in the same group (redundancy). Perhaps more impressive is that when a new plant or animal is discovered or when something new is learned about a collection of plants, the taxonomy can be easily adjusted to reflect the new findings (adaptability). In fact, the taxonomy has withstood the test of time. It has remained mostly intact through the discovery and application of evolution, molecular biology, and DNA. In short, this represents everything we hope to achieve in designing inheritance trees in our code. There are many lessons from the taxonomy of living things which can be applied to the design and maintenance of large inheritance trees.

Tree Divisions and Branches

In the taxonomy of living things, none of the categories or divisions are arbitrary. Each represents a meaningful distinction between groups of organisms, and all the members contained in each group share a common characteristic. For example, all members of the Animalia kingdom share the same characteristics: they can move, consume oxygen, and eat. Not only is it easy to classify a new living thing as an animal, it is also easy for a non-Biologist to recognize an animal.

Best Practice 43.1 Every base class should map to a design concern

Just as biologists create categories to represent the living things they see around them, software developers create classes to represent design concerns. In both cases, the challenge is to create meaningful divisions so real distinctions and similarities are captured in the inheritance design. The Chordata phylum, for example, consists of all animals with a spinal cord. This is subdivided into several classes, including Amphibia (amphibians), Sauropsida (reptiles), Actinopterygii (fish), and Mammalia (mammals).

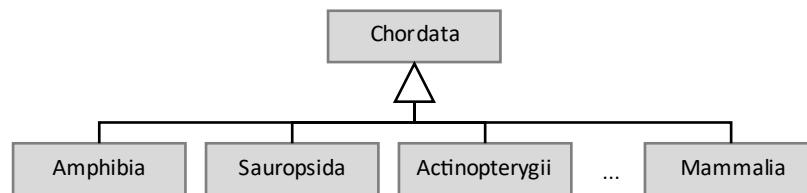


Figure 43.2:
Different types of animals
with spinal cords

Each division within Chordata corresponds to an important distinction; fish are obviously quite different from mammals. All members of the fish biological class share several important characteristics (such as gills for breathing) whereas all members of the Mammalia class share different defining characteristics (females produce milk, and they have hair). When creating inheritance hierarchies, we should seek to codify meaningful differences in a similar way.

Best Practice 43.2 Do not worry about balancing an inheritance tree

Each node in a balanced tree has roughly the same number of children. If the taxonomy of living things was balanced, then there would be roughly the same number of bacteria as plants and animals. This, of course, is not the case. There are, for example, more species of insects (1.2 million) than all the other types of animals combined (0.25 million). If it was a priority of biologists to balance the tree, then some of the members of the insects would need to be placed in the mammals or fishes. This would be counterproductive and make no sense. Just because we “need” more members of the fish class does not mean that a beetle should be a fish!

Artificially balanced inheritance trees are often illogical and hard to work with

When designing an inheritance hierarchy, put no thought into how balanced the tree will be. It does not matter if one branch contains a hundred derived classes and another contains only two. What does matter is that each division is meaningful.

Tree Layers and Ranks

There is a curious feature of the taxonomy of living things. There are only three species in the elephant order Proboscidea, yet there are over 400,000 species in the beetle order Coleoptera (constituting 25% of all animals). In fact, within the entire Proboscidea order, there is only one family (Elephantidae), one genus (Elephas), and three species.

	Proboscidea	Order
1 family	Elephantidae	Family
1 genus	Elephas	Genus
Palaeoloxodon	Maximus	Species
	Primelephas	

Figure 43.3:
The elephant order only
has three members

The beetle family tree looks quite different. Under the Coleoptera order, there are 21 families, of which Curculionidae is one. Under that family, there are 6,800 genera, of which Polydrusus is one. Under that genus, there are 225 species, of which Minutus is one. Note that many new species are described every year.

	Coleoptera	Order
21 other families	Curculionidae	Family
6,800 other genera	Polydrusus	Genus
225 other species	Minutus	Species

Figure 43.4:
The beetle order has over
400,000 members

Why does the elephant order look so different than the beetle order? More importantly, what is the point of having the Proboscidea order with only one family and the Elephantidae family with only one genus? The answer is this: divisions at the order level are uniform across the taxonomy. If two organisms are close together, they must be distinct species in the same genus. If two animals are further apart, they must be distinct genera in the same family. For the taxonomy of living things, each of the seven layers means roughly the same thing.

Best Practice 43.3 Each level in an inheritance hierarchy should mean something

Divisions at the dominion level (a new level above kingdom) are based on cellular structure. Within the Eukaryotes dominion (where plants and animals reside), divisions at the kingdom level are based on how food is procured. Within the Animalia kingdom, divisions at the phylum level are based on the organization of the nervous system. This process continues down to the species level. The important thing to note is that each subdivision at a given rank in the inheritance tree is uniform among the children. This is why the Proboscidea order (elephants) was separated from the other mammals: because their level of difference from other mammals was similar to that of Rodentia (rodents such as mice), Carnivora (carnivores such as dogs), and Chiroptera (bats).

A well-designed inheritance tree should seek to make level distinctions consistent across branches and map to design concerns.

Tree Adjustments

In many ways, the process of creating software is a process of discovery. Seldom is the case when all the required knowledge about a software project is available from the beginning. This means that things are discovered about the problem domain long after the inheritance tree is created. The inheritance tree should never be considered “done”; it is a work in progress.

Seldom is all the required knowledge about a project available from the start

Adding New Derived Classes

When a new organism is discovered, a considerable debate in the scientific community usually follows regarding how this new organism fits into the taxonomy. Is it a new species in an existing genus? Is it a new genus in an existing family? The determining factor of where a new species gets placed in the taxonomy is based on how well it conforms to the defining characteristics of a given division.

Best Practice 43.4 Carefully place new derived classes in the inheritance tree

When an inevitable addition needs to be made to an existing inheritance tree, great care needs to be taken to place the new class in the right spot. While it might be easiest to just throw it into the most convenient spot, this could result in illogical combinations that might be difficult to remedy in the future. Be wary of this temptation! It will almost always create more work for the entire team.

Careless insertion of a class into a large inheritance tree inevitably produces more work for the development team

Modifying Classes

Biologists are continually making new discoveries about individual organisms and how they relate to each other. Some of these new discoveries run contrary to what was previously believed. A great example of this was the infusion of knowledge from genetic analysis of plants and animals. Animals previously thought to be related were actually proven to be not, and plants previously thought to be distant relatives turned out to be close cousins. Biologists do not leave the taxonomy unchanged in the face of these discoveries.

Do not leave the inheritance tree unchanged in the face of new discoveries

Best Practice 43.5 When new understandings invalidate old assumptions, move derived classes as needed

Barnacles are hard-shelled animals living their lives affixed to rocks, animals or boats. They were initially classified as mollusks (such as snails) due to their hard shells. When it became clear that they have jointed legs, they needed to be reclassified as arthropods (insects). The reclassification happened promptly after this discovery.

As changing software requirements are revealed or implementation details are discovered, it often becomes apparent that assumptions are incorrect under which a given inheritance tree were built. When this happens, refactor the hierarchy so it is better aligned to the design concerns.

Best Practice 43.6 Refactor the base classes when needed, but do so with great caution

In October of 2018, the Taxonomic Subcommittee for Lactobacilli, Bifidobacteria and Related Organisms opened the discussion of splitting the *Lactobacillus* genus into several. There are 232 species of this genus currently described. Despite the apparent obscurity of this genus (mostly describing probiotic bacteria useful for digestion and other purposes), several dozen researchers and taxonomists debated the relative merits to a small adjustment (from one genus into four, perhaps requiring future subdivisions once more is learned about the organisms contained therein) or a large adjustment (into eleven genera, risking too fine of granularity and not capturing similarities between related organisms). These options were carefully

weighed by the taxonomic subcommittee, but only those interested in this narrow field of biology were involved in the discussion. This is analogous to refactoring a base class near the bottom of an inheritance hierarchy. A measured amount of caution should be exercised.

Caution should be exercised when altering a base class near the bottom of an inheritance hierarchy

In 1977, Carl Woese proposed a fundamental change to the taxonomy of living things. Before, kingdom was the top rank, separating plants from animals. Woese proposed that the distinction of cell types represented in some microscopic organisms was more fundamental than these. Forty years later, the scientific community is still debating the merits of this change. This is analogous to refactoring the base class at the top of a vast inheritance hierarchy. It is appropriate to place the process under a vast amount of scrutiny.

Modifying the root class in a large inheritance hierarchy should be done under much scrutiny

Best Practice 43.6 The higher the rank of the change, the greater the caution that should be exercised

When refactoring a base class containing a small number of derived classes, a certain amount of caution should be exercised. However, when refactoring a base class high in the inheritance tree, then great care should be taken. As a rule, an order of magnitude more caution should be exercised for each level up the inheritance hierarchy.

An order of magnitude more caution should be exercised for each level up in the inheritance hierarchy

Tree Size

The compiler can handle inheritance trees of arbitrary size; there is no absolute limit on the size of an inheritance tree and there is no performance penalty for large trees. The number of derived classes is often dictated by the problem domain. While an inheritance tree can certainly have duplicate, redundant, or unused classes (all of which should be avoided), having too few can often result in missing functionality.

There is a practical limit to the size of an inheritance tree: the capacity of a programmer to internalize a design.

Best Practice 43.7 Keep the depth of the inheritance tree reasonable

The taxonomy of living things represents an inheritance hierarchy that is eight levels deep. If eight levels are all that is required to describe 1,500,000 members, then perhaps it should be considered an upper limit as to the practical depth of a class inheritance hierarchy. Most applications should be no deeper than three or four.

Seldom should you create an inheritance hierarchy deeper than three or four levels

Best Practice 43.8 Unambiguously communicate the meaning of each division in an inheritance tree

Members of the Mammalia class can sweat, females possess mammary glands (produce milk), can breathe with the aid of diaphragms, have four-chambered hearts, and have hair. However, there are members of the Mammalia class that do not produce milk (such as the Platypus). The defining characteristic of all Mammalia (unique to mammals and present in all members) is a single-boned lower jaw and three-boned middle ear. Every single node in the taxonomy of living things inheritance tree is unambiguously defined and communicated to all taxonomists.

The meaning of every division in an inheritance hierarchy should be unambiguously communicated to all

Best Practice 43.9 Choose base class names carefully

Each division in the taxonomy of living things is carefully named. These names might not make sense to you and me, this is because they are all in Latin. Choosing to name things in a dead language was intentional: it is stable (does not evolve with time) and is equally accessible to all people. Note also that division names are based on the defining characteristic of the division. For example, “Mammalia” refers to the mammary gland. “Chordata” refers to the spinal cord.

When choosing a name for a base class, make sure it honors any established naming conventions in your organization and make sure it captures the defining characteristic of the group of derived classes it is meant to represent.

Examples

Example 43.1: Graphical User Interface

This example will demonstrate how to design a large inheritance tree.

Problem

Create a master plan for an inheritance tree representing the following problem:

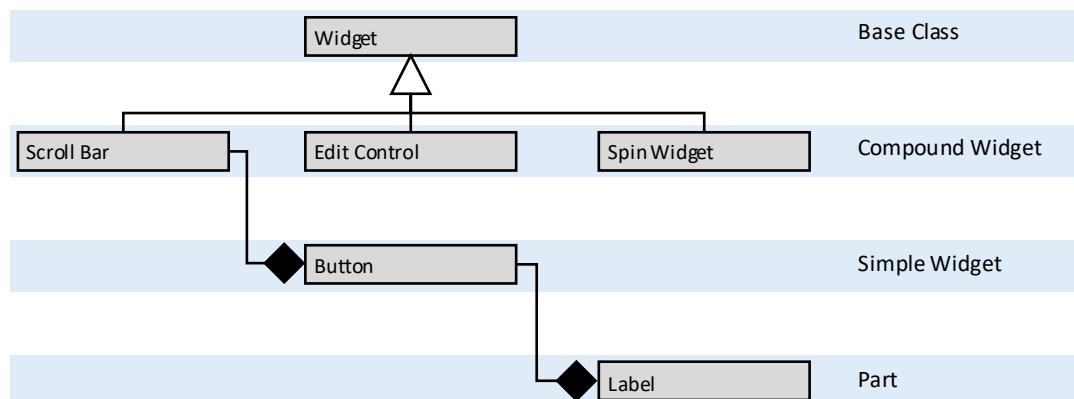
A graphical user interface consists of several types of widgets (buttons, scroll bars, edit controls, etc.), each of which is made up of a series of parts. The complete set of widgets and parts is not known at the beginning of the project.

Solution

We know that a widget is a design concern, as is a part. Note that there are simple widgets such as buttons and complex widgets such as scroll bars that consist of simple widgets. A part, such as an editable rectangle or a clickable region, is the building block of all simple and complex widgets. From this, our initial levels of abstraction are:



With this masterplan, we can create a first draft of our class diagram:



Notice that some levels of this class diagram are represented with inheritance, some with composition. This class diagram is incomplete but serves as a template to place all the elements.

Example 43.2: Shapes

This example will demonstrate how to design a large inheritance tree.

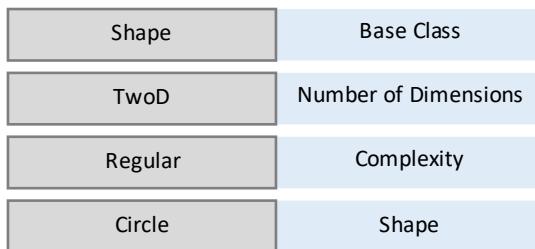
Problem

Create a master plan for an inheritance tree representing the following problem:

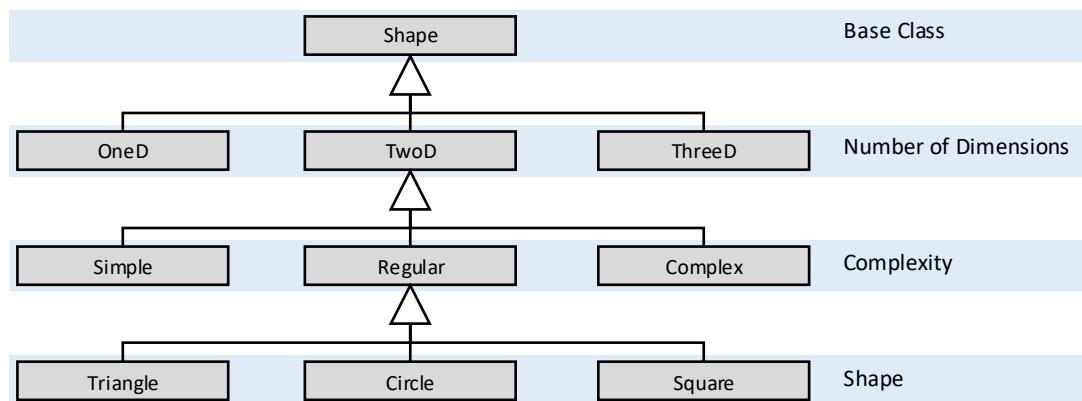
A drawing program needs to represent many types of shapes: points, lines, circles, rectangles, parallelograms, arrows, curves, ellipses, cubes, spheres, etc. The total number of shapes is unknown at the outset, but there will probably be several dozen when the application is finished.

Solution

All shapes need to have the same base class for polymorphic reasons. There appear to be four basic types of shapes: zero-dimensional shapes (of which a point is the only member), one-dimensional shapes (those with a length), two-dimensional shapes (those with a surface area), and three-dimensional shapes (those with a volume). On the next level of the hierarchy, there appears to be some diversity according to the specifics of the branch, but most seem to have a distinction based on the complexity of the shape. A draft of the master plan is the following:



Notice that each level has a meaning in the problem domain, helping with alignment. The design is sufficiently general to allow for many new shapes to be added, helping with adaptability. Finally, there is an obvious place to place all the attributes and operations relating to each division. This helps with redundancy.



Example 43.3: Cars

This example will demonstrate how to design a medium sized inheritance tree.

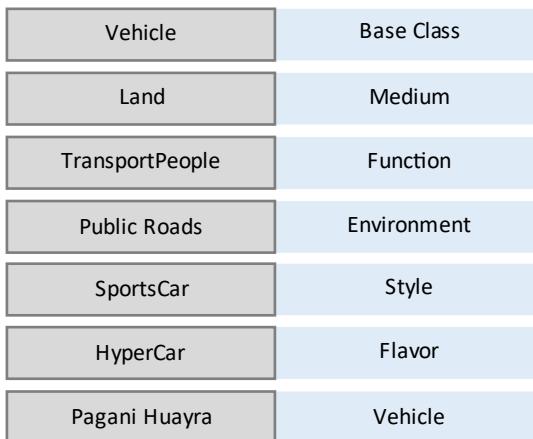
Problem

Create a master plan for an inheritance tree representing the following problem:

An online auction service wishes to create a notion of a vehicle. Each vehicle will have different defining attributes (for example, forklifts measure usage with hours of operation whereas cars do so with miles driven). There are approximately 1,000+ different types of vehicles that this auction service works with.

Solution

There are several candidates for the highest level of the hierarchy: usage (work vs. recreation) or medium (land, air, snow, or water). Since a pickup can be both a work vehicle and a recreation vehicle, the medium appears to be the most important division. Under land vehicles, the next most important distinction appears to be its primary function: transporting people, transporting things, or lifting things. Under the transport people, there next distinction appears to be its primary environment: on public roads, on the dirt, in the sand, or on the racetrack.



Before we build the class diagram, it is important to ask: are we to store different types of data for each sibling at a given level in this hierarchy? Are we going to implement different functionality? While it is clear that the types of things presented in a sailboat ad will be quite different from that of a snowmobile, is the same true for a hypercar vs a regular sports car? Are the differences best represented with a categorization variable or with inheritance? At this point in the design process, there are no clear answers.

The next step is to categorize a random sampling of the 1,000+ vehicle types. From these, build prototype classes. When we present them to the client, he or she will be able to tell us if there are missing layers in the hierarchy or if there are unnecessary distinctions. Recall that designing software is a discovery process on the part of both the client and the developer.

Exercises

Exercise 43.1: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Modification of root classes should be done very carefully.	
Large inheritance trees are inherently unmanageable.	
An inheritance tree deeper than 4 levels is a sign of a bad design.	
Inheritance siblings should share common traits.	
Deep and skinny inheritance trees are always better than shallow and broad ones.	

Problems

Problem 43.1: Units of Measurement

There are several types of units that can be applied in a recipe:

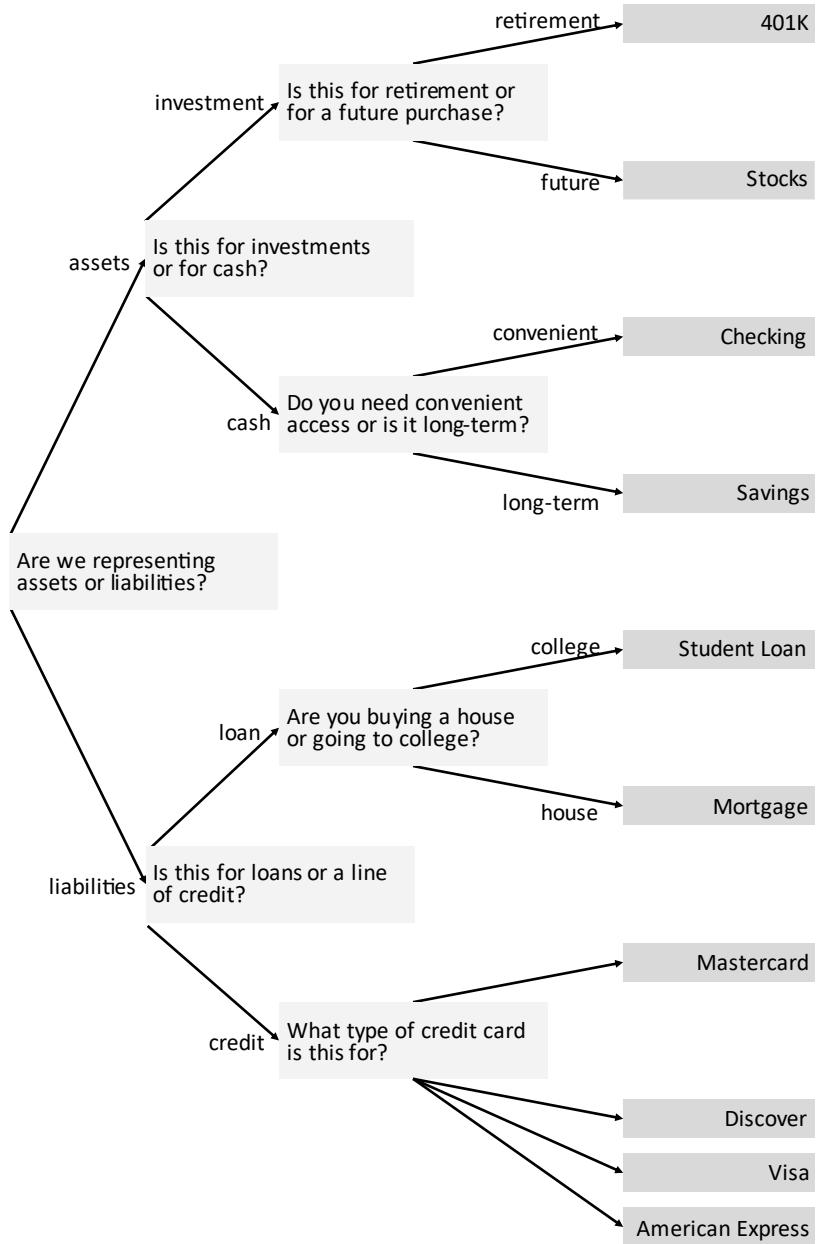
Unit	Type	Class
Gram	Weight	Metric
Cup	Volume	Imperial
Teaspoon	Volume	Imperial
Liter	Volume	Metric
Moment	Time	Informal
Pound	Weight	Imperial
Count	Scalar	Universal
Gill	Volume	Imperial
Ounce	Weight	Imperial
Second	Time	Universal
Milliliter	Volume	Metric
Quart	Volume	Imperial
Celsius	Temperature	Metric
Gallon	Volume	Imperial
Handfull	Volume	Informal
Minute	Time	Universal
Tablespoon	Volume	Imperial
Pint	Volume	Imperial
Centimeter	Length	Metric
Fahrenheit	Temperature	Imperial
Inch	Length	Imperial
Hour	Time	Universal
Meter	Length	Metric
Pinch	Volume	Informal

Please do the following:

1. Create a master plan for an inheritance hierarchy for all units of measurement.
2. For each level in the hierarchy, give it a name.
3. For each base class, describe the defining characteristics of the derived classes.
4. Add a new measurement type: *immediately*. How does this impact your design?

Problem 43.2: Account Types

There are many types of financial accounts:



Please do the following:

1. Create a master plan for an inheritance hierarchy for all account types.
2. For each level in the hierarchy, give it a name.
3. For each base class, describe the defining characteristics of the derived classes.
4. Add a new account type: *mutual fund*. How does this impact your design?

Problem 43.3: Data Types

Consider the following scenario:

There are many data types in the typical programming language: letters, numbers, and logical data. Pick a programming language and research all the data types.

Please do the following:

1. Create a masterplan for an inheritance hierarchy for all built-in data types
2. For each level in the hierarchy, give it a name
3. For each base class, describe the defining characteristics of the derived classes
4. Add a new data type: a custom data type like a class. How does this impact your design?

Challenges

Challenge 43.1: MMORPG

Consider a massively multiplayer online role-playing game such as *World of Warcraft*, *The Elder Scrolls Online*, *EverQuest*, *Final Fantasy XIV*, or any other game you may be familiar with.

Create a master plan for an inheritance tree for this game representing all the meaningful divisions of types of game entities. For each base class, define at least one representative derived class.

Challenge 43.2: Employees

Consider a large organization employing more than a thousand individuals. This could be an organization you have personal familiarity with, a large company such as Walmart or Amazon, the university you currently attend, or a company you are thinking of working for in the future.

Create a master plan for an inheritance tree describing all employees. For each base class, define at least one representative derived class.

Message Passing

The message passing collection of design patterns are intended to simplify the coupling between components by channeling messages through a generic, standard interface.

Captain Campbell is the Alpha company commander in the United States Marine Corp's (USMC) 42nd battalion. Reporting to Captain Campbell is Lieutenant Lewis of the green platoon, Lieutenant Larson of the blue platoon, and Lieutenant Lowe of the red platoon. Each platoon has three squads commanded by a staff sergeant and each squad has three fire teams commanded by a corporal. During field exercises, one of Captain Campbell's greatest challenges is to receive and disseminate information. When everyone is in the same room, things are easy. However, when the captain is away from headquarters and when his platoons are dispersed in the field during an exercise, things become complicated. Captain Campbell will need to adopt more sophisticated message passing strategies to meet the needs of the company.

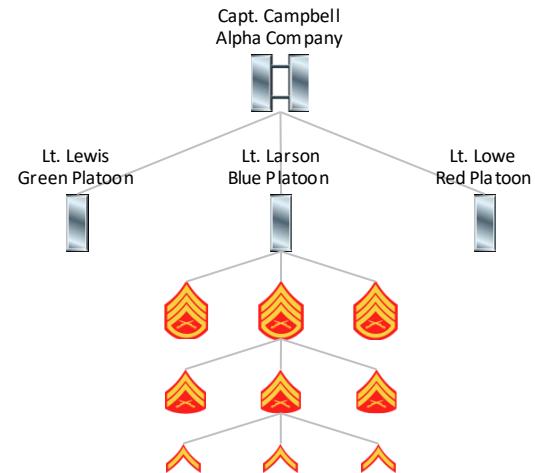


Figure 44. 1:
Organization of Alpha Company

Just as there are many ways of passing information through human communication channels, there are a variety of message passing strategies that can be employed by computers. Recall that computers do three basic things: they process, store, and move information. At the lowest possible level, information movement occurs with a single assembly command where data is retrieved from main memory and placed in a register, or when data is copied from a register to a location in main memory. From this elementary foundation derive a wide variety of message passing strategies: procedure call, mediator, chain of responsibility, observer, and visitor. We call these *message passing design patterns*.

Design Pattern	Description
Procedure Call	A direct, synchronous communication between two functions.
Mediator	All communications between many program entities pass through a central hub.
Chain of Responsibility	A collection of recipients each have opportunity to handle an event.
Observer	Many program entities can subscribe to an event and, when the event occurs, each receives a notification.
Visitor	A reporting structure making little demands on the reporting elements.

Procedure Call

Captain Campbell feels that the best way to relate orders is through face-to-face interactions. When a member of Alpha Company needs a piece of information, that individual is called into Captain Campbell's office and the information is presented verbally. This is analogous to a procedure call.

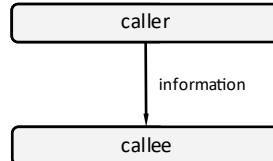


Figure 44.2:
Structure chart illustrating
a simple procedure call

The procedure call is one of the simplest forms of message passing. This occurs during a function call when a collection of data is passed directly from one function to another. This type of message passing is synchronous. The callee receives the data immediately after the caller sends it. The execution of the caller is suspended until the callee completes its task.

Property	Description
Type	Synchronous, both message and control are passed
Distribution	1:1. One caller, one recipient
Strength	Efficient with built-in language support
Weakness	Coupling is tight

Because procedure calls involve only functions, the best way to represent them is through the structure chart, DFD, or pseudocode. Virtually all programming languages provide support for procedure calls through a variety of mechanism.

There are two large disadvantages to using procedure calls. The first is that communication must be synchronous. The caller must pause and wait while the callee is processing a request. While this is desirable in most situations, there are some where it is not. There are times when the caller wishes to send a message, but then must wait for the request to process. Procedure calls do not allow for this to happen; passing messages and passing control happen at the same time.

The second disadvantage is that the caller must know details about the callee. At very least, the caller must know the callee's required parameters. In most cases, the callee must know the caller's name. This makes it difficult to change one without changing the other. In many cases, we would like a system where the caller and the callee know absolutely nothing about each other so both can be updated independently. Procedure calls between components often exhibit tighter coupling than necessary.

Mediator

Captain Campbell was encouraging his platoon leaders and squads to be in constant communication with each other during an exercise. This worked well initially; the three lieutenants were able to efficiently share information and coordinate their activities.

Unfortunately, messages passed from Lieutenant Lewis to Lieutenant Larson often never made it to Lieutenant Lowe. To make matters worse, Captain Campbell was often left in the dark about what was happening in the field. As messages are passed between individual Marines, things quickly get out of hand. With n individuals in the company (there are 243), there are $O(n^2)$ possible communication channels (over 58,000)!

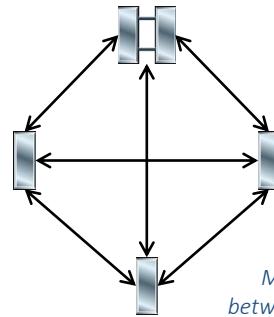


Figure 44.3:
Message passing
between the captain
and the lieutenants

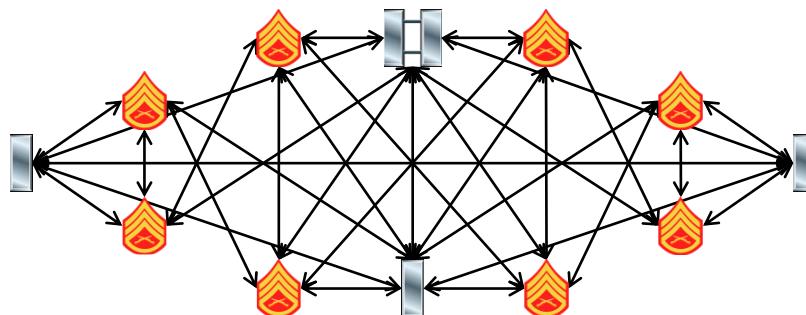


Figure 44.4:
Message passing
between the captain,
the lieutenants,
and the sergeants

To address these critical shortcomings, Captain Campbell instituted a new plan: all communication travels to the Captain's office and they will then be relayed to the necessary party. A mediator is a central hub through which all communication travels.

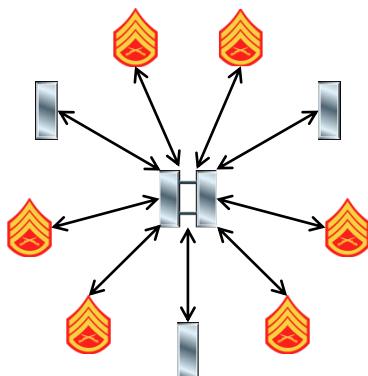


Figure 44.5:
Message passing
using the mediator
design pattern

With this new design, Captain Campbell can ensure that all the necessary information makes it to the required destination. It also enables his team to log communications, provide oversight, and adjust as the situation requires. Halfway through one exercise, a fourth platoon is added to his company. Captain Campbell is relieved! If this happened with his old message passing scheme, every commander would have to be aware of every change. Now, only his team needs to be aware of the change so the new platoon can be easily integrated.

The mediator design pattern is a message passing strategy where all control logic for the entire system is localized into a single location. This location is a class called a director.

Property	Description
Type	Usually synchronous, but can be asynchronous
Distribution	1:Many. One message can make it to many recipients
Strength	Recipient knows nothing about the sender
Weakness	The director needs to know every recipient in the system

The mediator design pattern is characterized by a collection of concrete colleagues, each of which are derived from an abstract colleague. The concrete mediator then contains a reference to each concrete colleague, so it knows where to send messages.

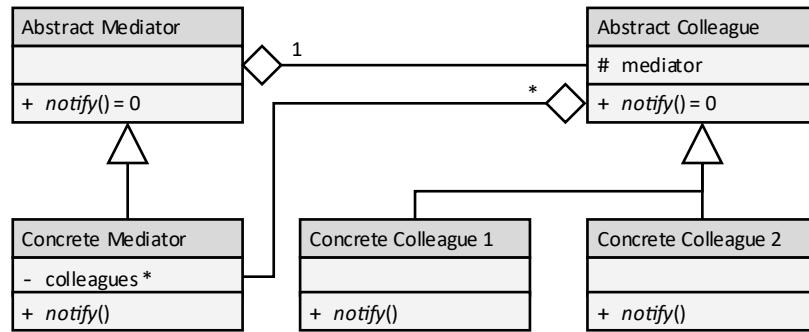


Figure 44.6:
Class diagram of the
mediator design pattern

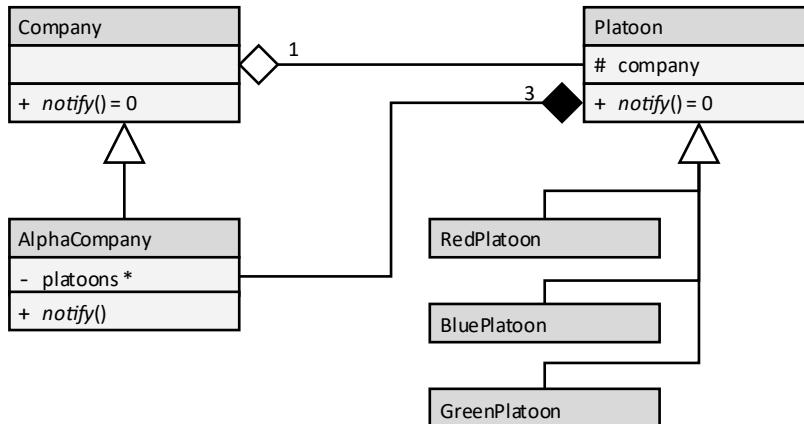
There are several parts to the mediator design pattern:

1. **Abstract Mediator**: This is the abstract class that services as the central hub. All the individual concrete colleagues always have a valid reference to the mediator object so they can send messages. This mediator class has a public `notify()` method which receives messages from the colleague.
2. **Concrete Colleague**: A concrete colleague is one element in the communication network which may send or receive messages. It sends messages by invoking the mediator's `notify` function. It receives messages through its own `notify()` method.
3. **Abstract Colleague**: Each concrete colleague is derived from a single base class: the abstract colleague. This class has two elements: a reference to the mediator and a pure virtual `notify()` method. The constructor to the abstract colleague initializes the mediator reference thereby guaranteeing that all concrete colleagues have a valid reference to the mediator.
4. **Concrete Mediator**: The abstract mediator serves only as an interface for the concrete colleague. This way, it is impossible for the individual concrete colleagues to know about each other. The concrete mediator, on the other hand, contains an instance of all the concrete colleagues. This way, it can send messages to all the concrete colleagues by invoking their `notify()` method.

Once the mediator is set up, communication between the colleagues is simple. Each colleague concrete class calls `mediator.notify()` with the message to be sent. It is then up to the mediator to consume the message, broadcast the message to all colleagues, or to route the message to the appropriate location. This relieves the colleagues of the burden of knowing about each other or managing the message passing process. It also localizes all knowledge of the network into a single location: the mediator class.

When a new colleague is added to the system, two things must happen. First, the new colleague is given a reference to the mediator. This is necessary for the new colleague to send messages. Second, the mediator needs to be made aware of the new colleague. This involves adding the new colleague to the list of colleagues. If the new colleague is to receive special messages, then any routine information needs to be encoded in the concrete mediator's `notify()` method.

To see how this works, we will return to Captain Campbell's communication challenges. Every member of Alpha Company knows how the command structure works: you send all messages to Captain Campbell. This general knowledge is encoded with the abstract mediator class **Company**. Derived from this mediator is the specifics of Captain' Campbell's company. This will be called **AlphaCompany**. There is a model on which all platoons are built, the **Platoon** abstract component. This class initiates the link with **Company** and has a common port through to which all messages flow. Each individual platoon is the concrete component: **RedPlatoon**, **BluePlatoon**, and **GreenPlatoon**. Notice how the platoons are related to Alpha Company through composition rather than association. In this case, the platoons are created when alpha company is created, and dissolved when alpha company is dissolved.



*Figure 44.7:
Class diagram of the
mediator design pattern
applied to Alpha Company*

When Alpha Company is organized, the **AlphaCompany** class is created containing three objects: **RedPlatoon**, **BluePlatoon**, and **GreenPlatoon**. If messages are to be passed to the squads or fire teams, they will need to be added as well.

Pseudocode
<pre> AlphaCompany.initialize() platoons.push_back(new RedPlatoon) platoons.push_back(new BluePlatoon) platoons.push_back(new GreenPlatoon) </pre>

*Figure 44.8:
Pseudocode of initializing
the Alpha Company class*

Now when a message comes to alpha company (through the **AlphaCompany.notify()** method), then Captain Campbell needs to decide what to do with it. The captain decides to log all messages and pass the messages on according to the message type. If the message is a broadcast, then each platoon in the collection receives a copy of the message. However, if the message is meant for an individual platoon, then just that platoon receives the message.

Pseudocode
<pre> AlphaCompany.notify(message) log.push_back(message) IF message.type = broadcast FOREACH platoon in platoons platoon.notify(message) ELSEIF message.type = individual platoons[message.id].notify(message) </pre>

*Figure 44.9:
Pseudocode of Alpha
Company's notify method*

Chain of Responsibility

Captain Campbell has another communication challenge in Alpha Company. As frequently happens when there are many people working in a challenging environment, individuals experience hardship and complain. The problem is that many people take their minor complaints directly to Captain Campbell. Others with serious issues report it to their fireteam leader and the message never makes it to the captain. Clearly, something needs to be done.

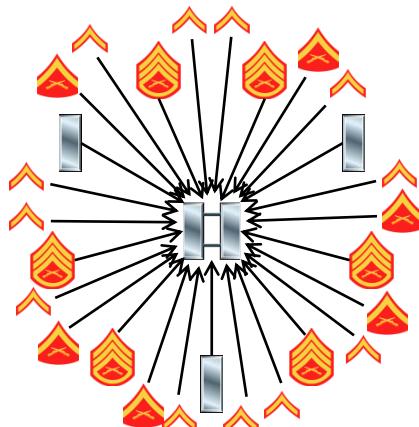


Figure 44.10:
The complaint structure of
Alpha Company before a
new policy is instituted

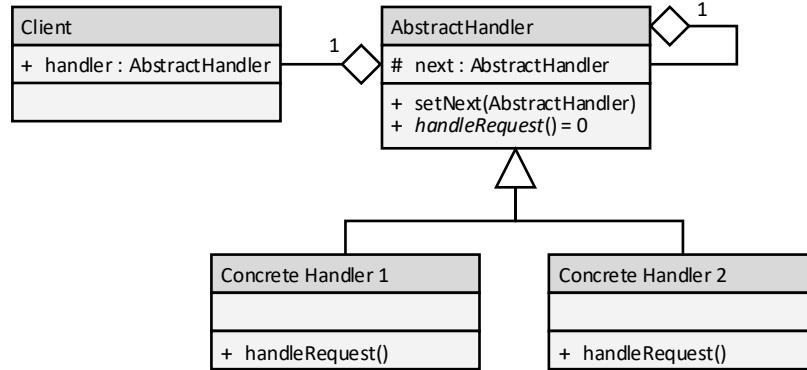
To address this problem, the captain initiates a new policy. All complaints are to be reported to the individual's commander. This individual will handle the complaint if possible but will pass it on to his/her commander if necessary. This way, all serious issues will make it to Captain Campbell, but the minor ones will be handled at lower levels. This works exactly like the chain of responsibility design pattern.

Figure 44.11:
The complaint structure of
Alpha Company after a new
policy is instituted

The chain of responsibility design pattern is a communication strategy where multiple handlers get a chance to consume a message or pass it on to the next handler in the chain. Essentially, it is a linked list of handlers, each of which is given an opportunity to handle a request. If the request is handled, then the process terminates. If the request cannot be handled, then it is passed on to the next node in the linked list.

Property	Description
Type	Usually synchronous, but can be asynchronous
Distribution	1:Many. One message can be handled by many nodes
Strength	The client knows nothing about the handlers
Weakness	Can be difficult to find and fix defects

The chain of responsibility design pattern is characterized by a linked list node called a handler. Observe how it contains a reference to another handler, this reference being the linked list portion of the design.

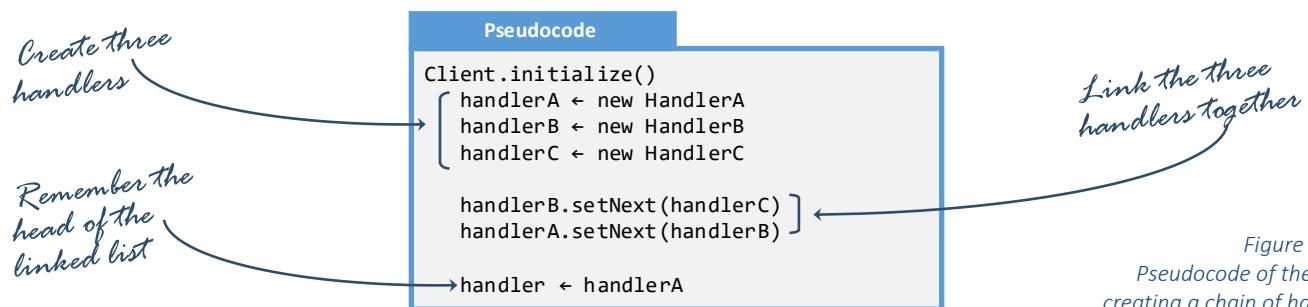


*Figure 44.12:
Class diagram of the
chain of responsibility
design pattern*

There are several parts to the mediator design pattern:

1. **Client**. The client class is responsible for creating the linked list of handlers. It also initiates all message passing.
2. **Abstract Handler**: The abstract node contains the linked list data (`next`), referring to the next handler in the chain. It also has a public `setNext()` method allowing the client to assemble a handler linked list. These two pieces (the `next` pointer and the `setNext()` method) constitute the linked list component of the chain of responsibility design pattern. In addition to this, the abstract handler a pure virtual method called `handle()` that receives messages from the client.
3. **Concrete Handler**: The concrete handler is one manifestation of the abstract handler. It contains all the logic to both determine whether it can handle the request, and to process the request if necessary. If it cannot handle the request, then it passes the request on to the next handler in the chain.

When the client creates a chain, it needs to specify the concrete handlers in the order in which they are to be used. For example, if the client wishes to try `HandlerA` followed by `HandlerB` and `HandlerC`, then the following code will be used:



*Figure 44.13:
Pseudocode of the client
creating a chain of handlers*

It is not uncommon to add a non-default constructor to the `AbstractHandler` so object creation and node insertion can happen in a single step. It is presented here in two separate steps to make the process more obvious.

Observe how the client is responsible for creating the chain of handlers. This creation process involves calling `AbstractHandler.setNext()`.

The pseudocode for this is the same as `setNext()` for any linked list.

Pseudocode

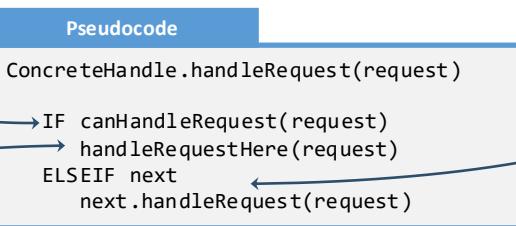
```
AbstractHandler.setNext(abstractHandler)
assert(NULL != abstractHandler)
next ← abstractHandler
```

Figure 44.14:
Pseudocode of the
`setNext()` method

The final piece of the pattern is the concrete handler's `handleRequest()` method. This involves three parts: determination of whether the request can be handled, passing the request on if it cannot be handled, and processing the request if it can.

Determine whether this handler can handle this request

Handle the request in here



If the request can't be handled here, then pass it on to the next handler in the chain

Figure 44.15:
Pseudocode of the
`handleRequest()` method

Back to our USMC example, Captain Campbell has a complain structure following the chain of responsibility design pattern. Every member of the Alpha Company has an individualized concrete `Complaint` object.

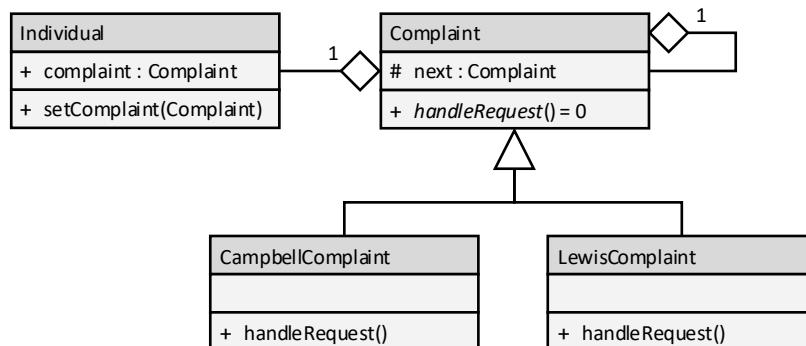


Figure 44.16:
Class diagram of the
complain structure using
the chain of responsibility
design pattern

Observe how `Complaint` is the abstract handler, `CampbellComplaint` and `LewisComplaint` are concrete handlers, and `Individual` is the client. In this case, creation of an individual will create the complaint chain. Since the command structure is uniform across Alpha Company, every `CampbellComplaint` object is the last handler in the chain (or, as Captain Campbell likes to say, “the buck stops here”). Every `LewisComplaint` object has just one element in the chain: `CampbellComplaint`. Staff Sergeant Smith, a squad leader in Lewis's platoon, has two elements: `LewisComplaint` and `CampbellComplaint`. Thus, it is possible to set up the complaint chain completely within the concrete handle constructors.

Pseudocode

```
CampbellComplaint.initialize()
    next ← NULL

LewisComplaint.initialize()
    next ← new CampbellComplaint

SmithComplaint.initialize()
    next ← new LewisComplaint
```

Notice how the creation of this object will also create a Campbell object.

Figure 44.17:
Pseudocode of the
constructor for several
complaint classes

It turns out that Private Pyle has just been assigned to Alpha Company. Captain Campbell has decided to place him in Lieutenant Lewis's platoon, Sargent Smith's squad, and Corporal Carter's fire team. Since Corporal Carter's complaint class has already been defined, it is very simple to add Pyle.

Figure 44.18:
Pseudocode of Private
Pyle's object

Pseudocode

```
Individual pyle
pyle.setComplaint(new CarterComplaint)
```

When finished, Private Pyle's chain of responsibility will look like the following:

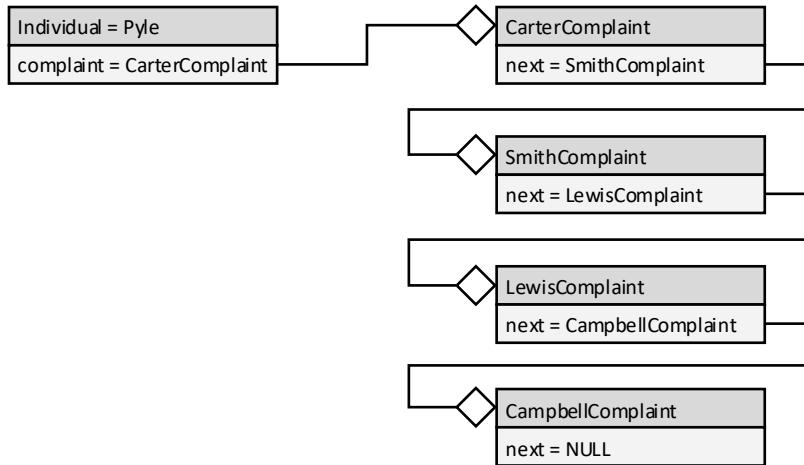
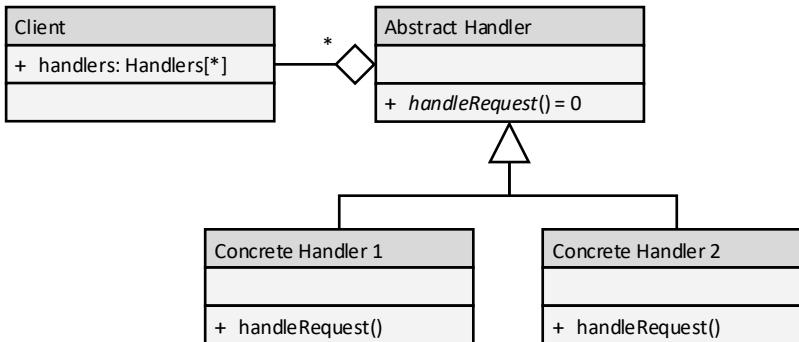


Figure 44.19:
Object diagram of Private
Pyle's object

Variation

Managing a linked list can be problematic. To simplify this process, a common variation is to use a list data structure to maintain the linked list. The client then iterates through the list of handlers, stopping when one is able to handle the event.

Figure 44.20:
Class diagram of a
variation of the chain of
responsibility pattern



There are several advantages to this approach. The client and the concrete handlers are freed from having to manage the linked list—the list class handles these details. The `handleRequest()` methods can also be simplified. They simply return `true` if they can handle the event and `false` otherwise. Finally, the client is afforded more flexibility in how events are handled. The client can then add or remove handlers from the list of handlers using the list's `push()` and `pop()` methods. The disadvantage is complexity. If the chain of responsibility only has one or two handlers, then using the list data structure is overkill.

Observer

Captain Campbell's Alpha Company is part of the 42nd Battalion in the 6th Marines. Every single day, dozens of messages arrive on Captain Campbell's desk from the 42nd Battalion headquarters. Most of these messages are relevant for a very small number of people. This presents Captain Campbell with two bad options: send all the messages to everyone or to not pass any of the messages on. Clearly, there has got to be a better way.

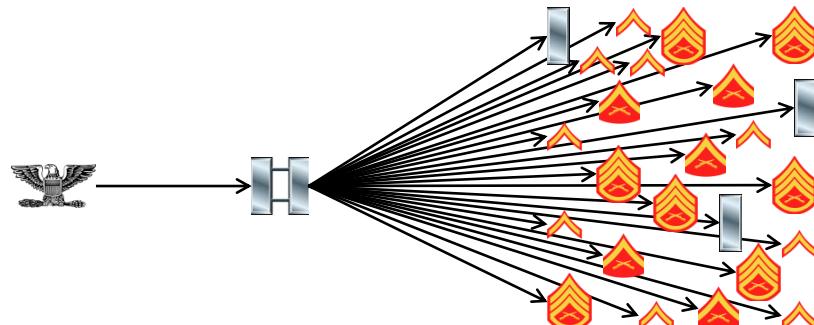


Figure 44.21:
Message passing in Alpha
Company before the
improvement

To address this problem, Captain Campbell has created a collection of special interest groups. Each message coming from battalion headquarters would then be categorized according to the contents and then forwarded to the interested parties.

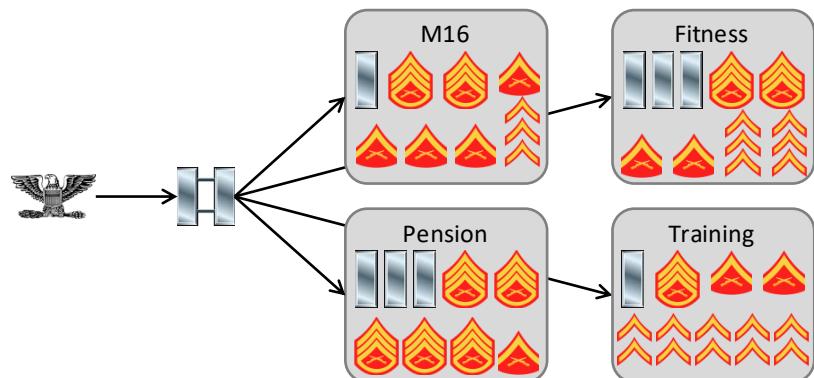


Figure 44.22:
Message passing in
Alpha Company after
the improvement

The observer design pattern, also known as publisher-subscriber, is a message passing strategy where multiple objects can subscribe to a class of events. This ensures that only interested objects will receive notifications. There are two main components to this pattern: subjects and observers. A subject is one that generates messages, and an observer is one who receives the messages. A subject can have zero or more observers, each of which is kept in a list. When the subject publishes a message, then every observer in the list is notified by calling its update method.

Property	Description
Type	Asynchronous
Distribution	1:Many. One message is sent to many nodes
Strength	The publishers and subscribers know little of each other
Weakness	None

The observer design pattern is characterized by a subject which maintains a list of observers, each of whom receives a message when a specific event occurs.

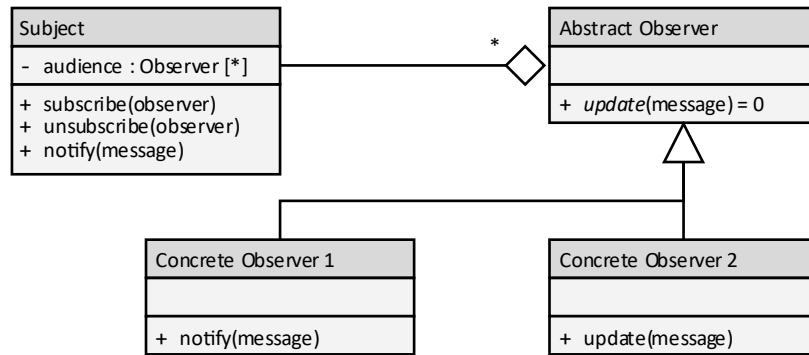


Figure 44.23:
Class diagram of the
observer design pattern

There are several parts to the observer design pattern:

1. **Subject.** The subject is the originator of messages. When the subject initiates the message sending process with the `notify()` method, every observer in the audience receives a copy of the message. This is accomplished by calling the observer's `update()` method. The subject has two methods: `subscribe()` which adds an observer to the audience list, and `unsubscribe()` which removes an observer from the audience list.
2. **Abstract Observer:** The abstract observer is a base class containing only a single public pure virtual method: `update()`. When a message gets sent to the observer from the subject, it is through this method.
3. **Concrete Observer:** The concrete observer is the one who receives an individual message. The specifics of what the observer does with this message is encoded in the various methods and attributes of the concrete observer class.

The key part of the observer design pattern is the `Subject.notify()` method. This method iterates through all the observers in the `audience` list and calls the corresponding `update()` method.

Figure 44.24:
Pseudocode of the notify
method

```

Pseudocode
Subject.notify(message)
  FOREACH observer in audience
    observer.update(message)
  
```

The last two pieces of the observer pattern are the `subscribe()` and `unsubscribe()` method. The former adds a concrete observer to the audience list, and the latter removes an observer from the audience list.

Figure 44.25:
Pseudocode of the
subscribe and unsubscribe
methods

```

Pseudocode
Subject.subscribe(observer)
  IF observer not in audience
    audience.push_back(observer)

Subject.unsubscribe(observer)
  item ← audience.find(observer)
  IF item
    audience.erase(item)
  
```

Back to our USMC example, Captain Campbell notices that many notifications are related to the M16 rifle that is used by about half the company. To address this, he creates a **NotificationM16** class. He also creates a class for pension updates (**NotificationPension**), gym and pool notices (**NotificationFitness**), and a plethora of other topics. Each of these are concrete subjects built from a **Notification** base class serving as the abstract subject. Captain Campbell also creates an **Inbox** class which is the abstract observer. Every member of his company is then given a concrete observer, starting with the captain (**InboxCampbell**) and Red Platoon's commander (**InboxLewis**).

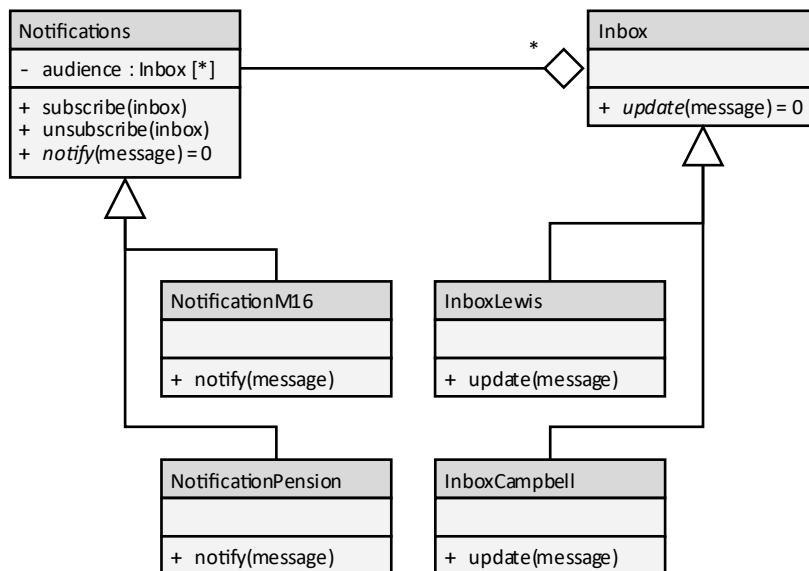


Figure 44.26:
Class diagram of message
passing in Alpha Company

Observe how there are multiple subject classes. This requires an abstract subject (**Notifications**) and several concrete subjects (**NotificationM16** and **NotificationPension**). Because there are many types of messages to be sorted, there will be many classes that derive from **Notifications**. There may also be many classes that derive from **Inbox**, or there may be only one class with a single object for each member of Alpha Company.

To test this out, Captain Campbell subscribes to the M16 rifle, pension, and fitness notifications. Captain Campbell uses his **inboxCampbell** object to do this.

Figure 44.27:
Pseudocode of the captain
subscribing to three types
of messages

```

Pseudocode

notificationM16.subscribe(inboxCampbell)
notificationPension.subscribe(inboxCampbell)
notificationFitness.subscribe(inboxCampbell)

```

The final step of the process is to filter all the messages from the battalion headquarters so they can be sent to the appropriate inbox.

Figure 44.28:
Pseudocode of the message
filtering process in Alpha
Company

```

Pseudocode

IF message.contains("M16") or message.contains("AR -15")
    notificationM16.notify(message)

IF message.contains("pension") or message.contains("401k")
    notificationPension.notify(message)

IF message.contains("pool") or message.contains("gym") or
    message.contains("PRT") or message.contains("track")
    notificationFitness.notify(message)

```

Visitor

Captain Campbell needs to create a weekly report to battalion headquarters. This requires the captain to collect a variety of information from the different aspects of Alpha Company. The platoon leaders need to provide readiness data. The squad leaders need to report on fitness levels. The supply officer needs to report on equipment and expenditures. Even the secretary needs to report on the number of paperclips used. Captain Campbell realized that the process has been a burden to the entire company; every member of the company needs to know about the details of this report, and the format of the report changes frequently.

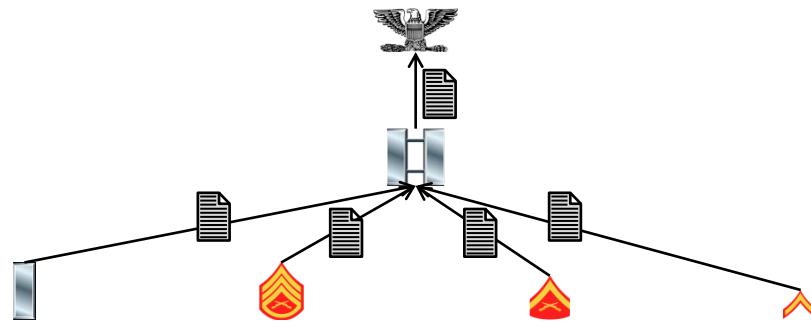


Figure 44.29:
Alpha Company forcing
each member to
understand the report

To simplify this process, Captain Campbell has developed a new strategy. The captain creates an individual form for each group of people, requesting only the information relevant in their position in the organization. This form asks questions that are easy for each individual to answer, but also facilitates generating his weekly report. The captain has assigned one of his assistants to visit each group of people and personally fill out the form. This shields the members of Alpha Company from needing to know about the details of the weekly battalion report.

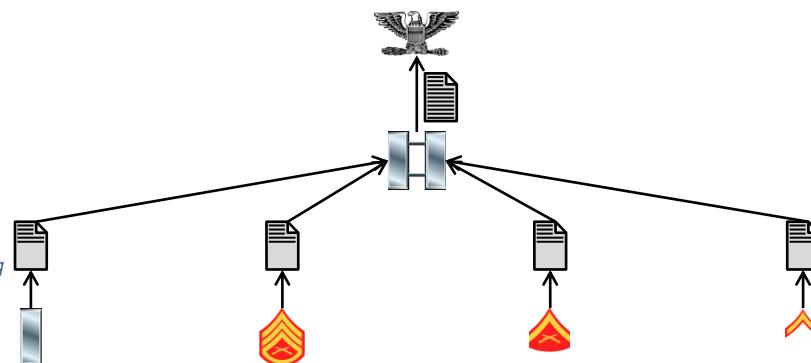


Figure 44.30:
Alpha Company simplifying
the report process by
having an assistant collect
the data for the report

The visitor design pattern is a communication strategy facilitating retrieving data from a collection of objects without forcing the objects into knowing about the data collection process. This is accomplished by placing the collection of objects in a single inheritance hierarchy and then creating a **Visitor** class that knows about the classes being visited.

Property	Description
Type	Asynchronous
Distribution	Many:1. Data is assimilated from many sources
Strength	Individuals are shielded from organization complexity
Weakness	The visitor needs to handle organizational complexity

The visitor design pattern is characterized by a visitor class having one method for each concrete element to be visited.

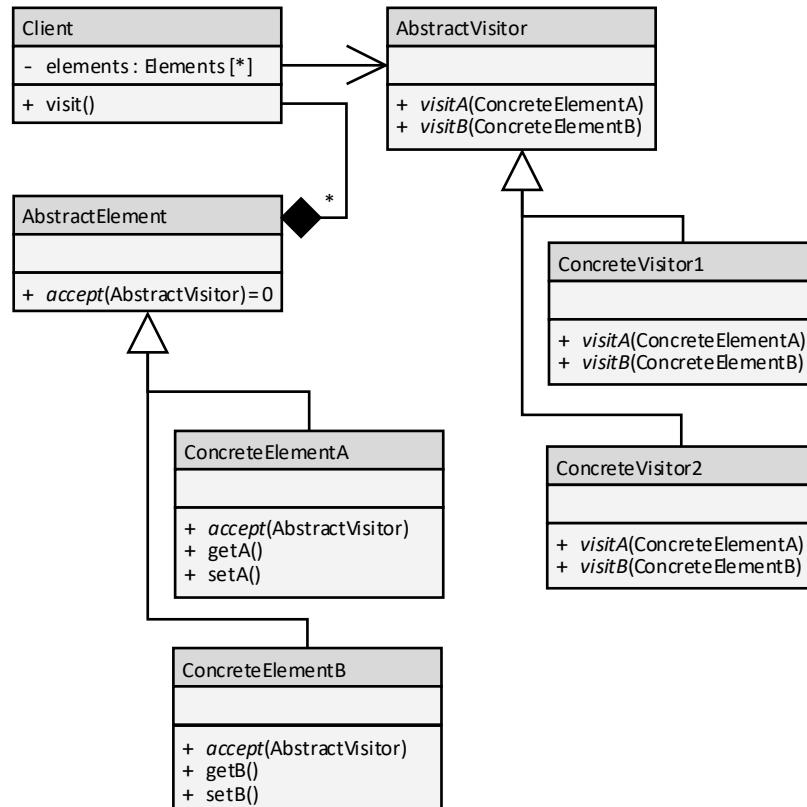
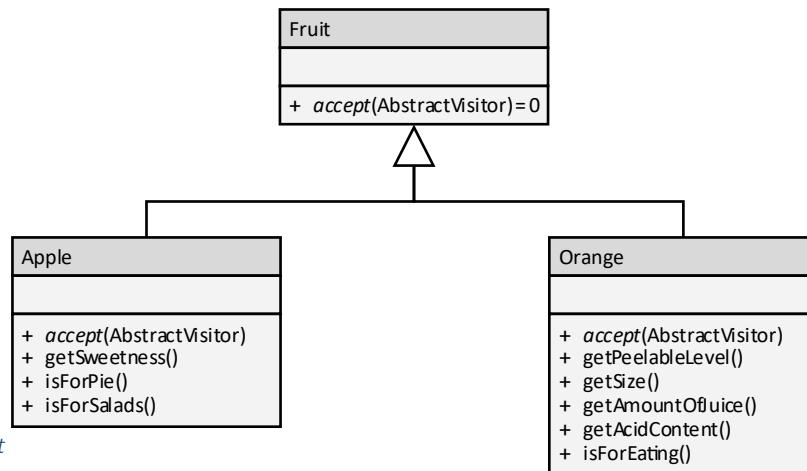


Figure 44.31:
Class diagram of the
visitor design pattern

There are several parts to the mediator design pattern:

1. **Abstract Element.** The base class for all the elements in the collection that are to be visited. This base class must have a pure virtual method called `accept()` which accepts a visitor as a parameter.
2. **Concrete Element:** Each concrete element implements the `accept()` method from the abstract element and contains all the logic necessary to complete its identity. This `accept()` method will then call the corresponding method in the visitor enabling it to collect the information it needs. Note that only the `accept()` method is inherited from the abstract element base class. Each concrete element can be very independent from each other.
3. **Abstract Visitor:** The base class for all the visitor variations, each abstract visitor will have a virtual method corresponding to each of the concrete elements that takes a concrete element as a parameter. This enables the visitor to collect the specific information needed.
4. **Concrete Visitor:** A class implementing the various abstract visitor methods.
5. **Client:** The client contains the collection of all the elements to be visited. It may also contain an instance of the concrete visitor that is to be used, but it does not have to. Since the visitors do not have member variables, they can be instantiated as local variables and not retained as member variables. This is why the association arrow is used between the client and the `AbstractVisitor` class rather than a composition diamond.

A key aspect of the visitor design pattern is how different the concrete elements can be. Aside from the shared `accept()` method inherited from the abstract element base class, these concrete element derived classes can be completely different from each other. With different public interfaces, how can the client hope to gather all the necessary information it needs? To illustrate this point, consider the following elements:



*Figure 44.32:
Class diagram illustrating
how each concrete element
may be very different from
each other*

Note that apples and oranges (two concrete elements) are quite different; they have a completely different set of accessors. This is good; we wouldn't expect a method `isForPies()` to be in an orange class and we wouldn't expect `getPeelableLevel()` to be in an apple class. For us to create a report of all the elements in our collection, we need something which can speak to apples and oranges directly. This is where the visitor class comes in.

The visitor has a dedicated method for each fruit type. Its `visitApple()` method knows about `Apple`'s methods, queries them for the information that is needed, and returns the information that the client needs. It also has a `visitOrange()` method knowing about `Orange`'s methods.



*Figure 44.33:
Class diagram of a
concrete visitor*

The final part of the visitor design pattern is the element's accept method. The accept method in each concrete element takes a visitor as a parameter. This method calls the corresponding visitor method passing itself as a parameter. This way, the concrete element needs to know nothing about the nature of the data being collected; it only needs to pass an instance of itself to the corresponding method in the visitor.



*Figure 44.34:
Pseudocode of two
accept() methods*

Thus, the only code needed by a concrete element to facilitate visitation is a very simple `accept()` class. It is then up to the concrete visitor to serve as an intermediary between the specific interface provided by the element and the needs of the client.

Back to our USMC example, Captain Campbell needs to create a weekly battalion report. It is also necessary to update the general orders to every member of Alpha Company. This requires two types of visitors. The first visitor (**GetReportData**) is an accessor, gathering information from each type of commander in the company. The second visitor (**DistributeOrders**) is a mutator, requiring each element to make a change.

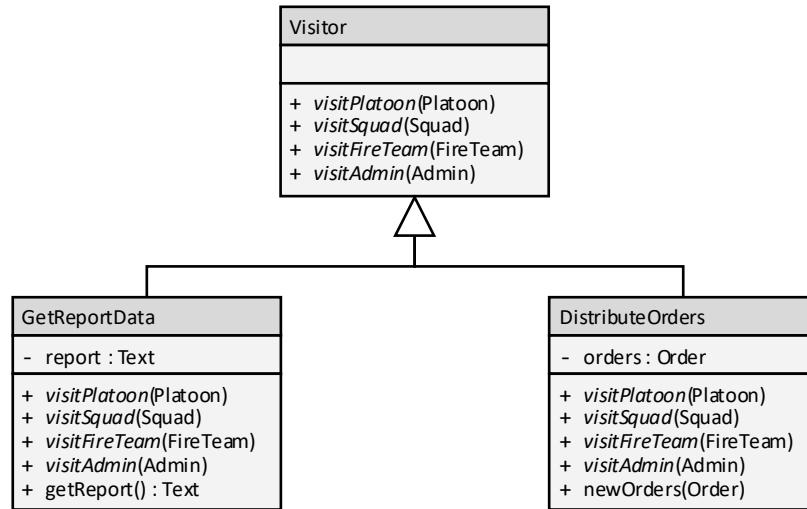


Figure 44.35:
Class diagram of the
visitor classes for
Alpha Company

The organization consists of a collection of roles, each of which has a variety of interfaces unique to their task. The abstract element is the **Marine** class, serving only to provide a common interface for the **accept()** method. The **Platoon**, **Squad**, **FireTeam**, and **Admin** classes are all concrete visitors. Each provides their own public methods specific to their task, as well as implementing **accept()** which will call the appropriate **Visitor** method.

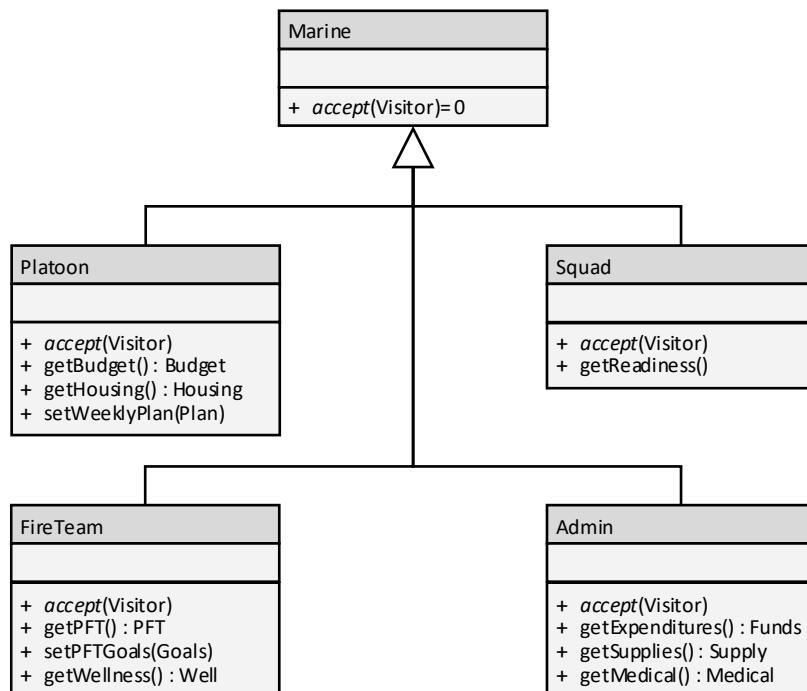
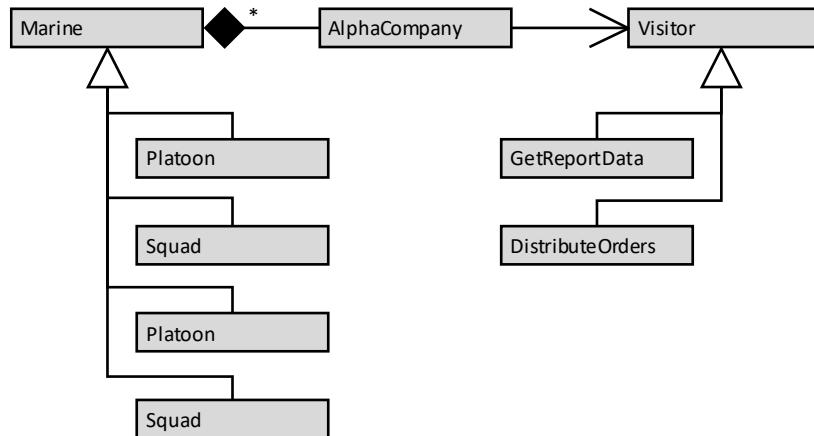


Figure 44.36:
Class diagram of the
elements classes for
Alpha Company

The overall class diagram for Alpha Company's visitor design pattern is the following:



*Figure 44.37:
Class diagram of the
visitor pattern applied
to Alpha Company*

For the weekly update to the company's general orders, Captain Campbell will utilize the `DistributeOrders` visitor. First, the orders are generated. Then a `DistributeOrders` object is created with the orders set as a member variable. Finally, each group will receive a copy of the orders.

*A concrete visitor is
created and set
with the necessary
information*

Pseudocode

```

AlphaCompany.sendGeneralOrders()
  [
    orders <- generateWeeklyOrders()
    distributOrders.newOrders(orders)
    FOREACH marine IN marines
      marine.accept(distributOrders)
  ]
  
```

*Every marine is
asked to accept the
visitor*

*Figure 44.38:
Pseudocode of
the client initiating
a visit*

The fire team leader only cares about the physical fitness test (PFT) goals for the week since it is his/her responsibility to train for and conduct this test. Therefore, the `visitFireTeam()` method of the `DistributeOrders` class will need to retrieve the PFT component of the weekly orders and send it to the fire team leader.

Pseudocode

```

DistributeOrders.visitFireTeam(fireTeam)
  pft <- orders.getPFTGoal()
  fireTeam.setPFTGoals(pft)
  
```

*Figure 44.39:
Pseudocode of a concrete
visitor's `visit()` method*

Observe how the captain does not need to be concerned about the specific fire team interface. It is also not necessary to force the fire team's concerns upon the platoon or squad. The visitor design pattern enables each of these classes to focus on representing their design concerns and yet still coordinate with each other.

Examples

Example 44.1: Mediator

This example will demonstrate the mediator design pattern.

Problem

Consider the following scenario:

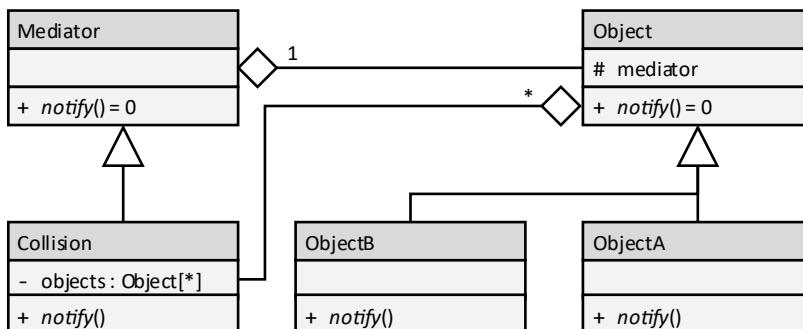
A video game contains many moving objects, each of which has a size and a position. When two objects collide (the distance between them is smaller than the sum of their radius), then both objects are destroyed. In the current design, each object contacts every other object in the game to determine if a collision has occurred. Not only is there redundant computation (object A checks to see if it has collided with object B and object B checks to see if it has collided with object A), but a host of optimizations are difficult to perform in this model.

Determine if a message passing strategy can be applied to this scenario and suggest a design.

Solution

The first thing to observe about this scenario is that each object must communicate with all the other objects in the game. In other words, this is a many-to-many relationship. If there are n objects in the game, then adding another object adds n more pairs to be compared. This is an $O(n^2)$ algorithm.

There are a variety of optimizations that can be performed. Each leverages the fact that only neighboring objects could collide; objects far apart do not need to be considered. Strategies such as segmentation, partitioning, pruning, and bounding can reduce the complexity of the algorithm. These can only be performed if all collision detection occurs in a single location. The mediator design pattern facilitates this process.



Every object in the game derives from the **Object** class. Collision detection occurs through the **Collision** class which maintains a list of all the objects in the game. Here the collision detection algorithm exists, determining whether two concrete objects need to perform collision detection. This design can facilitate other game features such as rendering by adding derived classes to the **Mediator** base class.

Example 44.2: Chain of Responsibility

This example will demonstrate the chain of responsibility design pattern.

Problem

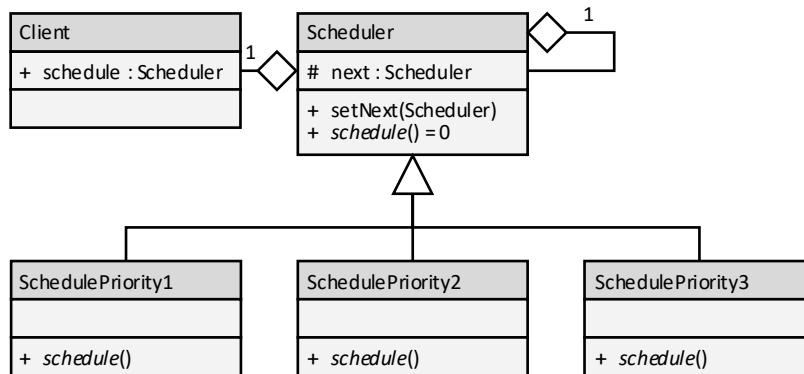
Consider the following scenario:

A calendar application is tasked with moving to-do items onto a schedule. Each item has a duration and a priority (1-5). The algorithm should first schedule all priority 1 items first, then move on to the priority 2 items.

Determine if a message passing strategy can be applied to this scenario and suggest a design.

Solution

This algorithm requires all priority 1 items to be scheduled first, followed by priority 2 items, and so on. This fits nicely into the chain of responsibility design pattern. Here, each priority is handled by a different scheduler.



Upon initialization, `SchedulePriority1` is first added to the `Scheduler` linked list. Next, `SchedulePriority2` and so on. When the client invokes the `schedule()` method, the priority 1 scheduler will attempt to handle it first. Only if it fails will it revert to the priority 2 scheduler and so on.

Example 44.3: Observer

This example will demonstrate the observer design pattern.

Problem

Consider the following scenario:

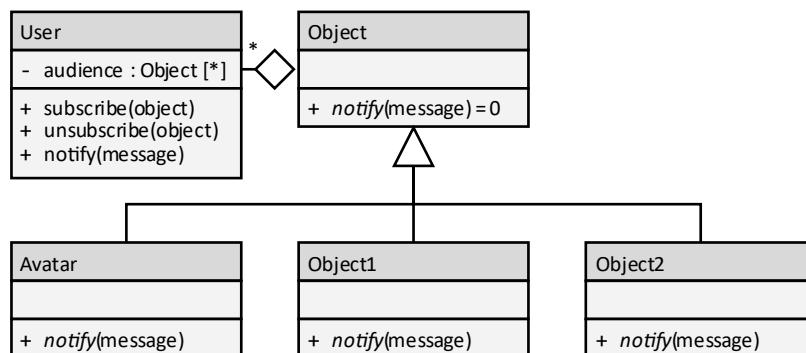
A video game consists of many movable objects in the arena. All the objects move on their own accord, but one (the avatar) can be controlled by the user. In certain circumstances, the user can control many objects at once.

Determine if a message passing strategy can be applied to this scenario and suggest a design.

Solution

In most cases, there is a one-to-one relationship between the controller (the user in this case) and the controlled (the objects in the game). This can be handled with a simple procedure call connection between the user and the avatar. However, there is a complication. There is a requirement that many objects can periodically be controlled by the user. It would be difficult to facilitate this through a procedure call.

The observer design pattern facilitates dynamically adding and removing event listeners to objects. Initially, only the avatar subscribes to user events. However, when the user commandeers many objects, these objects also subscribe to user events.



When the game is initialized, then a **User** object contains only a single **Object** in the **audience** collection: an **Avatar**. User input is then sent directly to the **Avatar** object through the **notify()** method. When the user is given temporary control over other objects, then those objects are added to the **audience** collection through the **subscribe()** method. This occurs until control is removed through the **unsubscribe()** method.

Example 44.4: Visitor

This example will demonstrate the visitor design pattern.

Problem

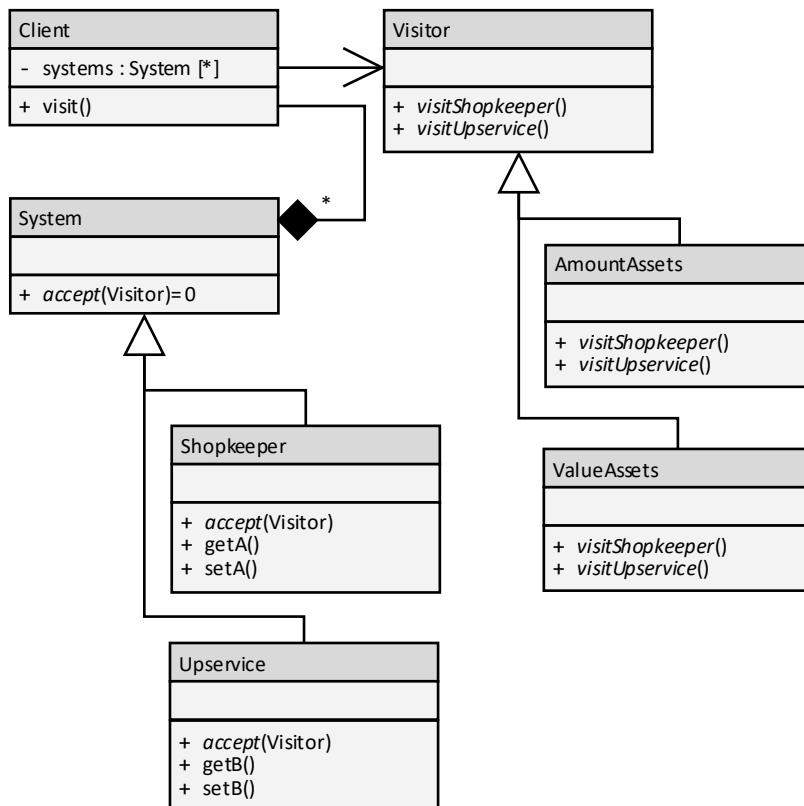
Consider the following scenario:

An inventory program has a reporting feature describing the amount of assets, the value of the assets, and a variety of other properties. The assets are each stored on several systems (Shopkeeper and Upservice), each with its own custom interface.

Determine if a message passing strategy can be applied to this scenario and suggest a design.

Solution

The store systems have unique interfaces that are not easily changed. The visitor design pattern can alleviate the need for the client from needing to couple directly with these interfaces.



Here, only the visitors **AmountAssets** and **ValueAssets** need to be aware of the specific interfaces in the various systems. To facilitate this, a single method needs to be added to the systems: **accept()**. This method will then call the corresponding **visit()** method in the **Visitor** base class.

Exercises

Exercise 44.1: Types of Message Passing

From memory, name and define the five types of message passing strategies.

Name	Definition

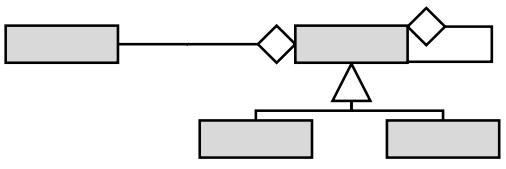
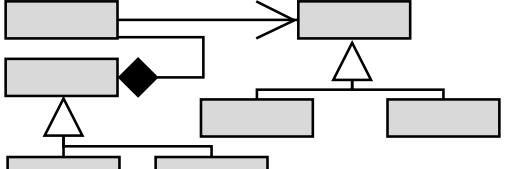
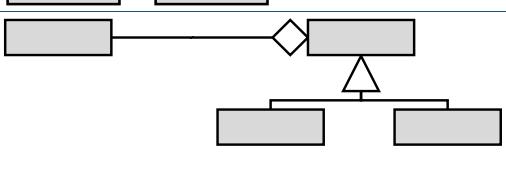
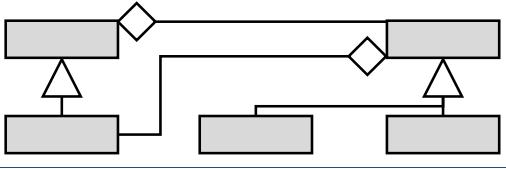
Exercise 44.2: Components

The five message passing design patterns consist of several component, each of which is represented as a class or as a method within a class. For each component, identify the corresponding design pattern and briefly describe the role it fulfills.

Component	Design Pattern	Role
Observer		
Colleague		
Client		
Element		
Mediator		
Subject		
Visitor		
Handler		

Exercise 44.3: Class Diagram

From each of the following class diagram, identify the design pattern.

Class Diagram	Design Pattern
	
	
	
	

Exercise 44.4: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
The observer pattern is for many individual nodes to send message to a single central node.	
The mediator pattern facilitates the colleagues to talk with the mediator, but not the other way around.	
The visitor design pattern requires each element to share the same public methods.	
The chain of responsibility pattern is essentially a linked list.	

Problems

Problem 44.1: Airport Control Tower

Consider the following scenario:

For airplanes to safely arrive and depart from an airport, they must communicate with each other using the radio. Currently, the airport requires each airplane to communicate with each other airplane in the area, so they maintain a safe distance from each other.

Provide a mediator design to address this scenario.

Problem 44.2: Spaceship Defenses

Consider the following scenario:

A video game is implementing a collision detection system. When an enemy fires at the main ship, the ship can use a shield, defection, or an anti-missile to defeat it. These have various degrees of effectiveness depending on the nature of the enemy missile.

Provide a chain of responsibility design to address this scenario.

Problem 44.3: Spaceship Control

Consider the following scenario:

A video game has a collection of elements on the screen. Each element can optionally be controlled by the player, commandeered by the enemy, or destroyed by a smart bomb.

Provide an observer design to address this scenario.

Problem 44.4: Video Game Score

Consider the following scenario:

A video game has a collection of elements on the screen. There are obstacles to be removed, bonuses to be collected, and enemies to be destroyed. The game provides several scores: the number of bad things to be killed, the good/bad ratio, the number of obstacles, etc.

Provide a visitor design to address this scenario.

Problem 44.5: Financial Report

Consider the following scenario:

A financial program allows the user to view many graphs and reports at the same time. The program also allows the user to edit the transactions at any time. If a transaction is changed, then any graph or report drawing data from the transaction must be updated as well.

Determine if a message passing strategy can be applied to this scenario and suggest a design.

Problem 44.6: Code Editor

Consider the following scenario:

A code editor as a feature that analyzes your source code and creates a report. The code editor supports a wide variety of different file formats, each of which may report very different things depending on the nature of the elements contained in the file format.

Determine if a message passing strategy can be applied to this scenario and suggest a design.

Problem 44.7: P2P

Consider the following scenario:

A video game allows multiple players to share the same game through a network connection. In the past, they have followed a peer-to-peer (P2P) format where any player can play against any other player. This has led to a collection of security problems. In the new design, all interactions go through the central server which authenticates users and prevents cheating.

Determine if a message passing strategy can be applied to this scenario and suggest a design.

Problem 44.8: Coding Errors

Consider the following scenario:

A code editor would like to provide a suite of error detection features to help programmers write better code. Some of the errors are catastrophic, meaning the code will not work until the error is fixed. Some errors are stylistic, meaning the programmer can safely ignore the error if he or she chooses. There are six levels of severity in this code editor: catastrophic, crashing, serious, questionable, warning, and stylistic. If a severe error is detected, then less severe errors are not reported.

Determine if a message passing strategy can be applied to this scenario and suggest a design.

Challenges

Challenge 44.1: News Feed

Consider the following scenario:

A news site would like to give a user only news articles which are interesting to him or her. To do this, the site has a category list containing dozens of types of articles. The site then asks the user to specify which categories interest him or her.

Please do the following:

1. Research the categories used by your favorite news site.
2. Determine if a message passing strategy can be applied to this scenario.
3. Suggest a design using class diagrams and pseudocode.
4. Implement your design using the programming language of your choice.
5. Create a dozen simple articles in JSON and demonstrate that your program will generate a news feed matching the user's preference.

Challenge 44.2: Complaints

Consider the Alpha Company complaint structure described in the chain of responsibility section of the reading. In the programming language of your choice, implement this design. For each handler, create a simple `canHandle()` function based on complaint priority or any other convenient metric. Demonstrate that your program works by creating a few sample complaints.

Separation of Concerns

Separation of concerns is the process of dividing the three responsibilities of a class (store data, implement public interfaces, and perform business logic) into separate classes for the purpose of making a design more cohesive, less redundant, and more adaptable.

Most classes perform three actions: store data, implement public interfaces, and represent business logic. For a simple application, it is perfectly acceptable to have a single class handle these three actions. However, for complex and multifaceted applications, this can result in overly complicated designs with duplicate code. One strategy for addressing this problem is to separate each of these actions or concerns into separate classes.

Separation of concerns (SoC) is the process of making separate classes for data storage, public interfaces, and business logic. A class representing a specific design concern would then be created by combining these three elements.

Avoid collocating code that does different types of things



*Figure 45.1:
Class diagram of
a class split into
the three concerns*

In the above example, the client only interfaces with the **GroceryInterface** class. The only purpose of this class is to provide the most convenient and abstract interface for the client. **GroceryInterface** sends all edit and information requests to **GroceryLogic**. This class handles the business logic of the grocery list, making sure that items are categorized correctly and that duplicate items are combined. It is important to note that **GroceryInterface** knows nothing about this business logic. **GroceryLogic** then sends updates to **GroceryStorage**. The only job of **GroceryStorage** is to maintain a list of data items.

It does not validate anything (that is the domain of **GroceryLogic**) and it does not deal with the specific needs of the client (that is the domain of **GroceryInterface**). Each class does one thing and one thing only.

[SoC], ... even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts
— Edsger W. Dijkstra

Three Concerns

Classes are concerned with three things: providing a convenient interface, representing business logic, and storing data

The encapsulation metric of fidelity states that a class should represent a single design concern (Chapter 21 Metric: Fidelity). This is highly related to the modularization metric of cohesion, that a unit of software should represent one single concept or perform one single task (Chapter 12 Metric: Cohesion). Since classes tend to represent three concerns (provide convenient interfaces, represent business logic, and store information), it is often possible to increase the fidelity and cohesion of a class by considering each of these concerns separately.

Storage

As the name suggests, the storage concern controls how information is represented in memory. For simple classes, this amounts to a couple of member variables. In cases like these, no special storage concerns are necessary. Many classes in large-scale applications are not this way. They store information in data structures, external files, remote network locations, or databases. Some even do a combination of these things. Dealing with the complex data needs of an application, let alone the situations when more than one data representation is necessary, is a full-time job. The storage concern focuses solely on data representation and manipulation.

The storage concern controls how data are stored in a class and is solely focused on data representation and manipulation

In many scenarios, the storage requirements of a class are simple. For example, if our application were a computer game with many different entities, then the storage requirements of each entity would be trivial: its position and size. Note that it might later be revealed that other properties should be added (health, status, etc.).



Figure 45.2:
Class diagram of
a simple storage class

Many storage classes manage collections of data. This requires them to provide each of the common collection operations (See “Chapter 28 Strategy: Metaphor” for a discussion on common collection interfaces):

Operation	Description
Insert	Add a new element into a collection
Remove	Remove an existing item from a collection
Get	Retrieve a value from a collection
Update	Change an element that exists in a collection
Find	Locate an element that is within a collection
Size	Determine the number of elements in a collection
Iterate	Visit each element in a collection

These collection interfaces are useful across a wide variety of data structures, file formats, and other data representation schemes.

If we apply this principle to our grocery list application, this class might look something like the following:

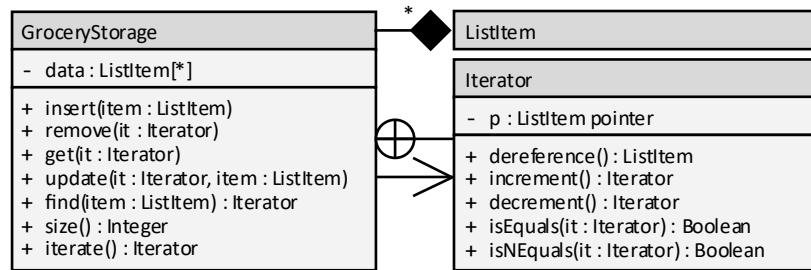


Figure 45.3:
Class diagram of
a collection storage class

Notice that the **GroceryStorage** class implements all the collection interfaces. It contains many **ListItems**, each of which describes a single item in a grocery list. There is also an **Iterator** nested class (as denoted by the pinwheel \oplus symbol). **GroceryStorage** does not have an **Iterator** as a member variable, but utilizes one through association (the open arrow \rightarrow symbol). The **GroceryStorage** class completely handles all the data storage needs of the list application. Of course, it does not provide a convenient interface for the client nor does it handle any of the business logic. It needs a couple of collaborators for that.

A financial application will also have nontrivial storage requirements. It needs to handle simple transactions (purchase, deposit), interconnected transactions (transfers between accounts), cash accounts (checking, savings), market accounts (stock, mutual fund, currency exchange), and aggregate accounts (a mutual fund tracks several stocks). All of these things need to be handled under a single set of shared interfaces.

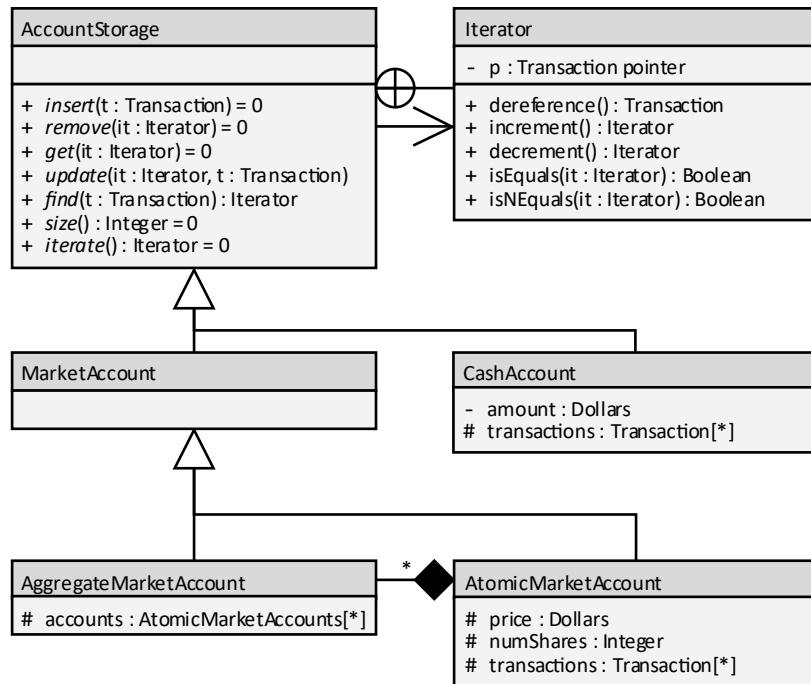


Figure 45.4:
Class diagram of
a class complex storage
inheritance tree

This storage class is quite complex. Through a single shared interface (**AccountStorage**), a wide variety of different account types are implemented. Each one stores account data in a different way. None of these implementation details are shared with the **AccountStorage** class, let alone with the business logic or the client-facing interface. The goal here is to be able to freely add different account types without having to alter any of the other parts of the model.

Business Logic

The business logic portion of a class governs the rules pertaining to how data are manipulated. Perhaps the term “business logic” is a bit misleading: this approach applies to all applications whether or not they govern commerce. If a class were to represent time/date, then the business logic would represent the number of days in a month and hours in a day. If a class were to represent currency, then the business logic would represent the exchange rates. If a class were to represent a zip code, then the business logic would validate that the value corresponds to an actual post office. In all these cases, the business logic makes sure the data are always in a valid state and that the behavior of the class is aligned with the design concerns.

In many cases, the business logic concerns can be handled with a single class and a handful of methods. For example, a financial application would have business logic pertaining to the rules governing account management: overdraft verification, transfer balance, application of fees, etc. It would then use `AccountStorage` to make the actual alterations to the user’s account.

The extent of the business logic concern

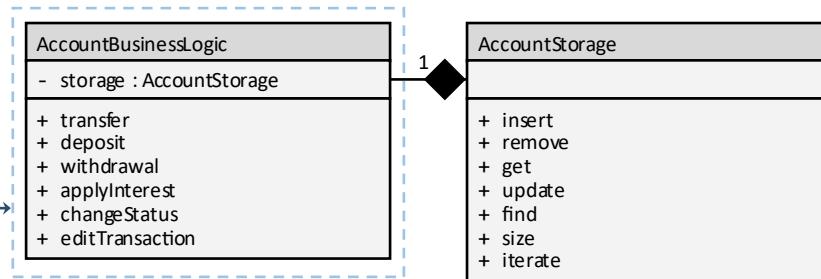


Figure 45.5:
Class diagram of a simple business logic class

The business logic class does not contain any data directly; it only contains an instance or a reference to a storage object. All alterations to program state occur through the storage class’s public methods. Thus, a single business logic method call (such as `transfer`) will involve multiple storage method calls (such as “`insert`” and “`remove`”) as well as the necessary validation to make sure that the action is legal.

As with the storage concern, the business logic component often has its own complex inheritance hierarchy. In a video game, it would represent the rules of the game, including how elements move and interact with each other. Notice that business logic classes contain almost no state; that is the job of storage.

The extent of the business logic concern

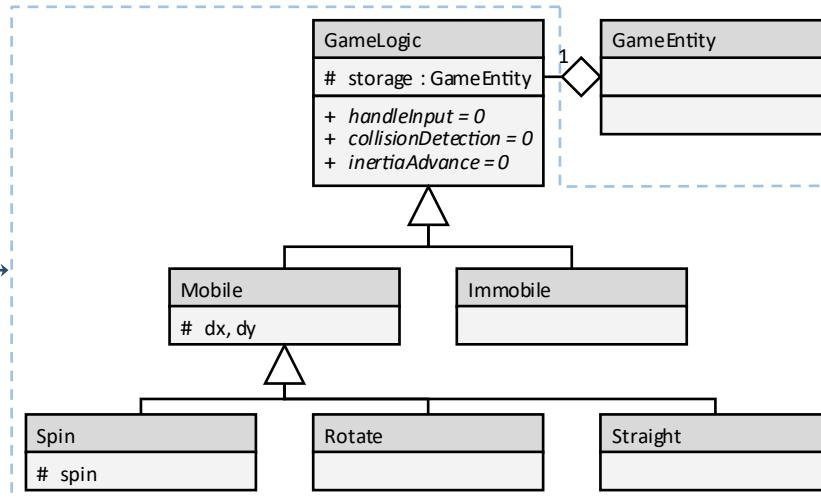


Figure 45.6:
Class diagram of a complex business logic inheritance hierarchy

Interface

The interface concern provides the public interfaces allowing the client to utilize the class. In order to maximize the usefulness of the class, the interface should be as convenient as possible. This means all public interfaces should be heavily aligned towards the needs of the application. In many cases, the interface concern is implemented with a simple class that translates client actions into business logic requests and then translates storage updates into client output.

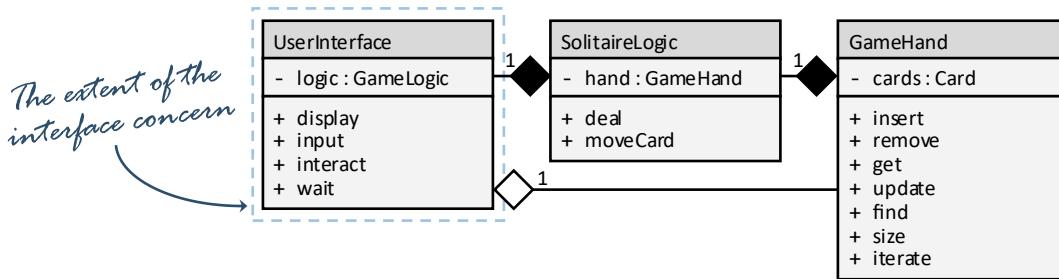


Figure 45.7:
Class diagram of
a simple interface class

The above example illustrates the game of Solitaire. Here, the interface class is **UserInterface**. It contains an instance of the **SolitaireLogic** class which represents all the business logic or rules of the game. Note that the storage class **GameHand** contains a reference to **UserInterface** so it can call **UserInterface::display()** in the case when the hand has changed.

Often more than one version of the interface class must be provided to meet the needs of the client. In the video game example, one interface might provide the player with a first-person perspective of the game; another might show an over-the-shoulder perspective, and another a map top-down perspective.

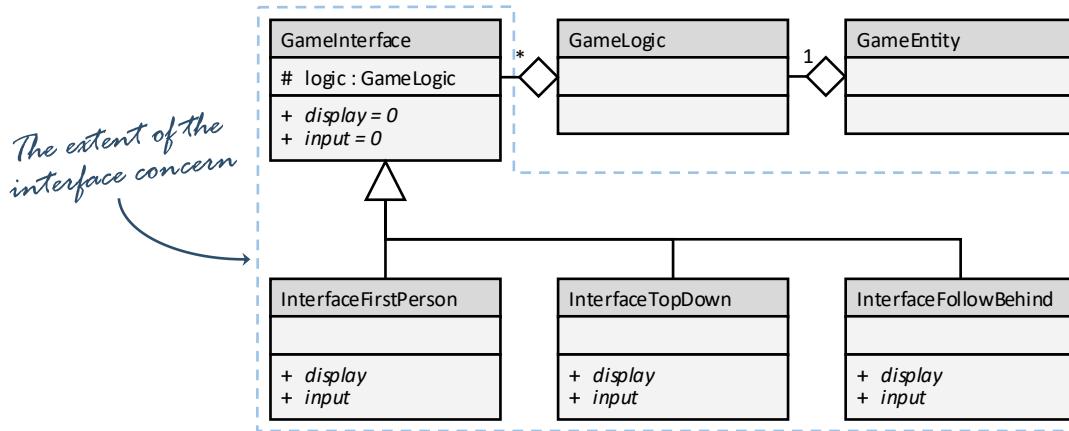
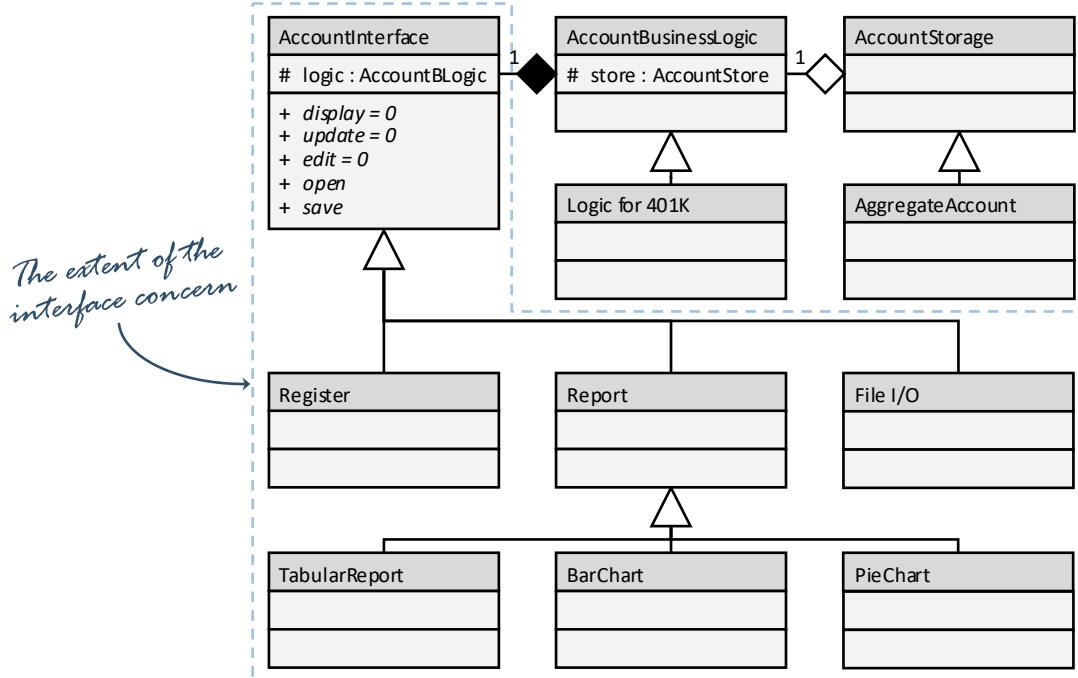


Figure 45.8:
Class diagram of
a complex interface class

There are several advantages of this design. First, we can add a new view to the game with minimal impact on the code. Since the game logic is not changed and the game storage is not changed, we can add a new view without changing any code at all! We only need to add a new derived class to the **GameInterface** base class.

Another advantage is that we can swap views dynamically as the game is going on. If the game is currently in first person view and the user changes to top-down view, we need to instantiate a new instance of **InterfaceTopDown** and connect it to **GameLogic**. Now we have effectively swapped the view while leaving the game state (the **GameEntity** storage class) unaltered.

It is often the case that the interface needs of the client are very complex and multifaceted. Consider, for example, a financial program. There may be many ways to interface with an account: an editable account register, one of several flavors of tabular report, a pie chart, a bar chart, reading and saving to files, and interfacing with the bank. Each of these honors the same business logic and accesses the same storage mechanism, but has completely different presentation needs.



*Figure 45.9:
Class diagram of
an interface concern
consisting of a multi-level
inheritance hierarchy*

From our class hierarchy, we can see that there is one interface division for reports, another for the account register (with several variations), another for interfacing with files (including network interactions with the bank and file-system interactions). On the report front, one would be for tabular reports, another for pie charts, and yet another for bar charts.

To make this complex example more concrete, say the user wished to display a tabular report of a 401(k) account. A **TabularReport** object will be created with a **Logic401K** member variable. This will then be connected to the specific account in **AccountStorage** which happens to be a polymorphic instance of an **AggregateAccount**. Now we have the correct storage class matched with the correct business logic class matched with the correct interface class.

Integration of Views

A single client-facing class needs to encapsulate the data representation, the business logic, and the public interfaces. Thus, once we have successfully separated the various concerns, they need to be reassembled. There are several strategies which can be employed: linear “own” integration, linear “reference” integration, circular integration, and ad hoc integration.

Linear Composition Integration

In linear composition integration, the client communicates directly with the interface, which communicates with the business logic, which communicates with the storage.

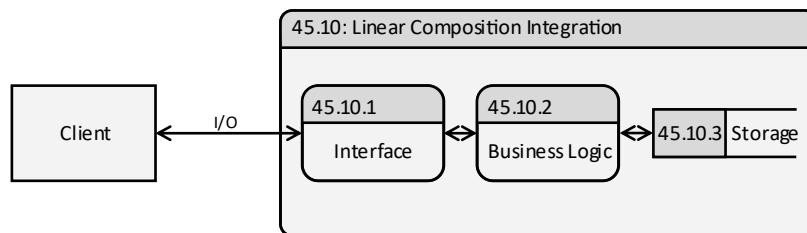


Figure 45.10:
DFD of a linear
integration of the three
concerns

Notice that all requests from the client pass through the interface-logic-storage sequence and all results pass through the storage-logic-interface sequence.

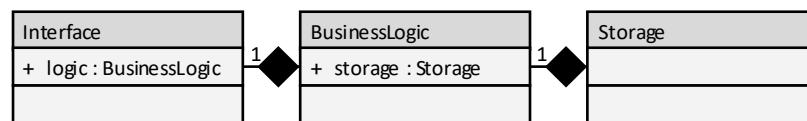


Figure 45.11:
Class diagram of a
linear composition
integration

For the client to make a change to the storage, an interface public method is invoked which, in turn, invokes a business logic public method which, in turn, invokes a storage public method. When the interface and the logic and the storage each represent a hierarchy, then one instance of each is assembled for a given object. For example, the following class diagram represents the game from a certain view (`GameInterface`), containing an element that has certain characteristics (`GameLogic`), and has certain attributes (`GameEntity`).

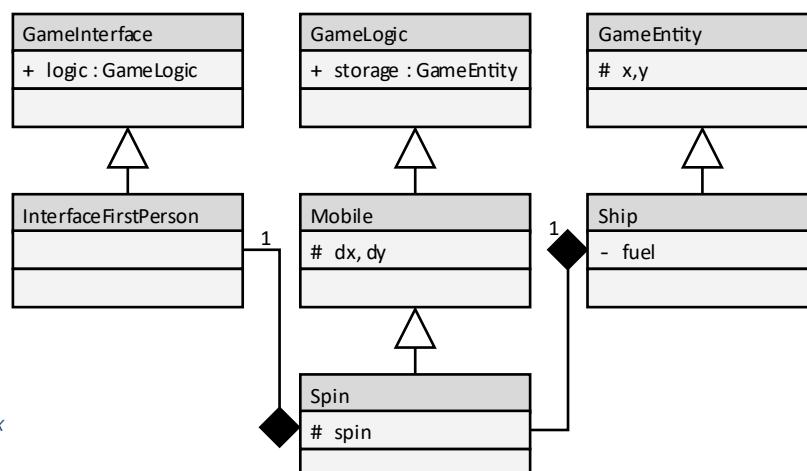


Figure 45.12:
Class diagram of a complex
linear composition
integration

Linear Aggregation Integration

A variation of linear integration occurs when one or more components are managed independently but a reference is held by the others. Here, the DFD looks the same as the composition integration because the information flows in the same linear way.

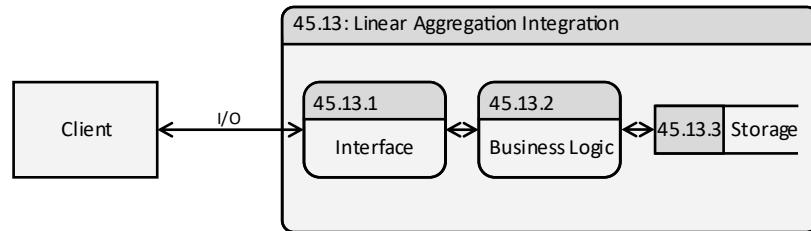


Figure 45.13:
DFD of a linear integration
of the three concerns

In the linear reference integration, each individual storage object is independently created. It is usually contained in one large storage class. Notice the open diamond indicating aggregation rather than composition. This implies that **Interface** contains a reference to **BusinessLogic** but did not create it. Similarly, **BusinessLogic** contains a reference to **Storage** but did not create it. In fact, **Storage** was created by **Collection**.

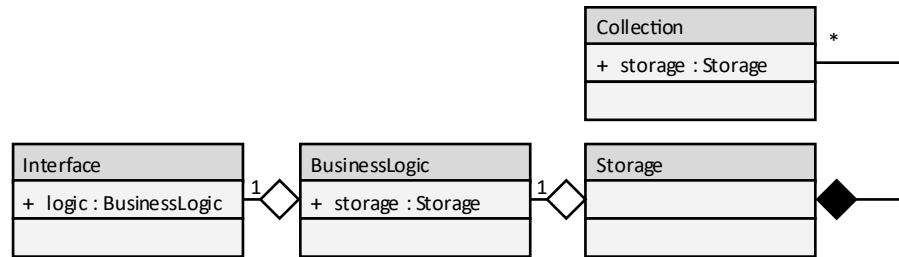


Figure 45.14:
Class diagram of a simple
linear aggregation
integration

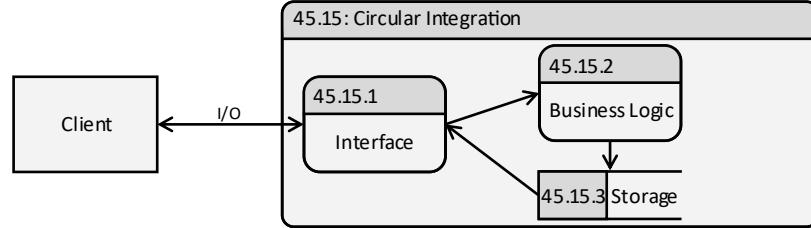
There are several advantages to this approach:

- **Single repository for the entire collection:** The **Collection** class would be an obvious place for file I/O, allowing for managing the entire collection without having to go through the **Interface**. With the advent of the **Collection** class, there is one place to find all instances of **Storage**.
- **No storage duplication:** Using the composition implementation, each **Interface** necessitates the creation of a **Storage** item. If we were working with an account register, then displaying a report of an account would result in a duplication of the entire account register! This is clearly undesirable; there should only be a single copy of the user's checking account in the system.
- **Easier to reassign state:** Imagine a checking account where one set of rules applies for deposits and another for withdrawals. If the user decides to change a deposit to a withdrawal, then a simple reassignment of a **Business Logic** reference in the transaction **Interface** would make this happen. With the "own" implementation, then the **Deposit** object would need to be destroyed and a new **Withdrawal** object would be created. This would be much more difficult than reassigning the reference. In applications where entities change interfaces, business logic, or storage details frequently, this is an important design consideration.

Circular Integration

The circular integration plan involves sending data from the interface to the business logic. It, in turn, sends data to storage. Updates are then sent directly from storage to the interface.

Figure 45.15:
DFD of a circular
integration of the three
concerns



The challenge here is how the storage can tell the interface that something has changed. For example, if the application was a financial report, than an update to the underlying account would necessitate an update to the report. Since the report code exists in the interface, how will it know to redraw? The answer is an event listener.

An event listener is a callback function or object sent from one component to another. When an event occurs (such as an account has been updated), then the callback is invoked. This means the **Storage** class needs to have a reference to **Interface**.

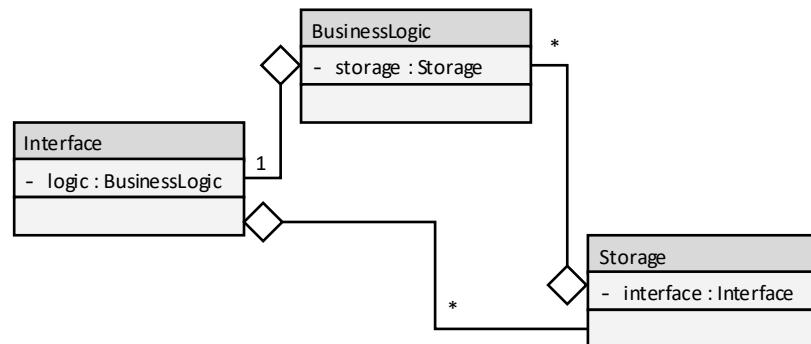


Figure 45.16:
Class diagram of a simple
circular integration

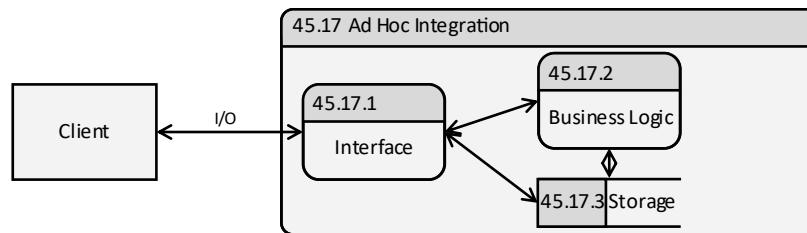
To set this up, three objects are created: an **Interface** object, a **BusinessLogic** object, and a **Storage** object. The **Interface** then contains a reference to the **BusinessLogic** object. This allows the **Interface** to invoke **BusinessLogic** methods. The **BusinessLogic** then contains a reference to the **Storage** object, allowing it to call **Storage** methods. Finally, the **Storage** object contains a reference to the **Interface** object, allowing it to call **Interface** methods.

To see how this works, imagine we are working with **AccountRegisterInterface**, connecting to a **CreditCardLogic** business object connected to a **CreditCardAccount** storage object. The user wishes to indicate she has paid her bill. She invokes the **pay()** method of **AccountRegisterInterface**. This runs through the **CreditCardLogic** to make sure the amount is valid and the money is reported. The **CreditCardLogic** class then adds a transaction to the storage account through the **CreditCardAccount::insert()** method. With at least one account modified, the storage class (**CreditCardAccount**) will notify the **Interface::updateDisplay()** method. This method will notify all interested interfaces that something has changed. Now the newly added transaction is displayed on the user's screen.

Ad Hoc Integration

The final integration strategy is called “ad hoc.” Here, each of the concerns can communicate directly with the others.

Figure 45.17:
DFD of an ad hoc
integration of the three
concerns



For information to be able to pass both ways, each component must have the ability to directly invoke the other. This means each class must contain a reference of the other two. Usually there is a one-to-one relationship between the **Interface** and the **BusinessLogic** (the interface class contains one reference to **BusinessLogic** and the business logic class contains one reference to **Interface**). However, the **Storage** class usually has a many-to-many relationship with both **Interface** and **BusinessLogic**.

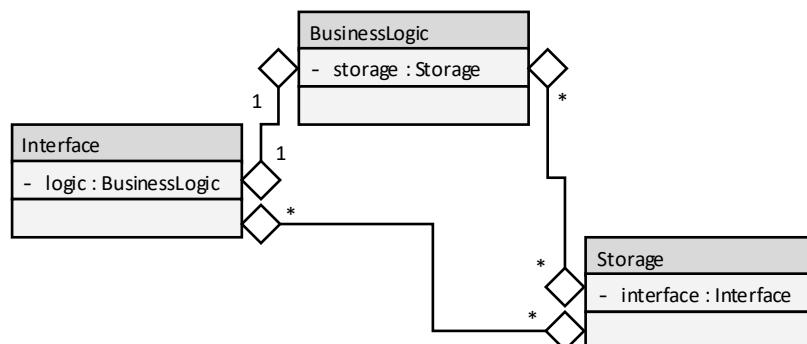


Figure 45.18:
Class diagram of a simple
ad hoc integration

As a rule, ad hoc integration strategies are dangerous. It is easy to bypass the business logic component, making it difficult to offer guarantees that the data in storage conforms to all the necessary rules. Usually ad hoc designs started as linear or circular, but extra connections were added. When this happens, it is worthwhile to carefully scrutinize the extra connections to make sure they are actually needed.

MVC

Perhaps the most famous design pattern is the model view controller (MVC). Here, the model is the storage component, the view is the interface, and the controller is the business logic. The MVC is often called a composite or compound design pattern because it is built from many patterns. It has also been called a recursive or fractal pattern because it is not uncommon for each concern to be built with the MVC. In other words, storage concern (called the model) can be its own MVC.

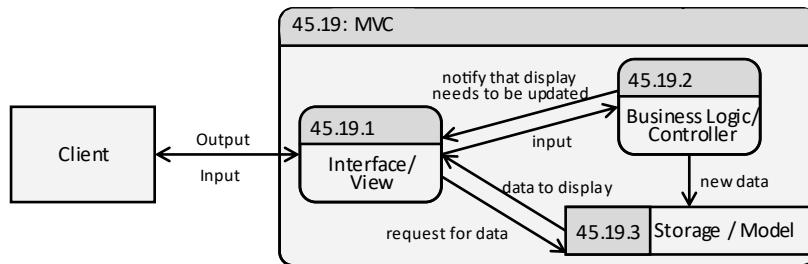


Figure 45.19:
DFD of the MVC
integration of the three
concerns

The first thing to notice is that while this appears to be a variation of an ad hoc integration. It is actually not; it is a circular integration. Data flow from the interface into the business logic, then into the storage, then back to the interface. The other two arrows are requests for information. In other words, data do not flow from the business logic to the interface. Instead, a notification gets sent that the interface needs to update its display. Also, data do not get sent from the interface to the storage. Instead, a notification gets sent that new data are needed. How does this notification process work? It works through the observer pattern.

First, the interface (view) will want to know the storage (model) has changed. This is accomplished by registering the interface as an observer to the storage. When anything in the storage has been changed, then it will run through its collection of observers and call the appropriate method. Notice that there may be many interfaces displaying the data contained in the model. If each one registers an observer with the model, then they will all display current information to the client.

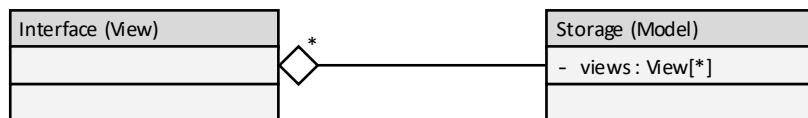


Figure 45.20:
Class diagram of how the
interface connects with the
storage in the MVC

Second, the business logic (controller) will need to send updates to the storage (model). The business logic (controller) will need to contain an instance of the storage (model) which again is usually done through aggregation so more than one business logic component can work with the storage (model).

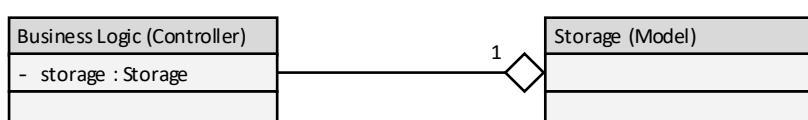
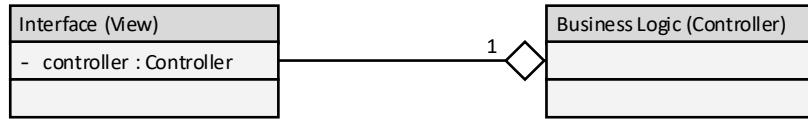


Figure 45.21:
Class diagram of how the
business logic connects
with the storage in the
MVC

Third, the interaction between the business logic (controller) with the interface (view) is the most complicated of the three. The business logic (controller) will need to verify that the input provided by the interface (view) is valid. This means that the interface (view) will need to contain an instance or a reference to the business logic (controller). This will allow the interface (view) to directly ask the business logic (controller) whether the input conforms to policy and perhaps to send it on to storage. Since more than one interface may exist in the system, this is typically accomplished through aggregation.

*Figure 45.22:
Class diagram of how the
interface connects with the
business logic in the MVC*



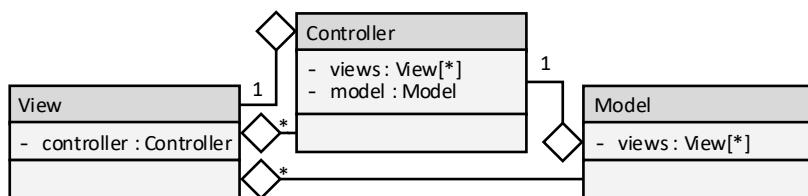
Finally, it is also the case that the business logic (controller) may need to notify the interface (view) of a change. If that notification is the direct result of input provided from the interface (view), then the message is passed through aggregation as described in Figure 45.22. If that notification is the result of data being updated in the storage, then the message is passed through the observer pattern as described in Figure 45.20. All other cases are handled through another observer. In this case, the various interfaces (view) can register an observer with the business logic (controller) to be notified of any system state updates.

*Figure 45.23:
Class diagram of how the
business logic connects with the
interface in the MVC*



The final class diagram showing all the relations between the three components is the following:

*Figure 45.24:
Class diagram
of the complete
MVC design pattern*



When the MVC is created, then all three classes are instantiated. After they are created, the view class is given a reference to the controller class. The controller class is given a reference to the model. All views register appropriate event listeners to the model and to the controller.

Examples

Example 45.1: Time

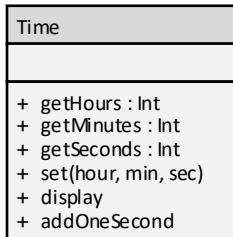
This example will demonstrate how to create a simple class design using separation of concerns.

Problem

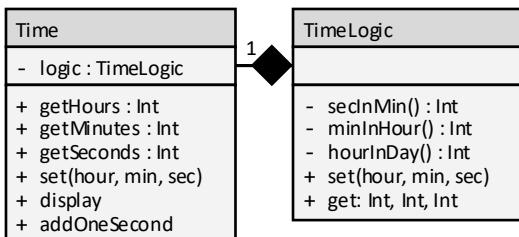
Create a class to represent the time of day using the separation of concerns strategy. The class will need to be able to set the time, display the time, query the hour or minute, and increment time by one second.

Solution

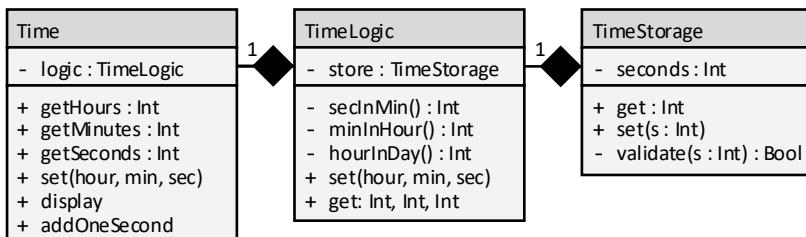
The place to start is with the interface. The most convenient interface we can offer provides a method for each of the client's needs:



Next, we need business logic to know about the number of hours in a day, minutes in an hour, and seconds in a minute. We will use a simple linear composition integration by having the **Time** class contain one instance of the **TimeLogic**.



The final step is to handle data storage. To maximize fidelity, we will keep track of the number of seconds since midnight where any value above 86,400 (seconds in a day) is treated as zero. This will be connected to **TimeLogic** through composition.



Example 45.2: Baseball Cards

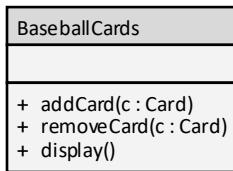
This example will demonstrate how to design a container class using separation of concerns.

Problem

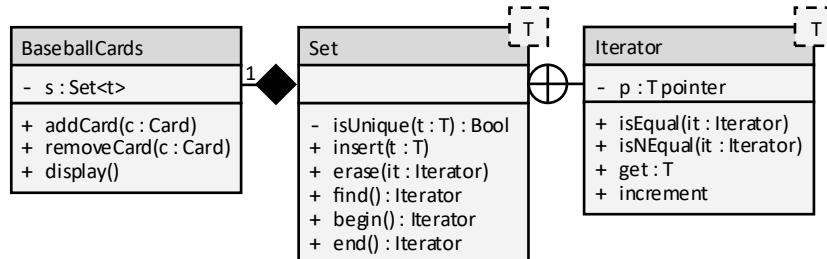
Create a class to represent a collection of baseball cards using the separation of concerns strategy. This class will store the cards where duplicates are not allowed.

Solution

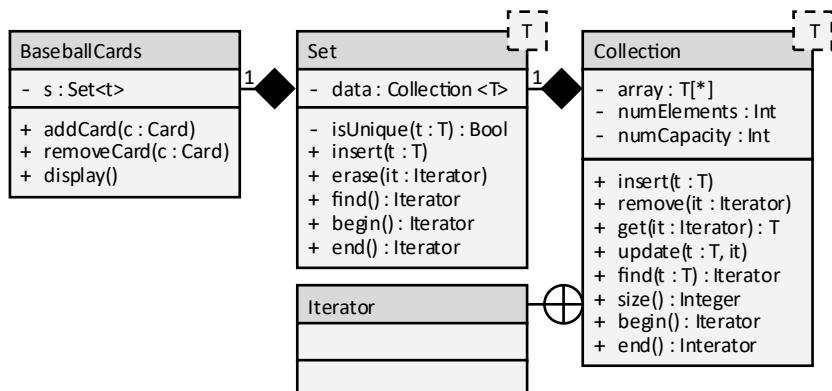
The first step is to identify the public interfaces. A user will want to add and remove cards and display the collection:



Next, the business logic needs to be represented. This is basically the functionality of the set data structure. Here, items are only allowed in when they are unique. To accomplish this, the set public methods are interfaced for a generic data type. Note that the set data structure has an iterator to loop through all the elements.



The final step is to implement the storage. Here, we will use the generic container interfaces, with an array being the underlying data structure.



Example 45.3: List Application

This example will demonstrate how to design a complex class using separation of concerns.

Problem

Design a class to represent the following scenario:

A list application manages a wide assortment of lists: to-do lists, grocery lists, recipes, birthdays, etc. All lists have at least two views: edit or view. Some also have a third: check-off. Some allow duplicates, some require uniqueness, and still others combine duplicates. The lists can be stored in a file, in memory, or on a remote server.

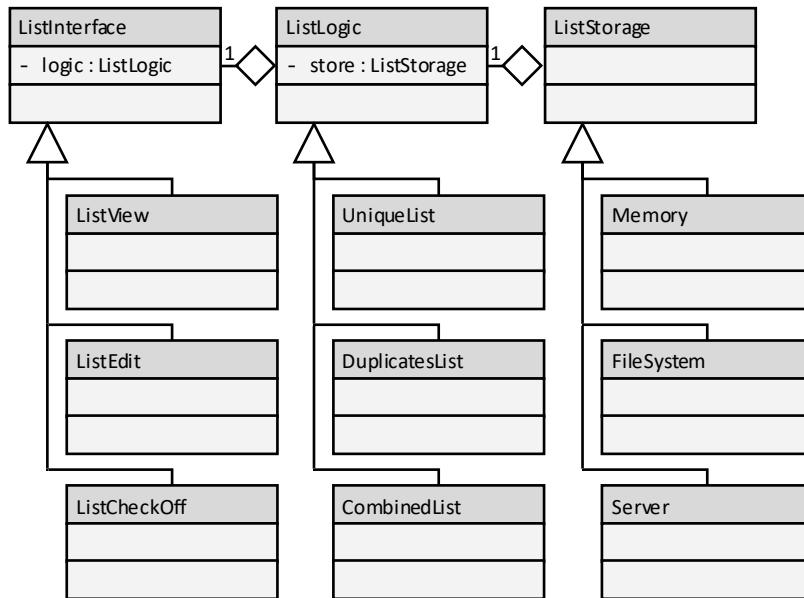
Solution

The interface component of this application has several variations: view, edit, and checkoff. All will be built from the same abstract class and will share the same methods: display, input, prompt, interact.

The business logic component will also have three main divisions: those that require uniqueness, those that allow duplicates, and those that combine duplicates by increasing the count.

The storage component will have three versions: those that are currently stored in memory, those that are in permanent storage in a file, and those that reside on a server.

We will integrate these views using linear aggregation integration so the view and the mode of the view can be easily altered. Note that there are 27 possible combinations with this design ($3 \times 3 \times 3 = 27$).



Exercises

Exercise 45.1: Three Concerns

Consider an application to play the game Monopoly. Classify each of the following components into the three concerns: storage, business logic, or interface.

Component	Concern
The component governing the number of houses required to make a hotel	
The component governing how much money each player currently has	
The component allowing the user to roll the dice	
The component determining whether the player can purchase a property	
The component governing the collection of properties that can be purchased	
The component allowing a user to exit jail if the fee is paid	
The component displaying the current state of the board	

Exercise 45.2: Fact or Fiction

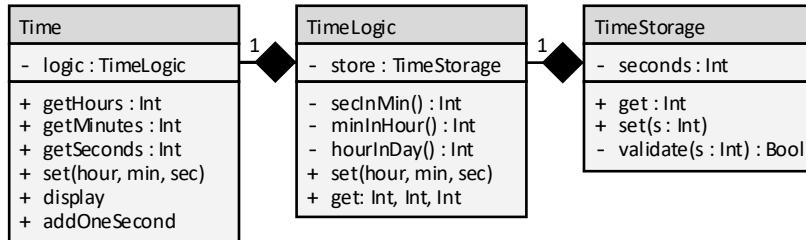
For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
It should never be possible for the storage to send messages to the interface directly.	
The business logic determines how the application will make money.	
Most integration strategies have the interface go through the logic to talk to storage.	
The storage component holds the data for the system.	

Problems

Problem 45.1: Time Using Linear Composition

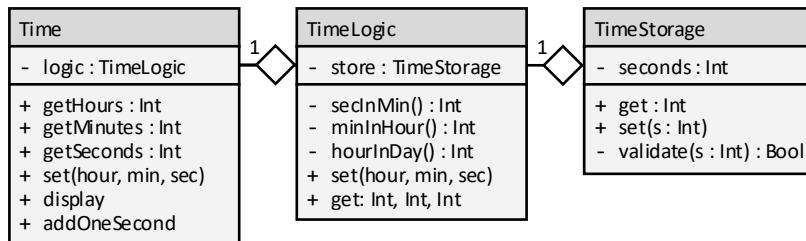
Consider the following `Time` class design from Example 45.1:



Implement this design using the programming language of your choice.

Problem 45.2: Time Using Linear Aggregation

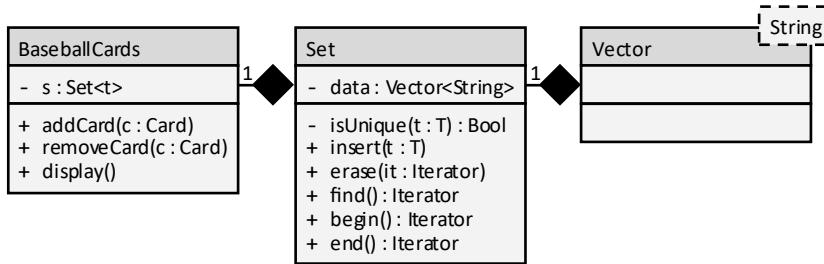
Consider a variation of Problem 45.1 where we will be using linear aggregation instead of linear composition:



Implement this design using the programming language of your choice.

Problem 45.3: Baseball Cards

Consider a variation of the `BaseballCard` class design from Example 45.2:



Note that this variation uses the `vector` class from a standard library. Implement this class using the programming language of your choice, making allowances for the data structures available to you.

Problem 45.4: Date Class

Using the separation of concerns strategy, create a class diagram for the following scenario:

A date class represents the day, month, and year. The epoch (date from which counting will begin) is January 1st, 1752. The client needs to be able to set the day, month, and year and to enquire as to the day, month, and year. Additionally, the client needs to be able to easily increment by a single day.

Provide a brief justification for your storage, business logic, and interface design. Also, briefly justify your strategy for integrating the concerns.

Problem 45.5: Authentication

Using the separation of concerns strategy, create a class diagram for the following scenario:

Create a class to authenticate a prospective user. The user will provide the username and password. The system will then tell the user whether he or she is part of the system. There is also an ability to add a new user (with the associated username and password) to the system.

Provide a brief justification for your storage, business logic, and interface design. Also, briefly justify your strategy for integrating the concerns.

Challenges

Challenge 45.1: War

Consider the card game *War*:

War is played with two or more players. The deck of 52 standard playing cards is evenly distributed to all the players of the game. With each turn, all the players simultaneously lay down the top card from his or her stack. The player with the highest card wins the turn and collects all the cards. If there is a tie, then each player turns one card face-down and another face-up. Again, winner takes all or the tie procedure repeats. The game ends when one player has all the cards in the deck.

Create a design for this game that has a single interface class, a single business logic class, and two storage classes (a “hand” representing a collection of “cards”). Implement this design in the programming language of your choice.

Challenge 45.2: Secret Keeper

Consider a program to store secret information.

This application will have two variations: a graphic user interface consisting of a single dialog, and a textual interface accessible through the command prompt. It will prompt the user for the password. If the password is correct, it will reveal the secret. If it is not, then the program will taunt the user with the message “Your mother was a hamster, and your father smelt of elderberries.” The program will have three data storage strategies: plain text where it is kept in a simple string, simple cipher where the text is shifted by 1 ASCII value, and “complex” cipher where the value is XOR-ed by 0x1010.

Please do the following:

1. Create a class diagram describing a design for this program.
2. Implement the design in the programming language of your choice.
3. Demonstrate all six variations:

Variation

- GUI with no encryption
- GUI with the shift cipher
- GUI with the XOR cipher
- Textual interface with no encryption
- Textual interface with the shift cipher
- Textual interface with the XOR cipher

Command Passing

The command passing collection of design patterns are intended to simplify the coupling between components by giving one component the ability to execute commands in another with the two components knowing as little about each other as possible.

In the early days of motorsports, the driver would show up to the starting line and simply try to finish the race before anyone else. The pinnacle of motorsports today, F1 racing, is somewhat more sophisticated. The driver is supported by a race crew who monitors all the competitors as well as the various systems of the car. To further complicate things, each team has more than one driver. Without proper communication between the race crew and the driver, a team can easily make costly mistakes. To address this problem, race teams have adopted a collection of protocols allowing them to send instructions to the driver. These protocols require much less radio chatter and leave little room for misinterpretation, giving them a competitive edge.

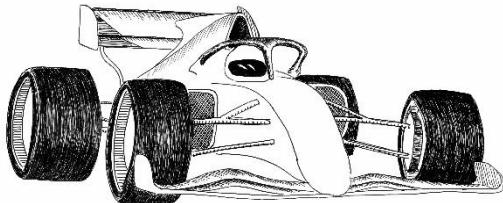


Figure 46.1:
F1 race car

As the race crew needs to send instructions to the driver, it is often necessary to pass instructions or commands between parts of software systems. This can be between two parts of the same component, between components, or even between independent systems. In each command dispatch scenario, there are three parts: the order, the commander, and the executor. The order represents the task which is to be accomplished. The commander creates the order because it wants the task to be completed. The executor receives the order from the commander and carries it out.

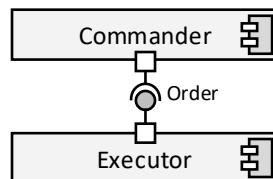


Figure 46.2:
Component diagram of
a commander,
executor and order

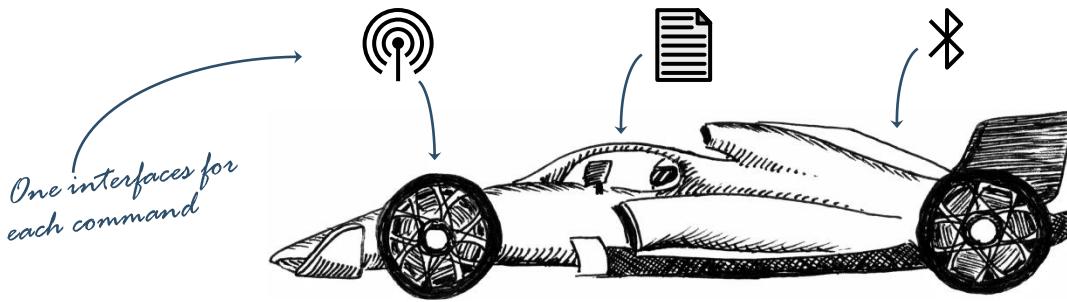
In many ways, this is like the message passing design patterns. There, the purpose is to send data between components. Here, the goal is to send commands. There are four command passing strategies, each characterized by the format in which the order is encoded and sent to the executor.

Design Pattern	Description
Direct Invocation	An order is represented by a function call to the executor.
Delegate Invocation	An order is represented by a function passed as a parameter to the executor.
Encapsulated Invocation	An order is represented by a class sent to the executor. This is also known as the command design pattern.
Interpreter	The order is represented in a rich language to be interpreted by the executor.

Direct Invocation

Direct invocation involves the commander sending orders to the executor by selecting a function provided by the executor to perform the desired operation. This design pattern is called direct invocation because the commander directly invokes an executor function.

The F1 team has decided that the simplest way to change a setting on the car during a race is to build a direct interface between the component and the race crew. Each setting and each driver instruction is communicated through a dedicated interface between the pit crew and the driver. The engine management engineer needs to adjust the fuel/air mixture on every lap, so a Bluetooth connection is established as the car crosses the start/finish line. When the engineer presses a button, then a signal is sent to the car which activates a servo to make the change. The suspension engineer needs to periodically adjust the front/rear brake bias. This is accomplished through a cellular connection between the engineer and an actuator in the car which makes the change. Finally, the strategist (the member of the team in charge of determining and communicating the race strategy) needs to tell the driver to lift the pace (when a competitor car is approaching from the rear) or slow the pace (when fuel economy becomes an issue). This is done by holding up a sign by the pit wall that the driver will read when the car passes by.



The F1 team is utilizing direct invocation in this example because each order is relayed to the car through a dedicated interface. Component diagrams representing direct invocation are easy to spot—there are multiple ports in the executor component and multiple interfaces connecting the components to the commander.

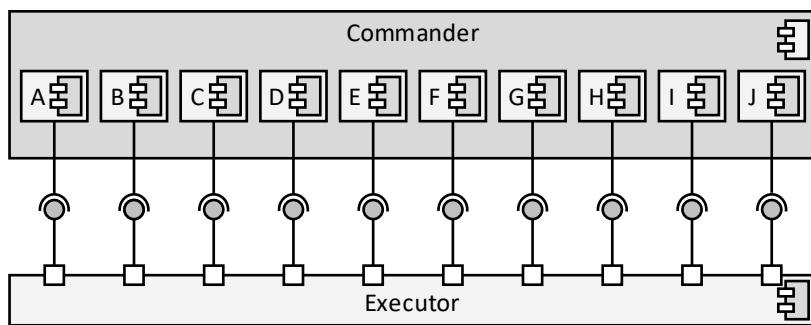


Figure 46.4:
Component diagram
of direct invocation

Observe in Figure 46.4 that the executor has ten ports. These ports are the functions provided by the executor to be called by the commander. This collection of functions is called the application programming interface (API). The commander specifies which order the executor is to perform through the selection of the function. For example, if there are ten distinct orders that the commander may need to call, then there will be ten functions in the executor.

Figure 46.3:
F1 team exhibiting
direct invocation
command passing

Implementation

The direct invocation design pattern is the simplest command passing design pattern to implement. If the commander and the executor exist in the same process space, then creating a direct invocation API is as simple as placing the public functions in a header so they are visible to all parts of the program. For example, the following code from a video game allows any part of the program to draw elements on the screen.

C++

```
void drawCrewDragon(const Point& center, double rotation);
void drawSputnik(const Point& center, double rotation);
void drawGPS(const Point& center, double rotation);
void drawHubble(const Point& center, double rotation);
void drawShip(Point& center, double rotation, bool thrust);
void drawEarth(const Point& center, double rotation);
void drawStar(const Point& point, unsigned char phase);
```

Figure 46.5:
C++ code of a direct invocation API

Here there are seven functions in the draw API, each using a consistent naming convention and each requiring a similar set of parameters. This consistency makes it easy for the commander to utilize the API.

When the commander and the executor are in separate process or address spaces, the code is more complicated. This type of direct invocation is called a remote procedure call (RPC) or remote method invocation (RMI). The details of how this works depends on the programming language and the environment on which the code is run, but the steps are the same. First, it is necessary to import a library supporting RPC or RMI calls. Second, the class or function is marked as being RPC or RMI. Finally, the function or class is written as it normally would be.

Java

```
package executor.port;

import java.rmi.*;
public class Server implements Executor {
    public Server() {}
    public String port1() { return "port1"; }
    public String port2() { return "port2"; }
    public static void main(String args[]) {
        try {
            Executor stub;
            stub = (Executor)UnicastRemoteObject.exportObject(
                new Server(), 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("port1", stub);
            registry.bind("port2", stub);
        } catch (Exception e) { }
    }
}
```

Figure 46.6:
Java code exporting
remote method invocation

This fragment of Java code creates two ports through RMI: `port1` and `port2`. Another component outside the process space can access `Executor.port1()` and `Executor.port2()`.

Recommendations

The direct invocation design pattern is the simplest command passing design pattern. It is therefore useful for small problems with minimal complexity.

Best Practice 46.1 Use direct invocation when there are a small number of orders

If there are a small number of orders for the executor to carry out, then direct invocation can be implemented with minimal lines of code. This makes the direct invocation design pattern ideal for small and simple problems. Direct invocation has the benefit of isolating the subcomponents of the system. If one function in the API needs to be changed, the other functions are not affected.

Best Practice 46.2 Use direct invocation when quality assurance is an overriding concern in the system design

When a change is made in the system, all related comments need to be revalidated. Thus, the more interrelated the system, the larger the number of test cases that need to be verified to provide the necessary quality assurances. Direct invocation has the benefit of allowing for a relatively small test burden. Adding new connections has minimum impact on the rest of the system. If the new connection does not interfere with an existing one, the other systems do not need to be verified when a new connection is added. There are several downsides to direct invocation. They all have to do with the difficulty of maintaining many independent interfaces.

Best Practice 46.3 Avoid direct invocation when it is likely that new interfaces will need to be added to the design

While it is easy to create a single connection between the commander and the executor using direct invocation, the second connection costs as much as the first. Thus, adding capabilities to the system can be prohibitively expensive.

Best Practice 46.4 Avoid direct invocation when system-wide features might need to be added to the design

Because each connection is independently engineered, it is very difficult to manage all the connections. Adding system-wide services such as security (to prevent eavesdropping) or logging (recording all the orders relayed to the executor) can be complex and time consuming. Direct invocation requires each connection to be independently studied and designed. When the size of the API is large, this can be prohibitively expensive.

Best Practice 46.5 Maximize uniformity when designing a direct invocation API

Most of the disadvantages of direct invocation can be mitigated by utilizing consistent API names, behaviors, and data types. Little things such as using the same representation for commonly used constructs go a long way towards making the interface more maintainable.

When Microsoft was preparing to release the first version of C# and the accompanying .NET framework in 2001, it became apparent that the naming conventions of the .NET libraries were not consistent. Every module was developed with a different design philosophy and the names were seemingly chosen at random. This made it very difficult for a developer to find the name of a function without reading the documentation. To address this shortcoming, the entire release schedule was postponed six months. During this time, each data type and library was reexamined. When finished, the .NET framework became the benchmark for how APIs are to be designed. A programmer can predict the name and functionality of an API without needing to read the documentation.

Delegate Invocation

Delegate invocation is the process of passing a function to another component which will execute the function at the appointed time and in the appointed context. These functions are called delegates. In politics, a delegate is an individual who represents another. In many ways, this is how delegate invocation works. A function is created representing the action that the commander needs to be performed and that function is passed to the executor. The design pattern is called delegate invocation because orders are represented as delegates.

The F1 team manager is frustrated with the direct invocation method employed earlier because it requires too much work to create a new connection. Instead, the manager adopts a new method. Every time the driver gets in the car to start a session, the driver is given a stack of cards. Each card contains instructions about how to configure and drive the car. The cards are created in the preseason, providing a unified interface through which all commands are sent to the driver. When the crew needs to send a command to the driver, the appropriate card is selected from the stack and placed on the driver's hands just before a session.

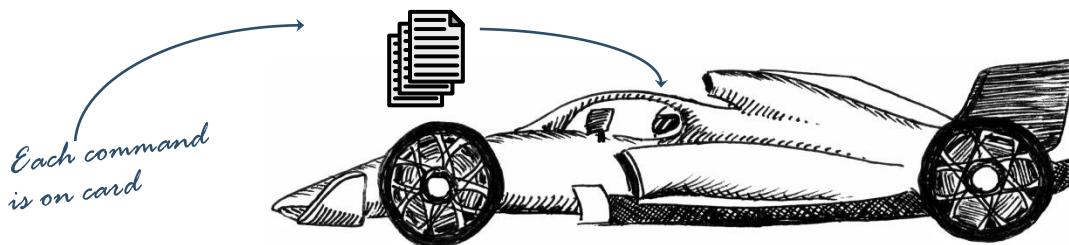


Figure 46.7:
F1 team exhibiting
delegate invocation
command passing

The F1 team is using delegate invocation. Observe that orders are encoded in delegates (the cards) created by the commander (the pit crew) which are to be invoked by the executor (the driver). When one or more delegates (cards) are sent to the executor (driver), they are invoked at the appropriate time.

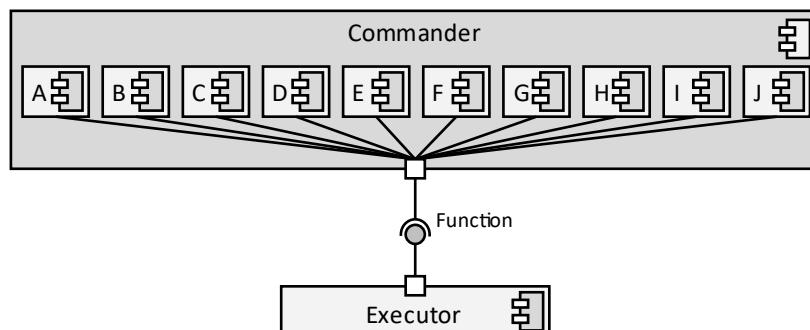


Figure 46.8:
Component diagram of
delegate invocation

Using delegate invocation, the commander is responsible for creating delegate functions representing the order that the executor will invoke. Since this list of delegates needs to be known at compile time, the commander port is usually responsible for selecting an appropriate delegate to match a subcomponent's request for action. Once the delegate is selected, it is then passed to the executor through a single port.

A crucial part of the delegate invocation design pattern is that the executor does not need to know anything about the order. The executor needs only to accept the delegate and then invoke it at the appropriate time.

Implementation

The key component of the delegate invocation design pattern is the process of passing a function to another function as a parameter. Recall from Chapter 19 that most languages provide a mechanism where functions can be passed this way. These are called callbacks, high-order functions, and delegates.

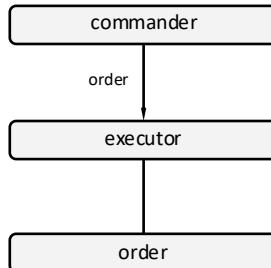


Figure 46.9:
Structure chart of
delegate invocation

For example, consider the scenario where the client would like to do one of three things: display the data on the screen, send the data to a server, or save the data to a file. Each of these three actions is represented with a function: `display()`, `send()`, and `save()`. The commander creates a delegate called `order` which will be one of these three functions. Each of these three functions must be created at compile time; it is impossible to improvise a command at runtime.

```
Pseudocode
commander()
option ← getOption()

SWITCH option
CASE 1
    order ← display
CASE 2
    order ← send
CASE 3
    order ← save

executor(order)
```

Figure 46.10:
Pseudocode of a
commander issuing a
direct invocation order

Notice how the commander does not execute the `display()`, `send()`, or `save()` function. Instead, a reference to the function is retained in the `order` variable. In pseudocode, we represent this by not putting the parentheses after the function name as we do when the function is executed. Other languages use a different syntax to represent this.

Once the order is specified, it is passed to the executor. The executor then invokes the order at the appropriate time.

```
Pseudocode
executor(order)

... code removed for brevity ...
order()
... code removed for brevity ...
```

Figure 46.11:
Pseudocode of an
executor invoking an order

Sending Context to the Order

There are two types of delegates. The first type requires context from the commander and the second requires context from the executor.

Commander Context

In many direct invocation scenarios, the order needs to have information from the commander to function. This information is called the context. For this to work, the commander sends both the order and an instance of itself to the executor. When the executor invokes the order, it passes the commander instance as a parameter.



Figure 46.12:
Structure chart of
delegate invocation
with commander context

For example, imagine a video game consisting of two components: the game state (commander) and the drawing engine (executor). When the drawing engine is initialized, the game asks to be notified when it is time to draw the state of the game on the screen. It does this by creating a delegate called `draw()`. The game engine periodically invokes the `draw()` function when the screen needs to be refreshed. In order for `draw()` to work, it needs to have a reference to the game state. Thus, `draw()` requires a game instance (commander) as a parameter.

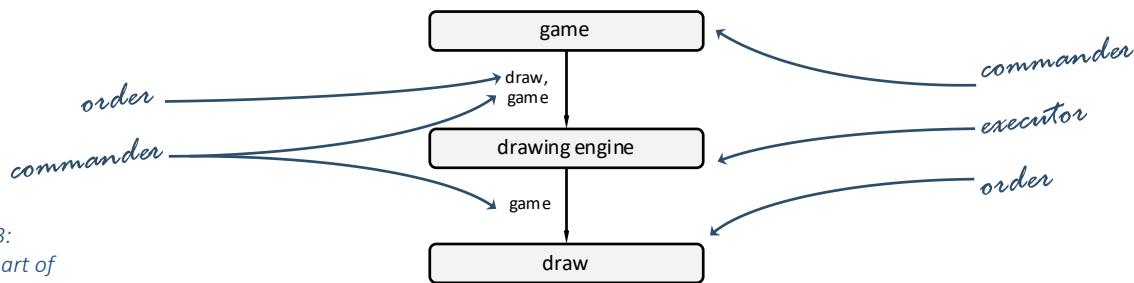


Figure 46.13:
Structure chart of
an implementation of
delegate invocation

Notice how `commander()` passes two parameters to `executor()`: both the function to be called (`order`), and an instance of itself (`commander`). When `executor()` invokes the `order()` function, it passes `commander` as a parameter. This style of delegate invocation is called callback invocation and the delegate itself is called the callback. The name comes from the fact that the executor calls back to the commander.

```
Pseudocode
executor(order, commander)
    ... code removed for brevity ...
    order(commander)
    ... code removed for brevity ...
```

Figure 46.14:
Pseudocode of an executor
invoking an order with
commander context

Executor Context

The second type of delegate requires context from the executor. In other words, the commander is telling the executor to do something with the executor's own data.



Figure 46.15:
Structure chart of
delegate invocation
with executor context

For example, consider a financial program containing a ledger representing the transactions of a credit card. The commander in this scenario is the user interface. This could be toolbar buttons, a dialog, or a variety of other interface tools. The executor is the ledger object. The user interface (commander) can tell the ledger (executor) to back up the data, generate a bar chart report, or generate a pie chart report. Each of these operations (`backup()`, `barChart()`, and `pieChart()`) requires a `ledger` object so it has the data necessary to perform its action.

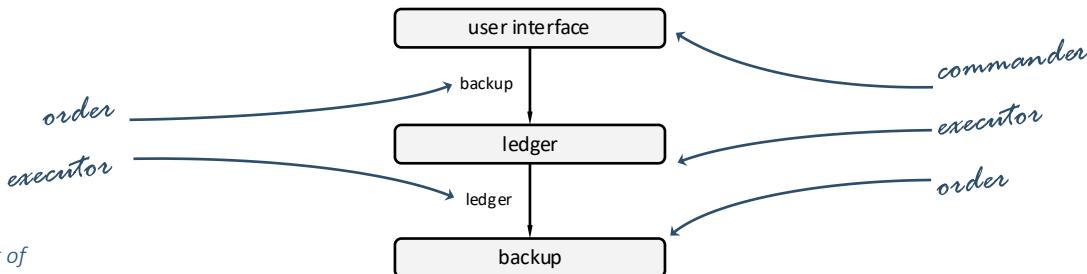


Figure 46.16:
Structure chart of
an implementation of
delegate invocation

Notice that the commander serves only to initiate the action. The commander is not required to complete the action. Instead, the commander is telling the ledger what to do with the ledger's data.

Pseudocode
<pre> executor(order) ... code removed for brevity ... order(this) ... code removed for brevity ... </pre>

Figure 46.17:
Pseudocode of an executor
invoking an order with
executor context

In some instances, the entire executor context is needed by the order. In most cases, however, a small subset of the context is required. For example, the click event for a graphics engine (executor) may accept a callback (order). The client (commander) will need to call back event (order) to know the screen position of the click. In this case, the graphics engine (executor) will just send coordinates (executor context) to the callback (order).

Order Invocation Strategies

In most instances, the executor invokes the order at the earliest convenience. However, there are two cases when this might not be the best design. Delegate invocation facilitates queuing orders and redoing orders.

Queuing

In the simplest terms, a queue is a line where the first to enter are the first to leave. Queuing is the process of a recipient accepting input that may arrive unpredictably by placing the input in a queue. This input is pushed in a queue when it is received and the recipient pops input from the queue when the recipient is prepared to handle it.

Queuing is easy to implement using the delegate design pattern because each command is a single function. The executor can push and pop functions in the same way one would push and pop any other data type.

Pseudocode

```
Executor.acceptOrder(order)
queue.push(order)

Executor.invokeOrders()
WHILE queue.size()
    order = queue.top()
    order()
    queue.pop()
```

Figure 46.18:
Pseudocode of an
executor implementing
a command queue

An executor that queues orders is called a command queue. Notice that the commander does not need to be aware that the executor queued the orders.

Redo

Redo is the process of repeating an action one or more times. Most editing applications have redo functionality, accessible through the Ctrl-Y shortcut key on Windows computers and Command-Shift-Z on Macintosh computers. It is possible to implement redo with the delegate invocation by simply executing the order more than once. In the following example, the commander's order will be executed five times:

Pseudocode

```
executor(order)
FOR num ← 1 ... 5
    order()
```

Figure 46.19:
Pseudocode of an
executor implementing
redo functionality

As with queuing, the decision of the executor to invoke the `order()` function more than once is completely shielded from the commander.

Most user-facing applications pair redo functionality with undo functionality. Undo functionality requires a more sophisticated command passing strategy. Encapsulated invocation in the following section will present a solution to that.

Recommendations

Best Practice 46.6 Specify the delegate function signature carefully

A function signature is the return type of a function and the data types of each of the required parameters. If a caller is to invoke a callee, the caller must use the callee's name and honor the callee's function signature. For example, if the callee expects three parameters and the caller provides only two, a syntax error will result.

The author of the delegate interface needs to be extremely careful when specifying the delegate function signature. The signature must be rich enough to support all the commander's needs. It also must not require any information that the commander is not able to provide. When all the commanders are known at code creation time, this is simple. Things get quite a bit more complicated when the executor component is written long before the commander component.

For example, consider a sort function that orders an array from the least to the greatest. Many sort functions allow for a callback to specify sequence. The signature for the comparison function in C takes two void pointers as parameters and returns an integer. Here, negative implies the first argument is less, zero if the two arguments are equal, and positive if the first argument is greater.

```
C  
int comparison(const void *, const void *);
```

Figure 46.20:
C code of a delegate
function signature

This callback function is sufficiently general that it can work with any data type. The function can also work with any sorting algorithm. By carefully creating the delegate function signature, the API author made the connection between the commander component and the sort component easy to use.

Best Practice 46.7 The executor should know nothing about the delegate

To decouple the commander from the executor, it is important that the executor makes no assumptions about the commander. The recipient is only responsible for accepting the callback and executing it at the appointed time.

For example, the graphics layer GLUT accepts a callback from the client that is to be tied to a specific event. The GLUT engine has no idea what is contained within the callback, especially because the callback was written long after the GLUT library was completed.

Best Practice 46.8 The commander should know nothing about how the executor will execute the callback

Once the commander hands the executor the callback, it is up to the executor to decide when and how to call the callback. Any assumption the commander makes about the executor serves to increase the coupling between the two components.

Encapsulated Invocation

Encapsulated invocation is the process of passing an object to another component. This other component will execute a predetermined method at the appointed time and in the appointed context. This design pattern is called encapsulated invocation because all the information necessary to perform the action is encapsulated in the sent object. It is also commonly known as the command design pattern.

The F1 team has been hard at work during the offseason preparing for the first race. A key part of their strategy is to improve the ways that the race team can send orders to the car and the driver. Building off the delegate invocation technique used previously, the team creates a new version of the cards which provide a space for the pit crew to augment the instructions with relevant data. For example, the old system required ten separate cards representing the various brake bias settings. The new system has a single brake bias card with space for the suspension engineer to record the new setting. This simple change has resulted in far less cards for the team manager to handle and provides far more flexibility in dealing with challenging race situations.

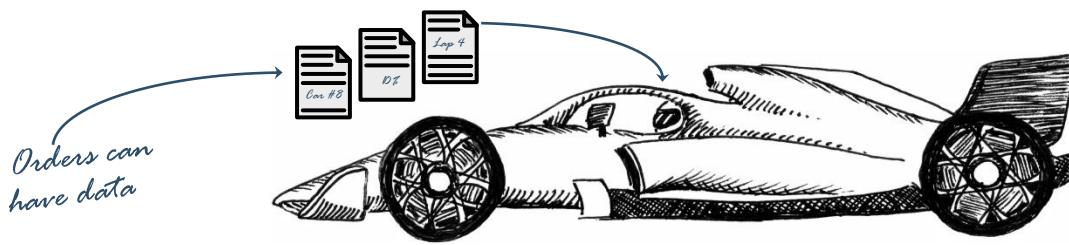


Figure 46.21:
F1 team exhibiting
encapsulated invocation
command passing

The F1 team is using encapsulated invocation to transmit orders from the pit crew to the drivers. Encapsulated invocation is a command passing pattern where orders are encoded in objects containing both a method and any data necessary to complete the task.

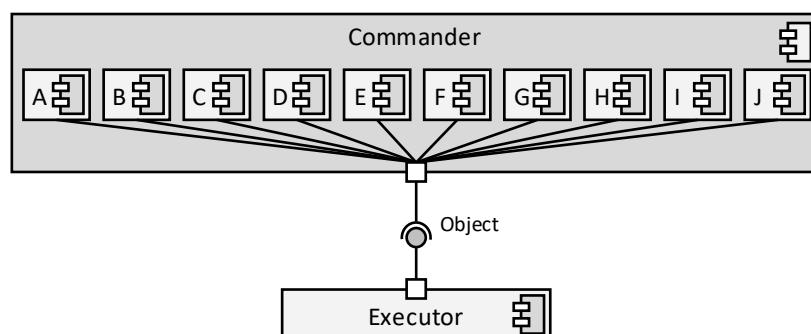


Figure 46.22:
Component diagram of
encapsulated invocation

If the commander's order is simple and does not require any data to accompany it, then the object sent to the executor is no different than the delegate used for delegate invocation. However, using an object to encode the order enables the commander to include an arbitrary amount of data. This provides far greater expressive power. This power does not make any additional demands on the executor; the executor only calls a single method in the order and relies on the order to be able to handle the rest.

Implementation

The encapsulated invocation design pattern is essentially an object-oriented design. It consists of several classes, the most important of which is the command class containing all the information necessary to carry out an order.

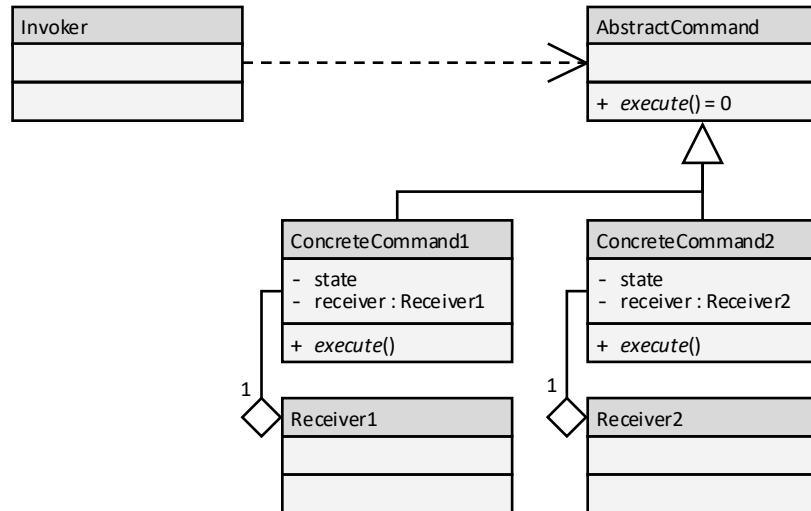


Figure 46.23:
Class diagram of
encapsulated invocation

There are several parts to the encapsulated invocation design pattern:

1. **Abstract Command**: This base class defines the interface through which a command is executed. It has no member variables and only a single pure virtual function: `execute()`. When `execute()` is called, then the underlying command is performed.
2. **Concrete Command**: Each concrete command represents an action or order to be carried out. The concrete command class is aware of the details of the action and contains all the information necessary to perform it. If the action requires parameters or configurations, then those are stored in the concrete command's member variables. When a concrete command is created, all the information necessary to perform the action must be given to the object at that time.
3. **Invoker**: The invoker (executor) is the class that initiates a request for a command to perform its action. The invoker only has access to the `execute()` command, meaning it has no special knowledge of how the action is performed.
4. **Receiver**: Each concrete command serves as an interface to the class that carries out the action. Note that if the receiver action is trivial, then that code may exist in the concrete command class. Otherwise, the concrete command simply formulates a call to the receiver. Because there is no guarantee that any two receivers are related, there is no unified receiver base class.

For a commander to send an order to an executor, the first step is to create a concrete command object. This may involve configuring the object with the information necessary to carry out the order. Once this object is created, it represents the commander's order. Next, the commander sends an order to the executor through a shared interface. Upon receipt the executor invokes the order's `execute()` method.

Order Invocation Strategies

The encapsulated invocation design pattern facilitates doing many things which were previously difficult. These include queuing, redo, undo, and logging. Queuing and redo are accomplished as they are with delegate invocation. The only difference is that queuing stores command objects in a queue rather than command delegates. Similarly, redo invokes an order object's `execute()` method multiple times rather than simply invoking the `order()` delegate multiple times. With the extra expressive power possible with a class, it is also possible to implement logging and redo.

Logging

Logging is the process of recording of all that took place on a system during a period of time. Sometimes the logging requires a summary of the commands that were executed. In this case, a logging method is added to the command classes.

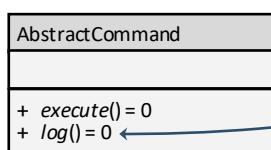


Figure 46.24:
Class diagram of a log
pure virtual method

It is often the case that the system needs to be able to replay the logs from the state where the logging began, thereby recreating the work from a given point. Though logging is generally a difficult process to implement, it is straightforward when the encapsulation design pattern is utilized. The algorithm is this: with each command to be executed, the command object is pushed onto the log before it is executed.

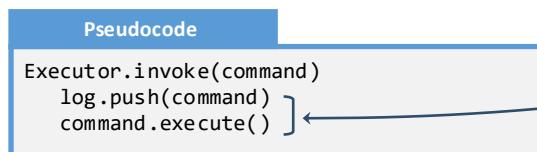


Figure 46.25:
Pseudocode of adding a
command to a log

The log can be replayed by iterating through the log data structure from the beginning. The `execute()` method is then evaluated with each instance.

Undo

Undo is the process of reversing the previously performed action. Most editing applications have undo functionality, accessible through the Ctrl-Z shortcut key on Windows computers and Command-Z on Macintosh computers. Implementing undo with the encapsulated invocation is slightly more complicated than redo or logging because each command object needs to support an `unexecute()` method. This `unexecute()` method performs the opposite action of what the `execute()` method does. The implementation details of `unexecute()` which depend on the action that is to support undo. Note that not all actions are undoable! For example, a print action may send a document to the printer. It would be difficult to un-print a document! With the implementation of `unexecute()` methods, supporting undo is a matter of pushing command objects into an undo stack and, when the undo action is selected, popping the object off the stack and calling the `unexecute()` method.

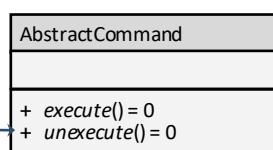


Figure 46.26:
Class diagram of an
unexecute method

Recommendations

A key aspect of the encapsulated invocation design pattern is that an order can be fully represented in a concrete command object. This means the commander must be able to configure a command object to represent its intent.

Best Practice 46.9 Make it easy to configure a command object to represent the action to be performed

Unlike delegate invocation where the commander simply selects the delegate to represent the action that needs to be performed, command objects may contain the data that accompanies the order. Concrete command authors should create constructor and mutators to make this process as easy as possible. The convenience and fidelity metrics of encapsulation design should be carefully considered.

Best Practice 46.10 If commander context is needed, add it to the command object rather than having the executor manage it

It is often the case that an order needs information from the commander to function properly. Using delegate invocation, this can only happen by asking the executor to pass commander context onto the order. The encapsulate invocation design pattern can make this easier on the executor. Because classes have member variables, it is easy to add any needed commander context to the command object.

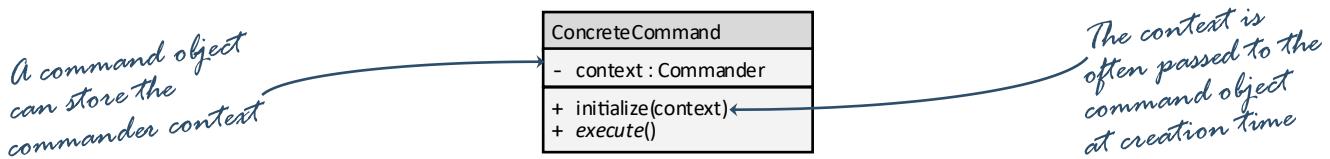


Figure 46.27:
Class diagram of
commander context in a
concrete class

Unlike delegate invocation where the `execute()` method receives the commander context and invocation time, encapsulation invocation sends the context to the concrete command object at object instantiation time so the `execute()` method can access the context as a member variable.

Best Practice 46.11 The executor should not need to know anything about the command object it invokes

Aside from the name of the abstract command base class and the name of the `execute` method, the executor should not need to know anything about the commands that are executed. This serves to give the commander great latitude in the type of commands which are created. It also gives the executor latitude to implement a variety of order invocation strategies such as queuing and logging.

Interpreter

The interpreter design pattern is a methodology of sending commands between components through a shared language. The commander's intent is encoded into an order written in this shared language, and the executor interprets the order into a collection of actions. This communication style results in document coupling between the components.

Back to our F1 team, the strategist is frustrated with the communication strategies adopted by the team up to this point. Race strategies are far too complicated to encode on simple instruction cards. A typical race plan may include conditionals (such as "If a car tries to pass you on turn 4, take it wide to block"), loops (such as "After lap 8, push the pace until the tires give out"), and sub-plans (such as "If there is a full-course yellow, execute plan C").

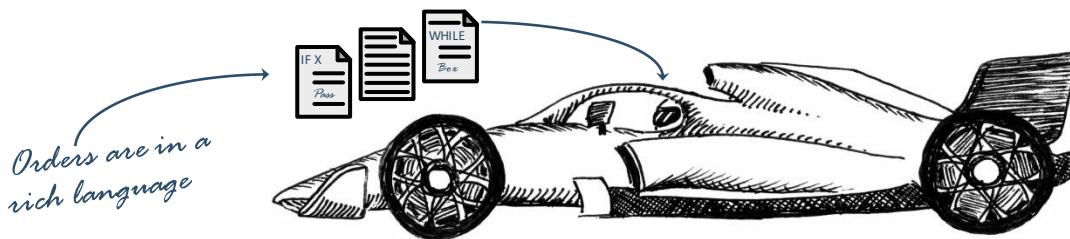


Figure 46.28:
Interpreter design pattern

The key to making this strategy work is to ensure the race team and the drivers share the same language. To address this challenge, the team manager first identifies all the actions that the driver will have to perform. Each command is an action and the associated data accompanying the action. Next, the manager identifies all the conditions that could be attached to an action or a loop. Finally, the manager packages actions which are related to a high-level strategy. Once all aspects of this language are identified, then the manager teaches the race team and the drivers to understand and use this language.

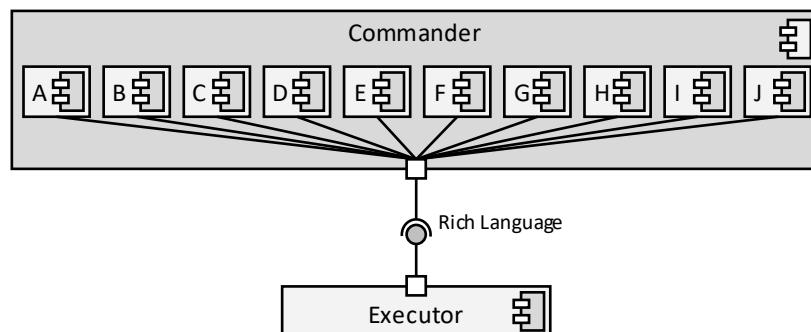


Figure 46.29:
Component diagram of the
interpreter design pattern

The F1 team is using the interpreter design pattern because orders are encoded in a rich language by the commander (the pit crew) which are to be interpreted and invoked by the executor (the driver).

Implementation

The interpreter design pattern is a command pattern involving sending a rich command language from the commander to the executor which will then be interpreted into specific actions. The command language is divided into two categories: terminal expressions and nonterminal expressions. The terminal expressions are simple commands translating directly into an executor function call. The nonterminal expressions are constructs allowing multiple expressions to be combined. The class diagram for the interpreter design pattern is the following:

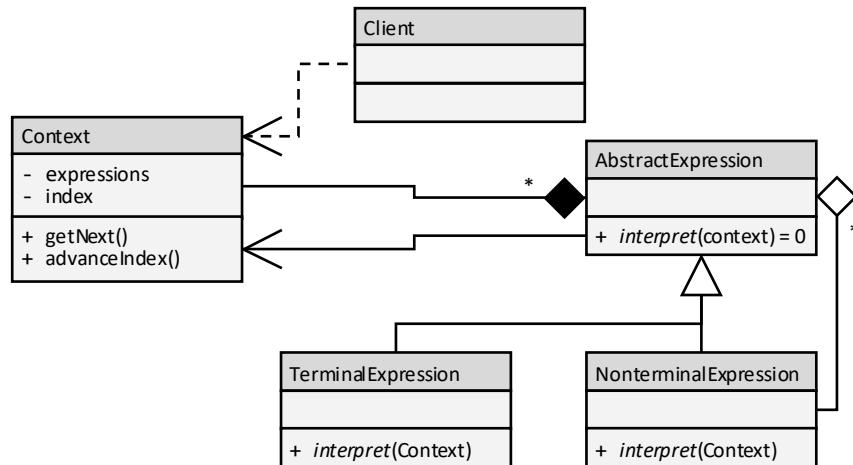


Figure 46.30:
Class diagram of the
interpreter design pattern

There are several parts to the interpreter design pattern:

1. **Context**: A class representing the message to be interpreted. The message can be stored as a string within the context, or it could be stored in a richer language encoded in JSON or a similar format.
2. **Abstract Expression**: An abstract class consuming part of the context. The abstract expression has a single pure virtual function: `interpret()`. This function takes a context as a parameter and consumes part of it. The most common way to do this is to include an index in the context diving the part of the context has already been executed from the part which remains.
3. **Terminal Expression**: A concrete class implementing the `interpret()` method. Each call to a terminal expression's `interpret()` method should consume part of the context. Note that there may be many terminal expression classes, one for each component of the context language. Terminal expressions look a great deal like a concrete command in the encapsulated invocation design pattern.
4. **Nonterminal Expression**: A concrete class that combines one or more terminal expression and other instructions. For example, a nonterminal expression could represent a conditional. This nonterminal expression would execute a terminal expression only if the condition is met.

The key feature of the interpreter design pattern is the context. This contains the richness of the command language in a way that is easy for the terminal and nonterminal expressions to consume.

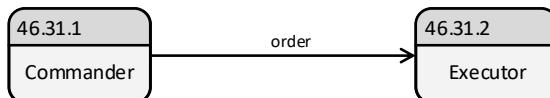
Order Invocation Strategies

Direct, delegate, and encapsulated invocation involve components communicating in the same process or address space for the most part. Interpreter command passing can occur many other ways. There are three invocation strategies that involve the interpreter design pattern: synchronous invocation, remote invocation, and delayed invocation.

Synchronous Invocation

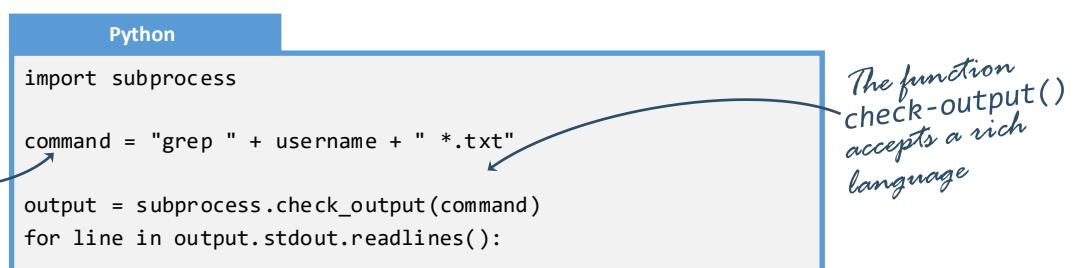
Synchronous invocation involves one component connecting directly with another component. Here, the sending and receiving port are connected with nothing mediating the exchange.

Figure 46.31:
DFD of synchronous
invocation



Synchronous invocation almost always involves a function call between components where the parameter is the order. However, when the components exist in different address spaces, they can occur through an RPC or RMI interface. An example of this interface is the `subprocess` library allowing an application to interact with the operating system (OS).

Figure 46.32:
Python using synchronous
invocation to send
commands to the OS

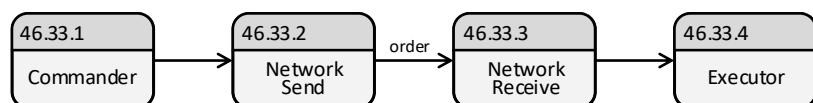


In this scenario, the commander is the application utilizing OS resources. The order is encoded using the command language of the OS. The executor is the OS itself. Since commands are sent to the OS through a function call which are immediately executed, this is a synchronous invocation scenario.

Remote Invocation

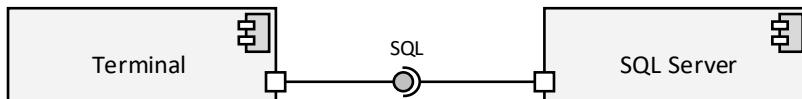
Remote invocation involves two components communicating through a network connection. Here the commander's orders are encoded in the command language and packaged in a network packet. The executor receives the message from the network, interprets the orders, and executes them.

Figure 46.33:
DFD of the interpreter
design pattern using
remote invocation



The F1 team stores their system logs in a database. A database engineer working with this database uses structured query language (SQL) to interact with the system logs. Here **Commander** is both the engineer and the terminal used by the. The order is encoded using SQL. The executor is the database and the server that is hosting the database.

*Figure 46.34:
Component diagram of a
SQL connection between a
terminal and a server*

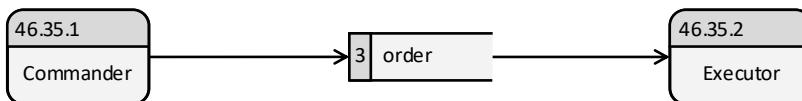


Because the commander and the executor are connected by a network, this is a remote invocation scenario.

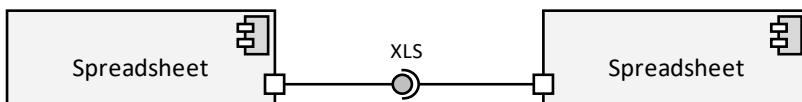
Delayed Invocation

Delayed invocation involves two components communicating through a saved file. Here the commander's orders are encoded in the command language and saved in a file. At some later point, the executor opens the file, interprets the orders, and executes them.

*Figure 46.35:
DFD of delayed invocation*



An engine management engineer has a spreadsheet containing tens of thousands of datapoints from the latest F1 test session. To facilitate analyzing the data, the engineer added a few functions to the spreadsheet. These functions are executed when the spreadsheet is opened. Here **Commander** is the engine management engineer and the software used to create the spreadsheet. The order is encoded using the spreadsheet's scripting language. The executor is the application used to open the spreadsheet and execute the scripts. Note that the commander and executor could potentially be the same system.



Because the orders are saved in a file, this is a delayed invocation scenario.

Recommendations

The interpreter design pattern facilitates creating rich and complex command languages

Best Practice 46.12 Carefully design the context to facilitate make the expression classes easier to implement

The commander needs to work with the context to create an order matching the system needs. The executor needs to work with the context to understand the actions contained therein. If you choose an overly general context (such as a string), then it may be very difficult to parse. Instead, choose a context format that leads to a simple or trivial `interpret()` design. In other words, your choice of a context can make the interface easy or difficult to implement. Don't make things harder for yourself than they need be!

Best Practice 46.13 Use a data interchange format to encode the document format

While it is possible to use virtually any file format to encode commands, not all formats parse equally easily. There are many data interchange formats that can make this job much easier. Formats such as XML and JSON are supported by numerous libraries in most languages.

When these formats are not powerful enough, parsing tools such as YACC and FLEX facilitate the creation of just about any format imaginable.

Best Practice 46.14 Mitigate against command and script injection attacks

A command injection attack is a security threat involving an attacker gaining access to an interpreter where such access is against policy. A script injection attack involves an attacker utilizing a provided interpreter in a way that is against policy. In both cases, the author of the system provided access to functionality that exceeds what the stakeholders intended. A malicious attacker can take advantage of such vulnerabilities to disclose, alter, or deny access to system resources. When designing a system to use the interpreter design pattern, be aware of command and script injection attacks. Make sure that malicious users cannot abuse the power that is contained in this interface.

Examples

Example 46.1: Direct

This example will demonstrate direct command passing.

Problem

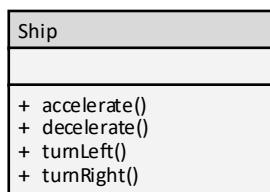
Consider the following scenario:

A video game involves moving a ship through a maze. The ship can speed up, slow down, turn right, and turn left. The ship will need to slow down, turn around, and speed up.

Create a design for this scenario using the direct command invocation strategy.

Solution

Direct invocation involves a dedicated function for each command. In this case, there are four commands: speed up, slow down, turn right, and turn left. Each of these commands is part of the `Ship` class.



The client needs to turn the ship around. This involves executing a sequence of commands: slow, turn, and accelerate.

Pseudocode

```
ship.decelerate()
ship.decelerate()
ship.decelerate()
ship.turnLeft()
ship.turnLeft()
ship.accelerate()
ship.accelerate()
ship.accelerate()
```

Example 46.2: Delegate

This example will demonstrate the delegate design pattern.

Problem

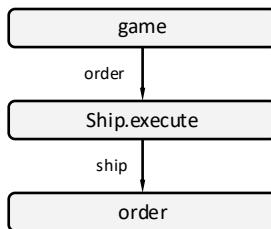
Consider the following scenario:

A video game involves moving a ship through a maze. The ship can speed up, slow down, turn right, and turn left. The ship will need to slow down, turn around, and speed up.

Create a design for this scenario using delegate invocation.

Solution

In this scenario, the executor will be changed by the order. This requires the order to accept executor context.



There are four functions which need to be implemented, one for each action. Each function must have the same function signature and take a **Ship** as a parameter.

C++

```
void accelerate(Ship & ship);
void decelerate(Ship & ship);
void turnLeft( Ship & ship);
void turnRight( Ship & ship);
```

The final step is to send the orders to the ship. This is done through **Ship's execute()** method.

C++

```
{
    ship.execute(decelerate);
    ship.execute(decelerate);
    ship.execute(decelerate);
    ship.execute(turnLeft);
    ship.execute(turnLeft);
    ship.execute(accelerate);
    ship.execute(accelerate);
    ship.execute(accelerate);
}
```

Example 46.3: Encapsulated

This example will demonstrate the encapsulated invocation design pattern.

Problem

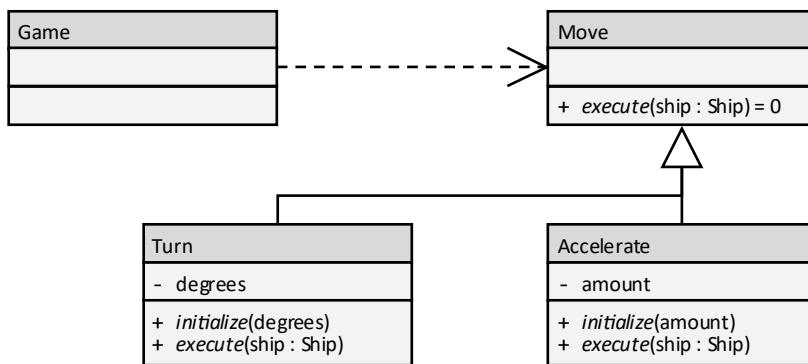
Consider the following scenario:

A video game involves moving a ship through a maze. The ship can speed up, slow down, turn right, and turn left. The ship will need to slow down, turn around, and speed up.

Create a design for this scenario using the encapsulated invocation design pattern.

Solution

Two concrete commands are created (accelerate and turn). Each concrete command has an `initialize()` method which configures the command.



With a richer command language, it is possible to move the ship with less actions than were required by the delegate or direct invocation.

Pseudocode

```
ship.move(new Accelerate(-3))
ship.move(new Turn(180))
ship.move(new Accelerate(3))
```

Observe how only one Turn object is required to turn the ship by 180° when previously it required two `turnLeft()` function executions.

Example 46.4: Interpreter

This example will demonstrate the interpreter design pattern.

Problem

Consider the following scenario:

A video game involves moving a ship through a maze. The ship can speed up, slow down, turn right, and turn left. The ship will need to slow down, turn around, and speed up.

Create a design for this scenario using the interpreter design pattern.

Solution

The data interchange language used to represent the command language will be JSON. This language has terminal expressions such as accelerate, decelerate, turn left, and turn right. It also has nonterminal expressions such as loops and conditionals. The body of the loop and the conditional can be any number of expressions.

```
JSON
{
  "commands": [
    {
      "type": "while",
      "expression": "speed > 0",
      "commands": [
        {
          "type": "decelerate",
          "amount": 1
        }
      ]
    },
    {
      "type": "turnLeft",
      "amount": 180
    },
    {
      "type": "accelerate",
      "amount": 3
    }
  ]
}
```

Exercises

Exercise 46.1: Command Passing Technique

Identify the command passing technique based on the description.

Description	Technique
Orders are represented as a class	
Orders are represented as a rich command language	
A unique interface is provided for every order	
Orders are represented as functions	

Exercise 46.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
Queuing orders is easy with direct invocation	
All delegates must have the same function signature	
Most delegates require commander or executor context to operate	
Encapsulated invocation is also known as the command design pattern	
All concrete commands must derive from the same abstract command base class	
RPC and RMI are forms of interpreter invocation	

Exercise 46.3: Components

The four command passing design patterns consist of several component, each of which is represented as a class or as a method within a class. For each component, identify the corresponding design pattern and briefly describe the role it fulfills.

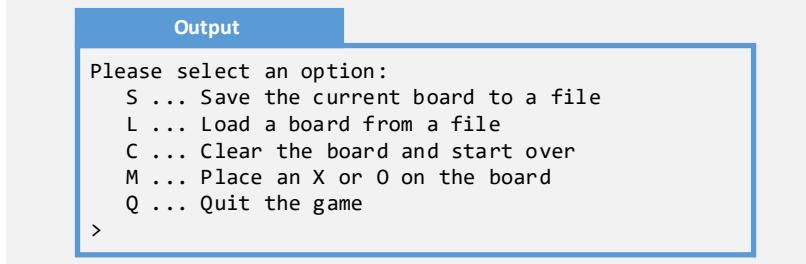
Component	Design Pattern	Role
Context		
Invoker		
Concrete Command		
Callback		
Nonterminal Expression		
Receiver		
Abstract Command		
Terminal Expression		
Abstract Expression		

Problems

Problem 46.1: Direct Tic-Tac-Toe

Consider the following scenario:

The game of tic-tac-toe has a 3x3 board where each square has an X, O, or space in it. A program will read a game from a file, save a game to a file, clear the board, and allow a player to make a move. Each of these actions is accessible from a simple text-driven user interface. The menu for the user interface is the following:



Create a design for the user interface of Tic-Tac-Toe using the direct command invocation strategy.

Problem 46.2: Delegate Tic-Tac-Toe

Consider the Tic-Tac-Toe scenario from Problem 46.1. Create a design for the user interface of Tic-Tac-Toe using the delegate command invocation strategy.

Problem 46.3: Delegate Tic-Tac-Toe

Consider the Tic-Tac-Toe scenario from Problem 46.1. Create a design for the user interface of Tic-Tac-Toe using the encapsulated command invocation strategy.

Problem 46.4: Delegate Tic-Tac-Toe

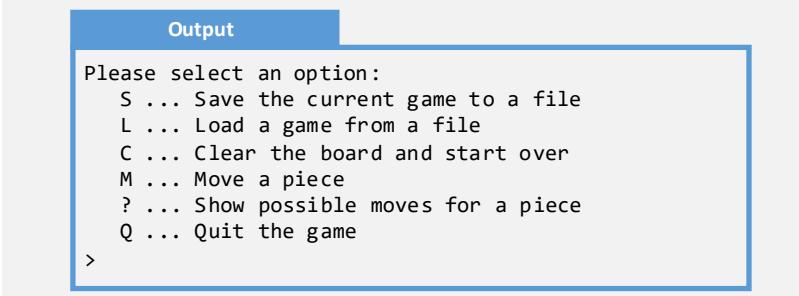
Consider the Tic-Tac-Toe scenario from Problem 46.1. Create a design for the user interface of Tic-Tac-Toe using the encapsulated interpreter invocation strategy. Specify the command language, provide examples of orders, and describe how to build the interface using a convenient design representation tool.

Problem 46.5: Direct Chess

Consider the following scenario:

The game of chess has an 8x8 board where each square has a black piece, white piece, or no piece on it. There are several pieces, including the king, queen, rook, bishop, knight, and pawn. Each piece is governed by a set of rules and the game will not allow a piece to move illegally.

A program will read a game from a file, save a game to a file, clear the board, show possible moves, and allow a player to make a move. Each of these actions is accessible from a simple text-driven user interface. The menu for the user interface is the following:



Create a design for the user interface of chess using the direct command invocation strategy.

Problem 46.6: Delegate Chess

Consider the chess game from Problem 46.5. Create a design for the user interface of chess using the delegate command invocation strategy. Include in your design the ability to queue commands.

Problem 46.7: Encapsulated Chess

Consider the chess game from Problem 46.5. Create a design for the user interface of chess using the encapsulated invocation strategy. Include in your design the ability to record a game and to allow for multi-level undo.

Problem 46.8: Interpreter Chess

Consider the chess game from Problem 46.5. Create a design for the user interface of chess using the interpreter strategy. The Smith Notation is a language used to describe a piece movement.

Problem 46.9: Direct To-Do

Consider the following scenario:

A mobile application that maintains a to-do list. The list can be manipulated in a variety of ways: adding items, removing items, re-ordering items, and changing the status of an item (where the status could be ready, in progress, or completed).

Create a design for this application using the direct command invocation strategy.

Problem 46.10: Delegate To-Do

Consider the to-do application from Problem 46.9. Create a design of the list manipulation component of the application using the delegate command invocation strategy. Include in your design the ability to redo and queue commands.

Problem 46.11: Encapsulated To-Do

Consider the to-do application from Problem 46.9. Create a design of the list manipulation component of the application using the encapsulated command invocation strategy. Include in your design the ability to record actions and to allow for multi-level undo.

Problem 46.12: Interpreter To-Do

Consider the to-do application from Problem 46.9. Define a command language that would capture everything a user would want to do with a to-do list.

Challenges

Challenge 46.1: Tic-Tac-Toe

Consider the following scenario from Problem 46.1:

The game of tic-tac-toe has a 3x3 board where each square has an X, O, or space in it. A program will read a game from a file, save a game to a file, clear the board, and allow a player to make a move. Each of these actions is accessible from a simple text-driven user interface. The menu for the user interface is the following:

```
Output
Please select an option:
S ... Save the current board to a file
L ... Load a board from a file
C ... Clear the board and start over
M ... Place an X or O on the board
Q ... Quit the game
>
```

Implement this program in the programming language of your choice. Demonstrate all four command passing strategies described in this chapter. Compare and contrast each strategy. Which is most appropriate for this application?

Challenge 46.2: Chess

Consider the following scenario from Problem 46.5:

The game of chess has an 8x8 board where each square has a black piece, white piece, or no piece on it. A program will read a game from a file, save a game to a file, clear the board, show possible moves, and allow a player to make a move. Each of these actions is accessible from a simple text-driven user interface. The menu for the user interface is the following:

```
Output
Please select an option:
S ... Save the current game to a file
L ... Load a game from a file
C ... Clear the board and start over
M ... Move a piece
? ... Show possible moves for a piece
Q ... Quit the game
>
```

Implement this program in the programming language of your choice. Demonstrate all four command passing strategies described in this chapter. Compare and contrast each strategy. Which is most appropriate for this application?

Interfaces

Interfaces are mechanisms used to simplify the coupling between system components. This is accomplished by hiding implementation details behind a simple, client-focused interface.

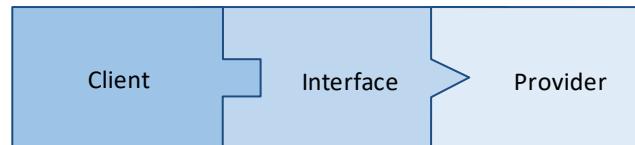
Developing small software applications can be described as a process of fabrication, where every piece is written from scratch. Large systems, however, are built differently. A big part of large system design consists of assembling components, each of which was independently developed and tested. These components can often be easily integrated into the system. It is frequently the case, however, that even the best-designed components do not fit seamlessly into the overall system design. What is the best way to handle situations like these?

*Figure 47.1:
Two incompatible
components*



For example, consider a bicycle shop that would like to sell their products on the web. Currently, this shop uses point of sale (POS) software to keep track of what they have in inventory. This system has been working for years and has proven very reliable. Rather than build a website from scratch, the bike shop decided to use a website builder service which has most of the functionality they need. The problem is that these two pieces do not speak the same language. This leaves the bike shop with two bad options: to modify the POS software so it can interface with the website, or to modify the website software so it can interface with the POS system. Each of these two options seems very difficult and expensive. Fortunately, a third option exists. Instead of modifying either of these components, a third component can be introduced, serving as an mediator between the other two.

*Figure 47.2:
An interface linking
two components*



There are four strategies we can use to bridge the gaps between clients and providers. We call these *interface design patterns*.

Design Pattern	Description
Adapter	Wrap an existing class with a new interface.
Bridge	Separate the interface from the implementation.
Façade	A single interface for a complex subsystem.
Proxy	A stand-in for a complex object.

Adapter

An adapter, also known as a wrapper, is a class exhibiting the interface that the client expects while also containing a class that the client needs.

Figure 47.3:
An adapter which
contains an instance of
the provider

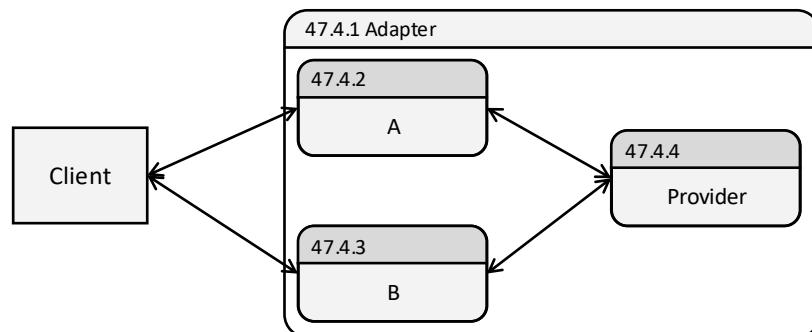


There are three parts to the definition of an adapter.

1. **Client.** The client is the consumer of an interface, requiring data or services from another part of the program. The client can be another class, it can be a component, or it can be the application in general. For the client code to function as it should, it communicates with the provider using a convenient interface. The client in our bike shop scenario is the new website.
2. **Provider.** The provider is a component or class that delivers services to the client. While the provider might consist of code that is written especially for the application under development, it is often the case that the provider is delivered as a package that cannot be easily changed. The provider in our bike shop scenario is the POS system.
3. **Adapter.** The adapter is an intermediary standing between the client and the provider. The client interfaces only with the adapter, which then passes on requests to the provider. The adapter in our bike shop scenario is the code written to facilitate the website communicating with the POS system.

The adapter class contains an instance of the provider using composition. Since the provider is contained as a private member variable, the client has no way of knowing that the provider exists as a separate class; it is only aware of the adapter.

Figure 47.4:
DFD of the adapter
design pattern



In the Figure 47.4, observe how the adapter provides all the convenient interfaces that the client needs (represented as A() and B()). Each of these public interfaces translates the client's request to corresponding interfaces on the provider's side. Often this translation is easy, amounting to reordering parameters or performing simple data translation. Other times, things can be far more complex, involving assembling data through several function calls.

Though the adapter pattern can be implemented at a variety of different levels of program design, it is most often implemented at the class level.

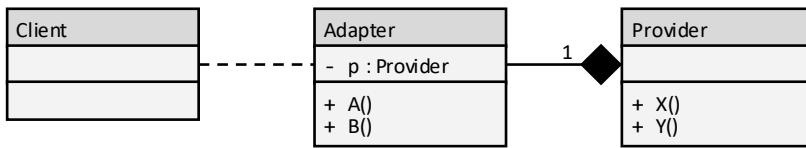


Figure 47.5:
Class diagram of the
adapter design pattern

In Figure 47.5, notice that the adapter contains an instance of the provider through composition. Because the adapter contains an instance of the provider (represented as `p`), it is easy for the adapter's methods to access the provider's interfaces (represented as `X()` and `Y()`). The client can then access the adapter's interfaces (represented as `A()` and `B()`) through a variety of ways: through composition, association, or even accessing the adapter as a static class. Therefore, the interface between the `Client` and the `Adapter` in the class diagram is presented as dependency (a dotted line).

Back to our bicycle shop example, an adapter is developed which contains an instance of the POS interface. The website, serving as the client, will direct all its requests to the adapter. This way, the POS component does not need to be modified, and the new website can be created using the website builder tools without modification.

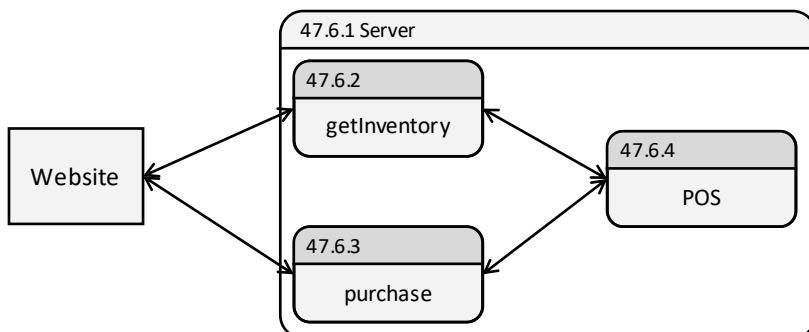


Figure 47.6:
DFD of the adapter in the
bicycle website

Notice that all requests to the `POS` component travel through the adapter (called `Server`); the client has no direct access to the `POS`. The class diagram for the bike shop website closely follows the adapter design pattern. `Server` provides the methods the website needs to display – the inventory and handle purchases. `POS` provides the methods to manipulate inventory. Note that the number of methods and the names of the methods differ between `Server` and `POS`. This is not a problem – the `Server` class translates between these interfaces.

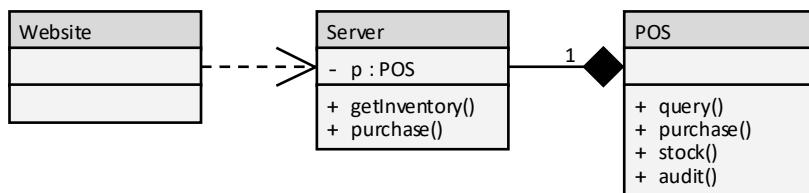


Figure 47.7:
Class diagram of
the adapter in the
bicycle website

The website code exists in a different component than `Server`. This means it cannot use composition, aggregation, or even association to connect to `Server`. In this case, the protocol is HTTPS. Therefore, the dependency arrow is used in the class diagram.

These interfaces are public methods in the adapter class. One of these methods is `purchase()` which facilitates moving a shopping cart item into the sold list.

Pseudocode

```
purchase(response, cookie)
    clientID ← getClientID(cookie)
    serverID ← SERVER_ID

    transaction ← getTransaction(response, clientID, serverID)

    transaction.callback ← https://www.bikeshop.com
    transaction.currencyCode ← USD
    transaction.version ← 1.5
    transaction.tenderType ← CREDIT_CARD, GIFT_CARD, BITCOIN
    transaction.date ← getCurrentDate()
    transaction.time ← getCurrentTime()

    IF error in transaction
        ERROR

    pos.purchase(transaction)
```

Figure 47.8:
Pseudocode of one
method in the adapter
for the bicycle website

Observe how the `purchase()` method is highly coupled with the POS system. Not only does it require a great deal of information that the customer is unlikely to possess, but the details are likely to change with updates to the POS system. Furthermore, if the bike shop decides to use a different POS vendor, then this code will have to completely change. While this is less than ideal, we can take comfort in one thing: only the adapter needs to worry about this. No other website code needs to be aware of these details and all the POS-specific code is concentrated in a single location.

Best Practice 47.1 Be wary of performance problems in adapter methods

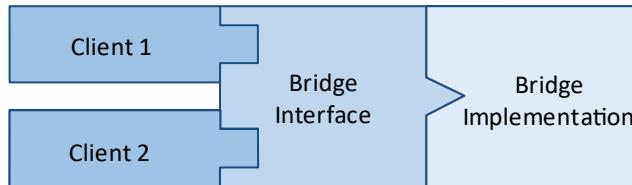
When the client uses adapter interfaces, there is an expectation that things will run as smoothly as if the provider interface was directly queried. Therefore, be extra cautious of performance problems in adapter interfaces. One poorly written adapter interface doing more work than necessary can bring down the performance of the entire system.

Bridge

A bridge is a mechanism to separate the interface (which the client sees) from the implementation (how the provider chooses to implement the code). Using this pattern, the provider designs the interface to best honor the needs of the client, with the understanding that the client's needs may change. The provider also designs the implementation to most effectively and efficiently honor the contract specified in the interface. The code from the interface then calls the corresponding code in the implementation.

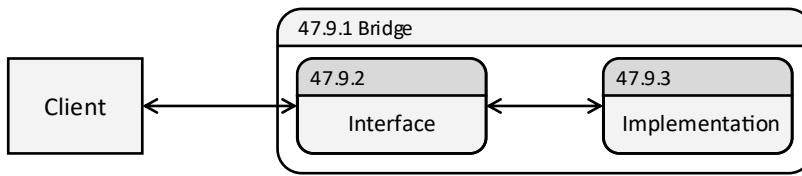
The bridge is like the adapter except the provider interface is designed in parallel with the interface

Figure 47.9:
The bridge



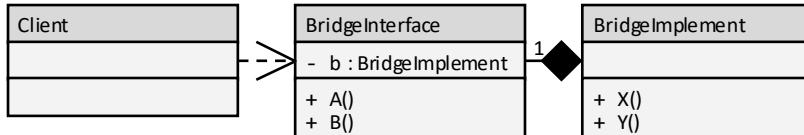
In many ways, this is like an adapter. There is one important distinction: whether the provider can be changed. With the adapter, the provider interface is fixed. With the bridge, the provider interface is designed specifically to both provide the necessary services and to be easy to integrate with the interface.

Figure 47.10:
DFD of the
bridge design pattern



In figure 47.10, observe that the bridge consists of two parts: the bridge interface and the bridge implementation. The bridge interface has one design constraint: to make the interface as convenient as possible for the client. The bridge implementation has two design constraints: to effectively implement the provider services as well as to simplify the translation to the bridge interface.

Figure 47.11:
Class diagram of the
bridge design pattern



In simple cases, the class diagram looks a great deal like that of the adapter, though with different names. The bridge implementation is contained within the bridge interface, which the client accesses. Recall that the bridge implementation is not a fixed entity (as is the case with the adapter pattern); the bridge implementation is created in parallel with the bridge interface. Thus, the bridge implementation is specifically designed to provide the necessary services and to make the bridge interface as elegant and efficient as possible.

Bridge Interface

The first part of the bridge pattern is the bridge interface. In simple cases, this consists of a single set of public-facing methods with which the client may access the provider services. There are times when this may be more complex. Consider the scenario when the client may require a variety of interfaces to support different platforms. In cases like these, the bridge interface can be an inheritance hierarchy.

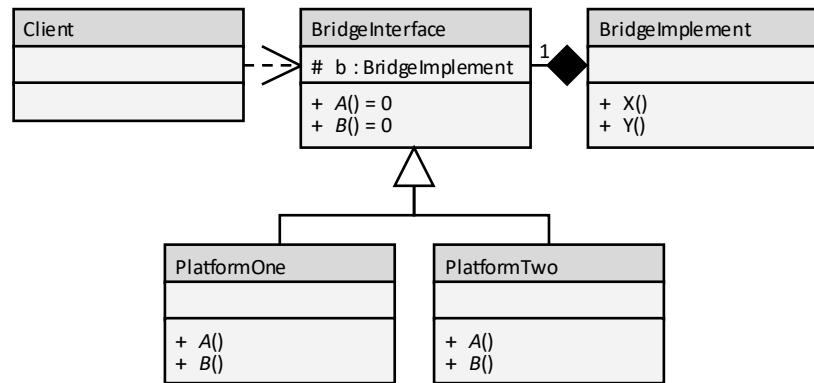


Figure 47.12:
Class diagram of a bridge
with two interface
derivatives

Notice that there is still one path from the bridge interface to the bridge implementation. That path is through the `BridgeInterface` base class, which handles all interface–implementation communication. However, the public-facing interfaces of the two derived classes are completely tailored to the needs of the various clients.

Bridge Implementation

The second part of the bridge pattern is the bridge implementation. This is the component that provides the services that the client requires. As with the bridge interface, this can be simple, with a single service, or it can be complicated, with a collection of services.

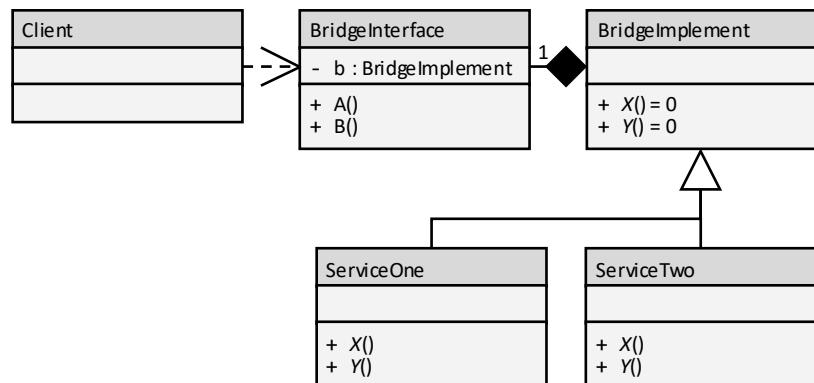
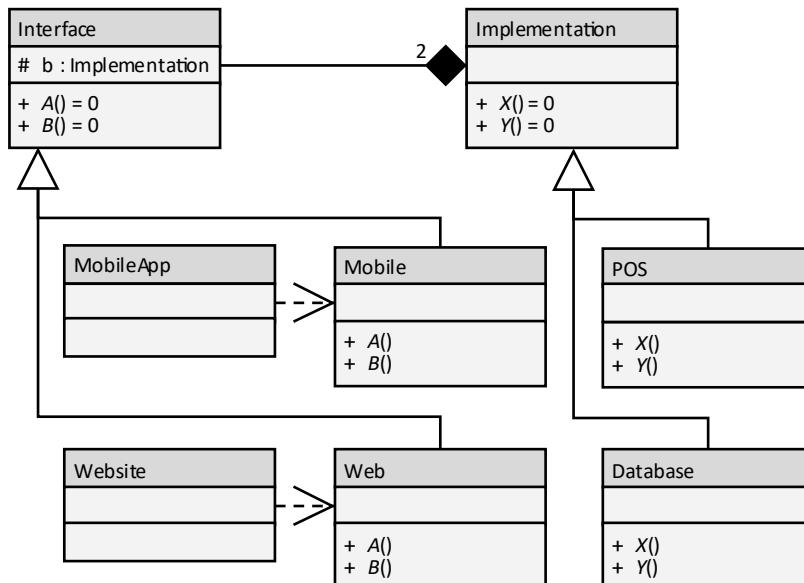


Figure 47.13:
Class diagram of a bridge
with two implementation
derivatives

In figure 47.13, there are two services: `ServiceOne` and `ServiceTwo`. Both services fully implement the public methods from the base class. Using this pattern, it is possible to have a wide variety of provider implementations, all of which are accessed by the client using the same convenient interface.

Back to our bike shop example, the owner is so pleased with the website that she wants to have a mobile application as well. She also wants to sell items directly from her warehouse that is never part of her store inventory. This means she wants to access the warehouse database directly without going through the inventory management system. Using the bridge pattern, all these changes can easily be implemented.



*Figure 47.14:
Class diagram of the bike
website and mobile
application*

In this example, there are two clients (the mobile application and the website), each of which uses both providers (the POS for store inventory and the database for warehouse inventory). What if there were n clients and m providers? Using the bridge pattern, we would have to create n bridge interfaces and m bridge implementations. Without the bridge pattern, we would need to create $n \times m$ adapters, each of which contains a large amount of redundant code.

Best Practice 47.2 Use a bridge pattern when there are multiple clients or multiple providers

The bridge pattern eliminates any duplicate code when there are several clients requiring different interfaces or when there are many providers with different interfaces. This design has the distinct level of redundancy.

Best Practice 47.3 Use the bridge pattern when you can control the implementation interface

Many situations involve integrating components with predetermined interfaces. The adapter is often the best solution in those cases. However, when the provider interface is negotiable, then the bridge pattern often yields more malleable and efficient solutions.

Façade

A façade is a simple, client-facing interface hiding the implementation details of a complex subsystem. The provider may choose to have a complex inheritance hierarchy with many subtle and varied algorithms, but these should not be exposed to the client. The name *façade* is an architecture term. A building's façade is the exterior's public-facing side. If a building has an obvious entrance or opens to a large public space, then that side of the building is the façade. The essential meaning of the façade design pattern is the same: the client-facing interface of a component or subsystem.

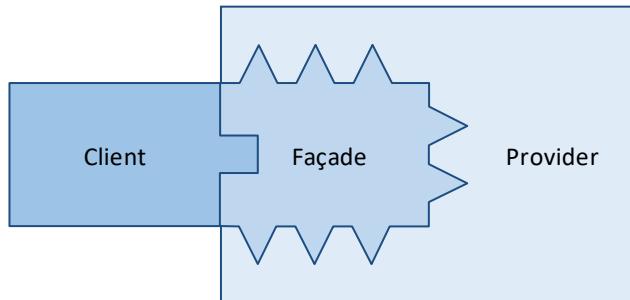


Figure 47.15:
Façade

Perhaps this is best explained by example. When a customer visits an e-commerce website (such as our bike shop's website), the client sees a very simple set of options. There are items to select and a shopping cart to facilitate purchases. Behind this façade, there are many complex things going on. The items may be stored in a variety of locations, the process of computing the price may be complicated with taxes and discounts, and the system that presents the customer with recommendations may be quite sophisticated. All these complexities are shielded from the customer with a simple façade.

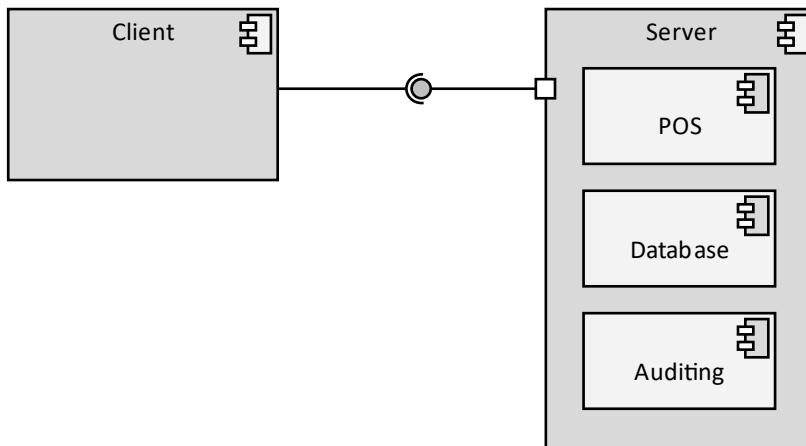


Figure 47.16:
Component Diagram of a
complex server design
interfaced through a
façade

In figure 47.16, notice the small square connecting the provided interface with the rest of the system. This square is the façade. The component behind the façade might be large and complex, and it may consist of many subcomponents which may be complex as well. The client is never exposed to this complexity because all the client's interactions with the provider are through the façade. There are many variations of the façade, the most common being single functions, ports, collections of functions, and classes.

Single Function

A façade can be a single function. Usually when this happens, the input parameter to the function is sufficiently rich to give the client access to all the provider's services. Probably the most commonly used example of this is the `system()` command available in many programming languages.

Figure 47.17:
Perl code interfacing with
the operating system
through the `system` façade

```
Perl
my @pingCommand = ("ping", "-a", "157.201.130.149");
system(@pingCommand)
```

Operating system services perform a wide variety of complex tasks, including controlling the various hardware devices, storing data, and providing access to a wide variety of software tools. These operating system services are made available to developers without the programming language having to enumerate them all. The `system()` façade provides full access to all these services with a single function. Notice that this function uses document coupling.

Port

A port façade is like a single function façade in that input is accepted through a single channel and that message comes through as text. There is one important difference. Single function façades are synchronous; one function invokes another. Ports are asynchronous, where a message arrives from an external entity. This can happen through a remote procedure call (RPC), a web request (HTTPS), an XML message (SOAP), or a variety of other protocols.

Figure 47.18:
Component diagram of a
JSON port façade

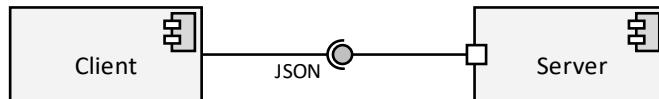


Figure 47.18 depicts a client component interfacing with a server component through a single JSON protocol. The server accepts the input through a port façade.

Figure 47.19:
Python façade code

```
Python
import json
import requests

# Accept input from an external source.
input_text = requests.get("https://www.someserver.com/data")

# Convert the data to JSON.
input_json = json.loads(input_text)

# Do something interesting...
```

In figure 47.19, we can see the bike shop server connecting to the warehouse database to make an inventory request through the `Inventory` port. The client will then unpack and interpret the resulting JSON data. Though the mechanism used to pass messages between functions is very different than that used to pass messages between subsystems using ports, the façade implementation is quite similar. In both cases, messages are passed as text and then converted by an interpreter using document coupling.

Collections of Functions

A façade can be a collection of functions, each of which should provide direct access to a given service. One example of this is OpenGL, an application programming interface (API). OpenGL provides the client with a vast suite of functions, each designed to facilitate drawing 2D and 3D graphics. Many programmers find OpenGL too complex for their applications. To address this need, another API was developed: OpenGL Utility Toolkit (GLUT). This family of functions provides a simple interface for the complex and powerful OpenGL. Note that both OpenGL and GLUT are façades. Not only does OpenGL provide a façade for programmers, but GLUT provides a façade for OpenGL.

Figure 47.20:
Component diagram of
two façades

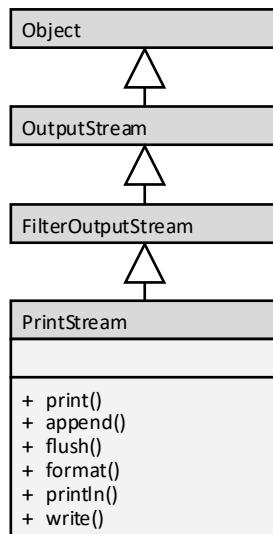


Observe how OpenGL has a façade, depicted as the small square on the left of the component. The entire GLUT component serves as a façade for the client. Each of these are coupled using collections of functions.

Classes

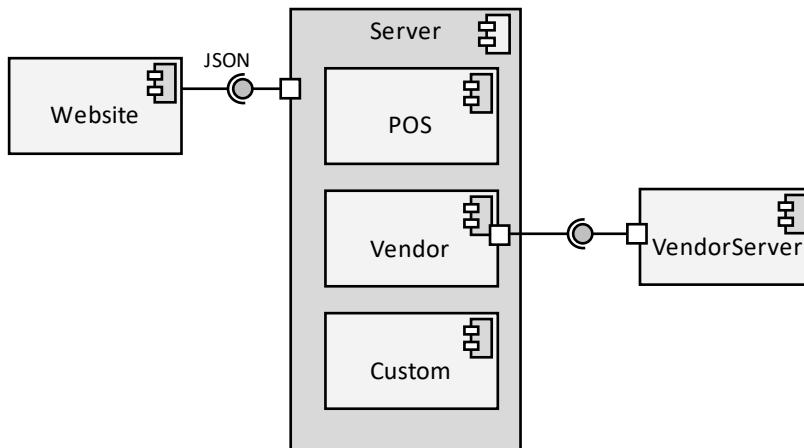
A final common manifestation of the façade design pattern is that of a single class. Here, the class serves as the interface for a complex subsystem, the details of which the client never needs to understand. The Java `System.out` object is just such an example.

Figure 47.21:
Class diagram of Java's
PrintStream façade



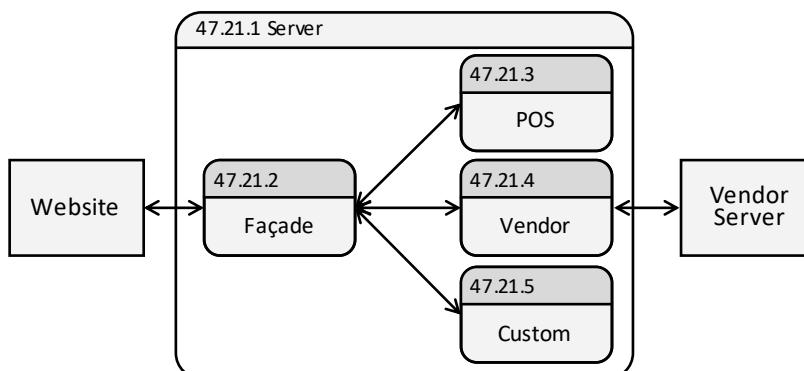
The `System.out` object is a member of the `PrintStream` class. This class allows the client to easily send text to the screen. Notice that the complexity of displaying text on a variety of terminal devices is shielded from the client. Most Java programmers are blissfully unaware of how difficult it is to put a character on the right screen in the right location. Fortunately, languages such as Java provide a convenient façade for us.

Back to our bike shop example, the new web store has been a big hit. Wanting to expand, the company wants to sell products from other vendors and special, made-to-order bikes. Each of these expansions involves a different backend and different ordering mechanism. However, the website front end needs to be the same. It should not matter if the bike is physically sitting in the inventory, if it exists at a partner's warehouse, or if it is custom made by a frame builder next door. To handle this, a façade is created. This façade provides a convenient interface for the website (list inventory, add items to cart, purchase, etc.) hiding the complexity that goes on behind the scenes.



*Figure 47.22:
Component diagram of
the bicycle website façade*

We will implement this façade with a JSON interface that communicates through HTTPS. The process of querying about inventory and making a purchase request comes through a JSON message. From here, the façade will determine how to handle the request and send it off to the various subsystems. Note that the **Vendor** component will need a façade to interface with the **VendorServer** component, and the **VendorServer** component will need a façade as well. This provides maximum simplicity; if anything needs to be changed, this change can be implemented without affecting the **Website**, **Server**, or **VendorServer**.



*Figure 47.23:
DFD of the bicycle
website façade*

The façade class will contain instances of some of the provider classes while maintaining a remote connection to others (the vendor server in this case).

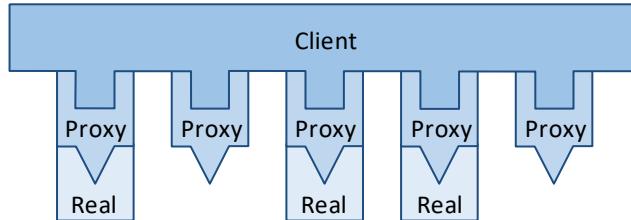
Best Practice 47.4 Use the façade pattern when the underlying provider is large or complex

Though façades can be vast and difficult to create, they can shield the client from having to understand complex subsystems. In times like these, a façade can greatly increase the convenience of an otherwise prohibitive API.

Proxy

A proxy is an object that stands in for the real or genuine object. From the client's perspective, the proxy *is* the real object. The proxy honors all the interfaces that the real object provides. However, when the client needs to access the real object, then the proxy knows how to invoke it. This way, the real object is not instantiated unless the client needs it.

Figure 47.24:
Proxy

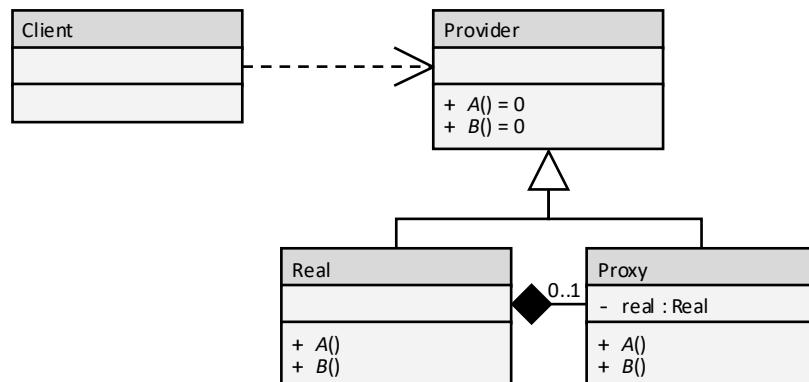


There are three parts to the definition of a proxy.

1. **Client.** The client is the consumer of an interface. It may need to invoke the real object or provider.
2. **Real.** The real object is the provider that the client may need to invoke. There could be one real object for the client, or the client could maintain a collection of these objects as depicted in figure 47.24.
3. **Proxy.** The proxy stands between the client and the real object. It honors all the interfaces of the real object but is little more than a stub. Note that if the client requires more than a trivial interaction with the real object, then the proxy instantiates it.

There are many ways to implement a proxy, but the most common is to create a **Provider** base class from which **Real** and **Proxy** derive.

Figure 47.25:
Class diagram of the
proxy design pattern



There are a few things to notice about the proxy design pattern. First, the **Proxy** class and the **Real** class honor the same interfaces. This means **Proxy** can stand in for **Real**. Second, **Proxy** contains either zero or one instances of **Real**. In the case where the client does not require **Real**, then it is not instantiated. However, when **Real** is needed, **Proxy** instantiates it. When **Proxy** contains an instantiated instance of **Real**, the client's requests are just passed through to it.

The **Proxy** class is a very thin veneer covering the **Real** interface. Consider, for example, a method that requires a real object.

```
Pseudocode
Proxy.essential()
IF NOT_real
    real = new Real
RETURN real.essential()
```

Figure 47.26:
Pseudocode of an
essential proxy interface

The proxy checks to see if it already has instantiated the **Real** interface and creates it if necessary. From the client's perspective, there is no difference between interfacing with **Proxy.essential()** and **Real.essential()**.

In a second scenario, imagine a method that does not require a real object. Here, some placeholder or default data will be enough.

```
Pseudocode
Proxy.notEssential()
IF real
    RETURN real.notEssential()
ELSE
    RETURN default
```

Figure 47.27:
Pseudocode of an
interface that is
not essential

There are several scenarios when one would want to use a proxy: when the real object is remote, expensive to create, or needs to be protected.

Remote Proxy

A remote proxy is a proxy that stands in for an object that exists in a remote location. This real object may be periodically unavailable or slow to respond because it exists on the other side of a network connection. A remote proxy is often called an ambassador or a cache.

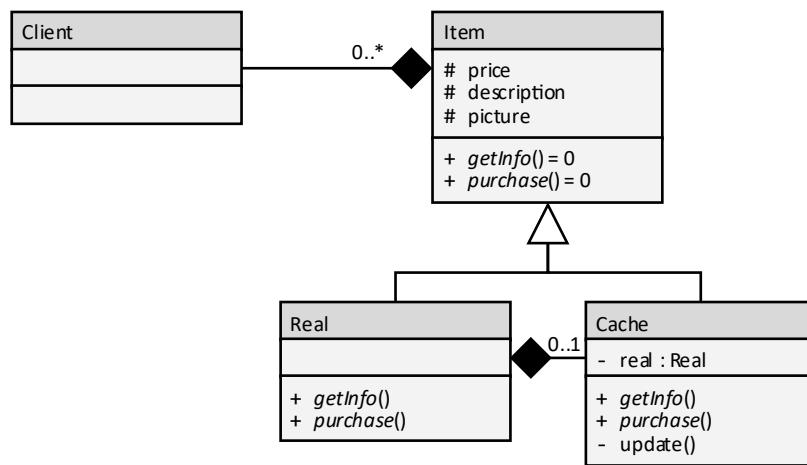


Figure 47.28:
Class diagram of
a remote proxy

Back to our bike shop example, the server maintains a list of all the bicycles that are available to sell. Some of these bicycles are in a partner's inventory, accessed in a database on a different computer. When a new item is added to the partner's inventory, a proxy is created in the bike shop's server. This proxy is a copy of the partner's inventory item. We call this copy a cache. Only when directly needed is the cache updated (through the proxy's **update()** method).

Virtual Proxy

A second type of proxy is a virtual proxy, a stand-in for a real object that is expensive to create. A virtual proxy will delay performing the expensive operation until the client makes a request that requires it. Only in this situation is the connection between the real object and the proxy established.

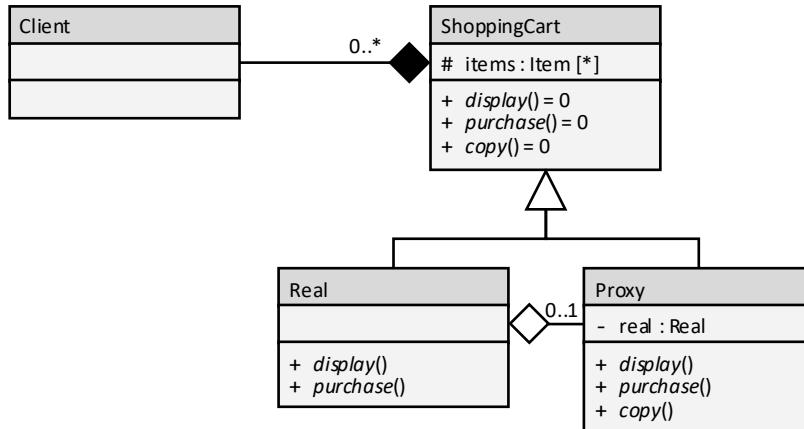


Figure 47.29:
Class diagram of
a virtual proxy

Our bike shop website has a shopping cart. The owner wishes to implement a feature where the contents of the shopping cart can be saved in a wish list. Any number of wish lists can be created, each of which is a duplicate of the shopping cart. To avoid making unnecessary copies of the shopping cart, a virtual proxy is used. Only when a modification is made to the wish list or the shopping cart is the link severed and the actual copy made.

Protection Proxy

A protection proxy is a mechanism to keep unauthorized clients from accessing the real object. The client has access to all the methods to manipulate the real object, but those methods performing unauthorized operations do not access the real object.

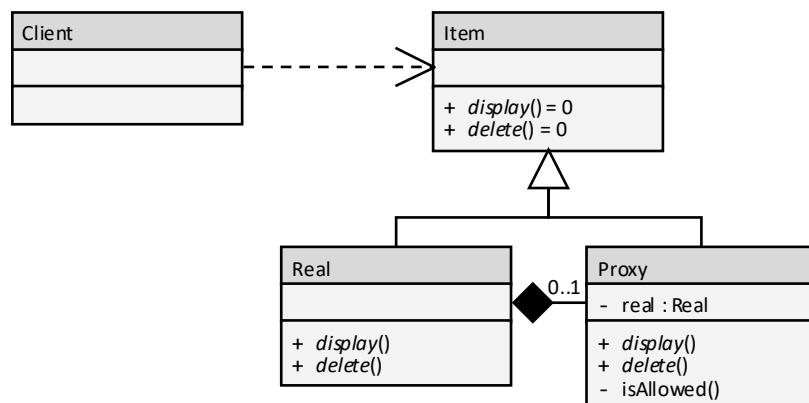


Figure 47.30:
Class diagram of
a protection proxy

In our bike shop website example, both the owner and the clerk can access the `display()` method of an item in the inventory. Thus, the proxy just passes the `display()` request on to the real item. However, only the owner can delete an item from the inventory. The `delete()` method will then call `isAllowed()`. If the action is allowed, then it is passed on to `real.delete()`. Otherwise, the action is ignored.

Examples

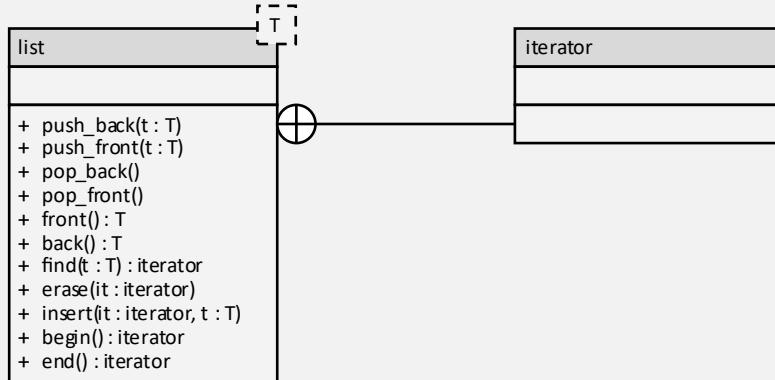
Example 47.1: Adapter

This example will demonstrate the adapter design pattern.

Problem

Create a design for the following scenario:

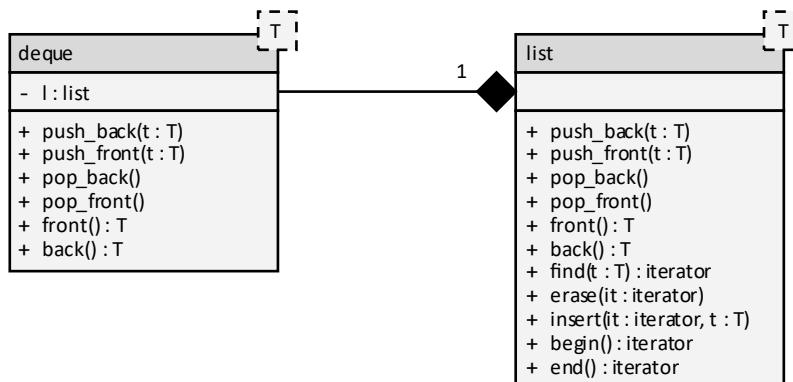
Consider the list abstract data type. With a list, one can add or remove elements from either end, or iterate through the list.



Research the deque abstract data type. Pay special attention to the public interfaces. Based on this, implement a deque with a list.

Solution

The first thing to notice is that the list has a superset of functionality of the deque. This means that every deque method has a corresponding list method, but not the other way around. We can therefore use the adapter pattern to create the deque.



Each deque method will simply call the corresponding list method.

Example 47.2: Bridge

This example will demonstrate the bridge design pattern.

Problem

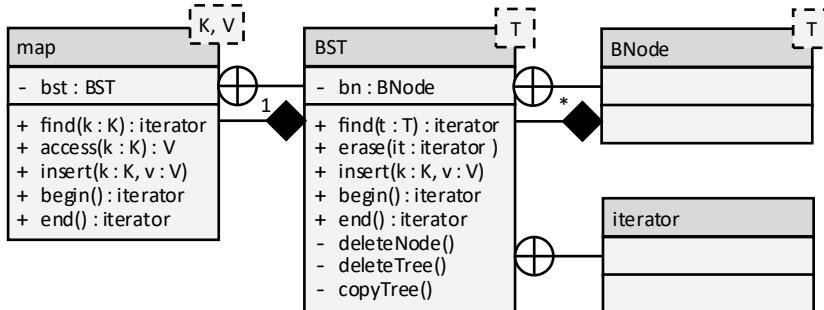
Create a design for the following scenario:

Consider the class diagram for a binary node called **BNode**. With this **BNode** class, it is possible to create a binary search tree (BST) data structure. The map abstract data type uses the BST as the underlying data structure. Create a map from the **BNode** class.

Solution

This is a good candidate for the bridge design pattern because both the interface (which is the **map** class, and the implementation (which is the **BST** class) needs to be defined.

The first step is to research and define the public interfaces for the **map**. The most important public methods are **access()**, **find()**, and **insert()**. These will be done with **find()** and **insert()** from the **BST** class. The goal is to create a **BST** class in such a way that it is easy to honor the interface of **map** without having to make the binary search tree aware of map details. The class diagram for the entire bridge is the following:



The **BST.find()** method will return a pointer to the node. The code for access is the following:

Pseudocode

```
map.operator[key]
pair ← key, Value()
it ← bst.find(pair)
IF it ≠ NULL
    RETURN it.second
ELSE
    bst.insert(pair)
    RETURN bst.find(pair).second
```

Observe how this design separates the interface the client expects (being the **map** class) from the implementation (the **BST** class).

Example 47.3: Façade

This example will demonstrate the façade design pattern.

Problem

Create a design for the following scenario:

In 1980, Atari released the video game *Asteroids*. This game allowed the user to fly a ship around the screen and shoot asteroids into oblivion. We wish to implement this game using the GLUT façade. Note that the game wants to be able to draw a rock or a ship, while GLUT allows the programmer to draw lines and dots. Create a façade to shield the game developer from having to understand GLUT.

Solution

The first step is to identify the types of things that *Asteroids* will need to draw on the screen. This includes drawing a bullet, the ship, the thrust behind the ship, the three types of rocks, and the score. Additionally, the game will need to be notified when the space and arrow keys have been pressed. Each of these is best accomplished with a single function.

The best choice here is the collection of functions façade. The following functions are needed:

Function

```
drawShip(position, orientation, isThrusting)
drawBullet(position)
drawBigAsteroid (position, rotation)
drawMediumAsteroid(position, rotation)
drawSmallAsteroid (position, rotation)
drawScore(score)
```

The small asteroid drawing function is the following:

C++

```
void drawSmallAsteroid( const Point & center, int rotation)
{
    // ultra simple point
    struct PT { int x; int y; } points[] =
    {
        {-5, 9}, {4, 8}, {8, 4},
        {8, -5}, {-2, -8}, {-2, -3},
        {-8, -4}, {-8, 4}, {-5, 10}
    };

    glBegin(GL_LINE_STRIP);
    for (int i = 0; i < sizeof(points)/sizeof(PT); i++)
    {
        Point pt(center.getX() + points[i].x,
                  center.getY() + points[i].y);
        rotate(pt, center, rotation);
        glVertex2f(pt.getX(), pt.getY());
    }
    glEnd();
}
```

Example 47.4: Proxy

This example will demonstrate the proxy design pattern.

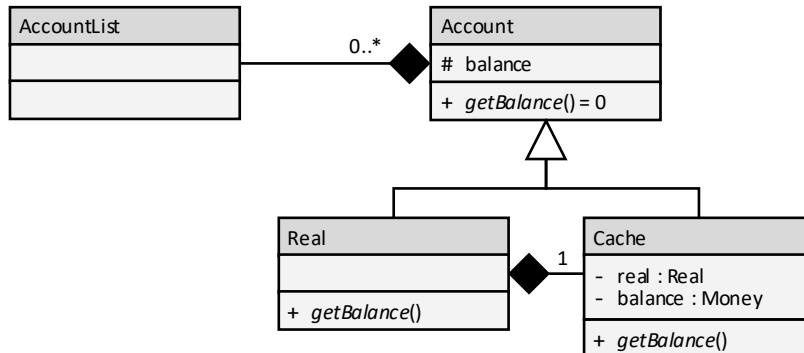
Problem

Create a design for the following scenario:

A mobile application displays a user's account balances for his or her various financial accounts. The application needs to work in a partially connected environment, meaning that it should display the most recent balance when disconnected from the network. However, whenever a network connection exists, it should present the current balance from the bank.

Solution

This scenario looks to be a good candidate for a remote proxy. The proxy will contain a cached account balance of the last value received from the bank. When the user attempts to access the balance, a new value is requested.



The `getBalance()` method on the proxy will first attempt to call the real value. If it fails, then the cached value is used. Otherwise, the cached value is updated, and the updated value is returned to the client.

Pseudocode

```
Cache.getBalance()
TRY
    balance ← real.getBalance()

RETURN balance
```

Notice that `Real.getBalance()` will throw an exception if the network connection cannot be made for any reason. When this happens, the cached balance is just returned. When the exception is not thrown, the cached exception is updated with the new value and that is returned. In either case, the most recent value is returned to the client.

Exercises

Exercise 47.1: Types of Interface Strategies

From memory, name and define the four types of interface strategies.

Exercise 47.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
It is possible for the bridge pattern to present multiple interfaces to the client.	
The adapter pattern expects the programmer to design the provider interface.	
Single function facades tend to use document coupling.	
The proxy pattern is a simple interface representing a complex subsystem.	
A façade design pattern can only be implemented using classes.	

Exercise 47.3: Viewpoints

For each of the following, use an appropriate viewpoint (class diagram, component diagram, structure chart, or DFD) to represent the design pattern.

Design Pattern	Representation
Adapter	
Bridge	
Single function Façade	
Port Façade	
Class Façade	
Remote Proxy	
Virtual Proxy	
Protection Proxy	

Problems

Problem 47.1: Stack

Consider the deque and the stack abstract data types. You may need to do some research to discover the essential interfaces for each. Create an adapter for the deque so it behaves like a stack. In the language of your choice, write the code to implement this design.

Problem 47.2: Big Money

One problem with representing numbers as integers is that there is a practical limit to the largest possible value. A 32-bit integer has only 4 billion possible values ($2^{32} \approx 4$ billion) and a 64-bit integer has about 20 sextillion possible values. We would like to store money values larger than that. Create a design for a money class that contain an arbitrary number of digits. Do this by representing large values as a list of integers, each element representing a digit. The public interface should do all the things people want to do with money (save, display, copy, add-to, subtract-from, etc.). The private implementation should be sufficiently generic that we can use our big number code to represent a variety of non-monetary things.

Problem 47.3: Graphics Window Instantiation

Research your favorite graphic engine (DirectX, OpenGL, etc.) and find the code necessary to create a single window that is 100x100 pixels. Create a single function façade for this process so the client can do this more easily. Add some parameters to the façade so it can create the window in a way that is useful for most of the simple video games that you would want to build.

Problem 47.2: Text Document

We have invented a new text document format that consists of a collection of links, each link being a file name for a plain text file. When the user opens this new text document format, all the plain text files are loaded and presented to the user as a single, continuous document. A problem occurs when one of the plain text files is not available. We would like to handle this case by putting a placeholder in the document presentation so, if the missing file is found, it can be inserted seamlessly. Create a design for this application and write some code in the language of your choice which demonstrates its functionality.

Challenges

Challenge 47.1: Bank Checks

Research the various ways that financial institutions represent bank checks using JSON. If possible, use a financial institution in which you personally have an account. Additionally, imagine a mobile application allow the user to view and make modifications to a check transaction. With this JSON format and mobile application, do the following:

1. Define a bridge implementation for the JSON data using a class diagram.
2. Define a bridge interface that would be convenient for the mobile application using a class diagram.
3. Introduce a remote proxy so cached data can be presented to the user when a remote connection to the financial institution is not possible. This remote proxy is also represented using a class diagram.
4. Create a DFD representing the entire design.
5. Identify two key methods in the design. Create a pseudocode draft of the algorithms.

Challenge 47.2: Game Interface

Research your favorite graphic engine (DirectX, OpenGL, etc.). Consider your favorite game (*Asteroids*, etc.). Create a façade for the graphic engine so it is convenient for your game. Present the design in a design description consisting of about a dozen viewpoints.

Challenge 47.3: Shared To-Do List

Imagine a mobile application that stores a collection of to-do lists. Each list is stored on a server and is associated to a single account. Many accounts are shared, meaning that more than one family member, or a work team, can share a single account. Each list in an account has a set of permissions associated with it, enumerating the individuals who have access to the list. All the lists are stored on the server, though any application can request an account provided it has the appropriate credentials.

Create a design for this application as a design description consisting of about a dozen viewpoints.

Tokens and Entities

Entities are items from the problem domain that a computer system needs to manage. Tokens are tools used to represent entities.

For years, George operated a tiny grocery store in a small mountain town. The town has recently experienced a large amount of growth in anticipation of the opening of a new ski resort. To accommodate this growth, George has expanded his store into a large new building. Instead of carrying a hundred items, the new store will carry tens of thousands. George is skeptical that his old pen-and-paper inventory system will work with the new store. Reluctantly, George is going to have to use a computer inventory management system (IMS).

As George gathers the requirements for this IMS, he realizes that there are three distinct components: one to add items to the inventory, one to manage the collection, and one to interface with the collection when purchases are made. A big challenge for George's new system is how to represent an item in the inventory. This is particularly important because George cannot recompile the system every time a new item is added. How can George represent an item? To solve this problem, George is going to have to come up with an entity representation strategy.

An entity is a component of the problem domain

An entity is a component of the problem domain. In the case of George's grocery store, an entity could be a bottle of milk, a package of cookies, or an apple. The problem domain is the store's inventory. This is the summation of all the entities that the system will need to manage. Of course, one cannot put an apple or a package of cookies into a computer system. To do that, George is going to need a token.

A token is a representation of an entity in a computer system. Fundamentally, computers can only store 1s and 0s. The goal is to make the 1s and 0s stand in for an entity. There are a very large number of different ways to represent a given entity in a computer program, and this decision can have a large impact on the quality of the overall system design. With George's grocery store, a token will need to be chosen to adequately represent a distinct item in the inventory.

Entity representation is the process of representing components of the problem domain in a computational construct

Entity representation is the process of representing components of the problem domain in a computational construct. In other words, it is the process of selecting appropriate tokens that both adequately represent entities and facilitate the workings of the token management system.



Figure 48.1:
UPC Barcode

Components

Each entity representation problem has four components: the token, the originator, the caretaker, and the client. The originator generates a token from an entity. This token is given to the caretaker which will store or process it. The client interacts with the caretaker, occasionally requesting tokens. The client can then translate the token back to the entity to fulfill the needs of the system.

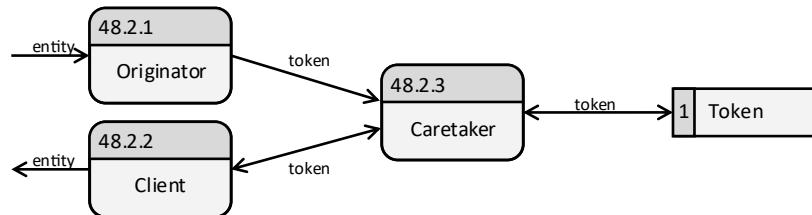


Figure 48.2:
DFD of the token/entity
relationship

Token

Tokens, often called mementos, objects, or variables, are computer constructs representing entities. They represent the data being moved through the system.

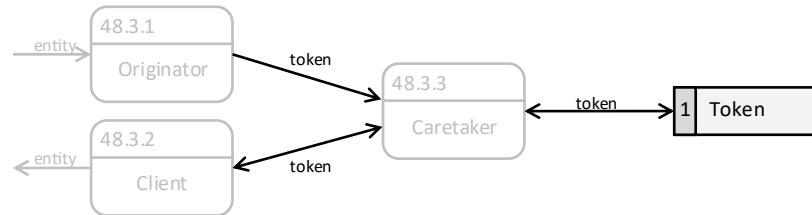


Figure 48.3:
DFD of the token part of
the token/entity
relationship

Tokens are usually of a fixed size, meaning they are the same size regardless of the entity they are meant to represent.

Best Practice 48.1 Each token should be the same size

The size of a token is the amount of memory required to store it on a computer system. A byte token, for example, would be eight bits in size. When each token on the system is the same size, it is easier to manage entity collections.

Best Practice 48.2 Token should be easy to move, duplicate, and delete

The caretaker should be able to easily manipulate the tokens it stores. This means that common operations such as moving, duplicating, and deleting should be fast and easy to accomplish. When a token is a built-in data type such as an integer, this is easy to do. When something more complicated is used, then work should be done to ensure that it is efficient and easy to move, duplicate, and delete the token.

Back to our grocery store example, George has many token options for the entities in his IMS. As grocery stores were becoming increasingly computerized in the 1970s, the Universal Product Code (UPC) was developed. This system generates a unique ID for every product (entity) sold in most stores, making UPCs a natural token choice for George's IMS. Note that the UPC-A format is a 12-digit number with a trillion combinations. This can be encoded in a 64-bit integer. Thus, each UPC-A token is the same size (satisfying Best Practice 48.1) and is easy to manipulate (satisfying Best Practice 48.2).



Figure 48.4:
UPC Barcode

Originator

The originator, sometimes called the owner, is the component which creates tokens from entities to be added to the system. It is called the originator because all tokens are generated by this component. When the problem domain is large or when the problem domain changes with time, the originator component can be complicated. Occasionally, a new token will need to be generated from a previously unseen entity. This process is called enrollment.

The originator generates a token from an entity

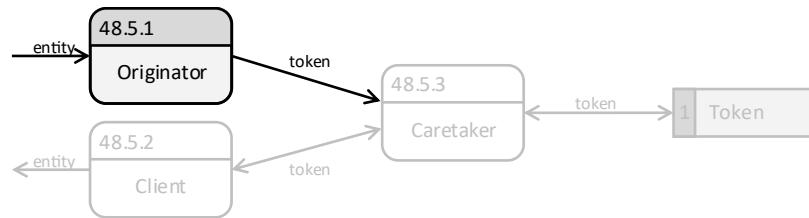


Figure 48.5:
DFD of the originator
part of the token/entity
relationship

Best Practice 48.3 Each token should map to a single entity

The originator should ensure that tokens are not reused or repurposed. If more than one entity corresponds to a single token, then it can be difficult or impossible to determine what the token stands for. While it may be unavoidable in some situations, care should be taken to minimize the instances when a single token maps to more than one entity.

Best Practice 48.4 Each entity should map to a single token

It is desirable but not required that the entire problem domain be represented in the set of possible tokens. This may not be possible when there are an infinite number of entities. However, when each entity has exactly one corresponding token, then it becomes possible for the system to determine if two entities are the same through comparison of their tokens.

Back to the grocery store example, all the products that are stocked in George's store have a UPC printed on them. Every product wishing to have a UPC must register with the GS1 corporation. This not-for-profit organization was created in 1974 for the purpose of maintaining a single worldwide product database. The GS1 corporation is the ultimate originator, with the stocking component interfacing with it.

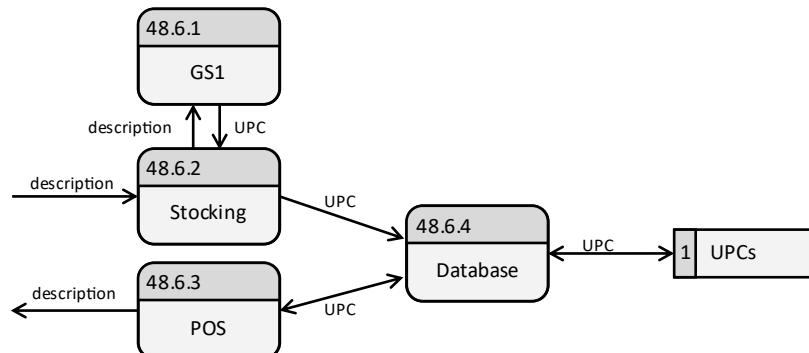


Figure 48.6:
DFD of the
grocery store IMS

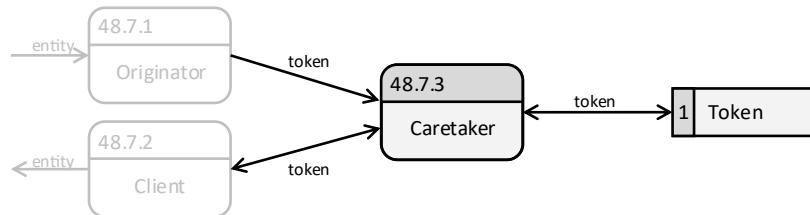
Since George only stocks elements with a UPC, then he can be sure there is a 1:1 mapping between the tokens (UPC) and the products (entities).

Caretaker

The caretaker is the part of the system which manages the collection of tokens. The caretaker accepts new tokens from the originator, can send tokens to the client, and can manipulate the collection of tokens as needed. The most common way for the caretaker to store a collection of tokens is in a data structure, database, or a file. The caretaker can move, copy, or delete a token, but knows nothing about what the token represents. From the caretaker's perspective, the token is a black box whose contents cannot be viewed.

The caretaker manages tokens

Figure 48.7:
DFD emphasizing the
caretaker



Best Practice 48.5 The caretaker must be blind to the contents of the token

The caretaker must not depend on the implementation details of any aspect of the token except for its size. In other words, if the originator chose to change the implementation of the token, the caretaker should not require any design alterations to accommodate this change. This simplifies the coupling between the caretaker component and the client or originator.

The caretaker never knows what the tokens represent

It is most convenient to think of the caretaker as a database or some other data storage mechanism. While this is certainly common, it captures only a small part of what a caretaker can do. A few examples include the following:

- Storage/Retrieval: the caretaker is essentially a data storage mechanism
- Logging: storing the state of a system that periodically changes
- Undo/Redo: maintaining a stack of actions which can be rolled back or run forward
- Queue: place items in a line so they can be handled later
- Event Notification: inform subscribers that an event has transpired

The types of actions a caretaker can perform is endless. The common thread between these examples is that the caretaker should be concerned only with managing the tokens, not with what the tokens represent.

In George's IMS, the caretaker is the database which stores all the elements in the grocery store inventory. The caretaker can accept new product tokens from the originator as new elements are added to the stock. The caretaker can allow tokens to leave the system from the cashier as purchases are made. The caretaker can even run reports on the inventory, describing the number of elements contained therein. However, the caretaker cannot know if the inventory consists of carrots, cars, or kittens. From the caretaker's perspective, they are all the same.

Client

The client is the component that interacts with the caretaker to implement the business logic of the system. To do this, the client must have the ability to interpret a token. This is accomplished one of two ways: either the client asks the originator to interpret the token, or client and the originator are a single component.

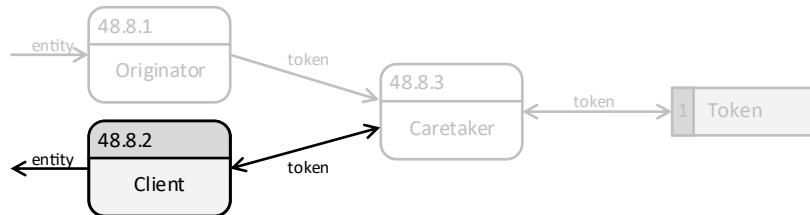


Figure 48.8:
DFD emphasizing the
client

Best Practice 48.6 Only one component should handle translation between a token and an entity

The goal here is to minimize or eliminate redundancy in the system. If any change is made in the entity representation strategy of the system, then only one component should have to change. There are three ways to handle this. The first is to make the originator and the client the same component.



Figure 48.9:
DFD with a unified
originator and client

The second way is to have the client utilize the originator to decode the token. This concentrates all knowledge of the relationship between the entity and the token in a single location. It also gives the client a single responsibility: to contain the business logic related to manipulating the entities.

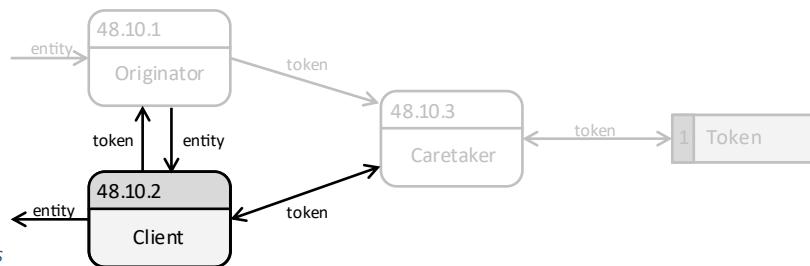


Figure 48.10:
DFD with the client
interfacing with the
originator to resolve tokens

There is one final option. It is possible to encode enough information in the token that it can auto-decode. In this case, no originator is needed once the token has been created. This is called a thick token, which will be discussed later in the chapter.

In George's IMS, a mechanism will have to be built to convert the UPC into a human-readable description so a receipt can be printed. While it may be tempting to keep a cached copy of all the UPCs utilized by the grocery store, it would be better to use the GS1 corporation as the oracle. This way, only the most recent name for a given product will be used.

Entity Representation Scenarios

Entity representation is a surprisingly common requirement in a diverse set of systems. At first glance, it may seem like there is no relation between these scenarios. However, the four components of all entity representation schemes are present: the token, originator, caretaker, and client.

Collection Management

Collection management, as the name suggests, is the process of a system managing a potentially large and diverse collection of entities. This is the scenario of George's grocery store IMS. Just about every factory, warehouse, and store has an IMS which utilizes a token-based collection management strategy. The most common operations performed in such a system are to maintain a list of all the entities in the collection, produce a variety of reports, add, remove, and enroll new entities into the system, and retrieve entities to the facilitate business functionality.

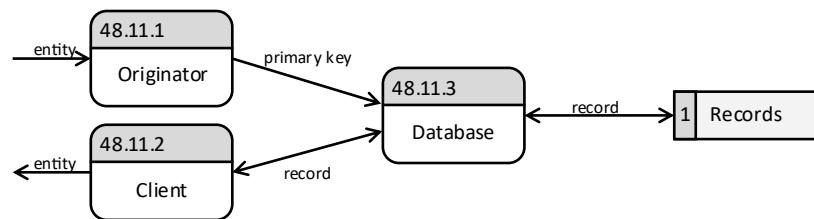


Figure 48.11:
DFD of a collection
management system

In all collection management scenarios, the entity is the stock to be managed.

- **Token:** Though a wide variety of tokens can be used, the most common is to use a unique ID for each product type. If a database is the data store, then the token is called the primary key.
- **Originator:** As products are created or added to the inventory, the originator component adds new tokens to the system. Note that the process of adding new products usually involves special enrollment functionality on the part of the originator.
- **Caretaker:** The caretaker is called the data store in collection management scenarios. In most cases, the data store is a database. Records are stored and retrieved from a database with a primary key. It is common for a database record to store other metadata with the primary key, such as the date the token entered the system, the status of the item, or any other relevant information. The primary key and all associated metadata is called a record.
- **Client:** In most cases, the originator and the client are the same component. These are terminals at each location where a product can enter or leave the business. As products are sold or are removed from the inventory, the client removes the tokens from the system.

Identity Management

Authentication is the process of proving individuals are who they say they are. This involves an individual presenting credentials to the system and, if the system accepts the credentials, the individual is granted the status of a user. During the authentication process, an individual can have several roles. The individual can be a subscriber who is permitted to have access to some or all the system resources. The individual can be an applicant who is not currently a subscriber but aspires to be one. The individual could be an outsider who is not a rightful user of the system. Finally, the individual can be a claimant who claims to be a subscriber but has not yet been verified.

The authentication process determines whether a claimant is a subscriber or an outsider. If the claimant proves to be a subscriber by providing valid credentials, then they are granted a digital identity called an agent. An agent could be a login, an account, or an online persona. Most identity management systems have two components: the authentication component and the enrollment component. The enrollment component accepts an applicant's identity proof and generates a new agent if the proof is sufficient.

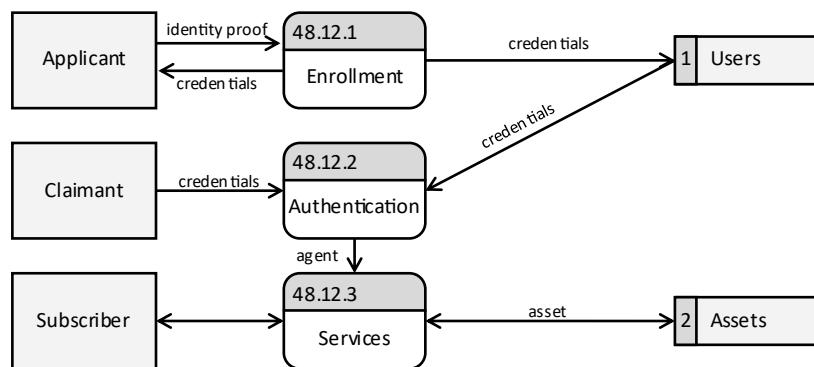


Figure 48.12:
DFD of an identity
management system

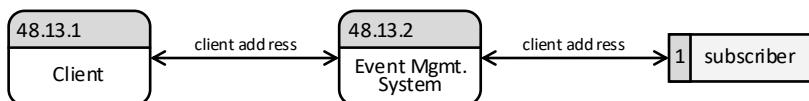
Identity management is an entity representation scenario. The entity in this case is the individual. This can be a subscriber, applicant, outsider, or claimant.

- **Token:** The token is the agent, usually in the form of an ID or username.
- **Originator:** The originator is the enrollment process, turning a claimant's credentials into a token which is added to the system. In many cases, the credentials consist of an agent token (such as a username) and an identity proof (such as a password).
- **Caretaker:** The caretaker is the system's authentication mechanism, which determines if a claimant is a subscriber. If the credentials match that which is in the user database, then an agent is produced. Otherwise, the claimant is labeled an outsider.
- **Client:** The client is the operating system service that the subscriber is attempting to access. The service usually provides system functionality or access to system assets. To do so, the service compares the subscriber's agent with the permissions required of the functionality or the asset. This process is called access control and can be built into the service or a separate component of the operating system.

Event Notification

Event notification is the process of one component subscribing to an event from some event management system in another component. There are many situations when this may occur. For example, the client of a graphics engine may subscribe to drawing and input events. The client of a network interface may wish to be notified when a message arrives. The client of a file management system may wish to know when a file resource has changed. In many ways, this is like the observer message passing design pattern. The only difference is that event notification can occur between process spaces, making the observer's OO design impossible. With the observer design pattern, all possible clients are derived from a single base class. This means that all elements need to be known at compile time. This constraint does not exist with the entity representation scenario.

Figure 48.13:
DFD of an event
notification system



Event notification is an entity representation scenario. The entity is the client application wishing to receive notifications or events.

- **Token:** In some cases, the token is the address of the client requesting notification. This address could be a network address of the client in a different memory address space, or a pointer if the client is in the same address space.
- **Originator:** The originator is the client wishing to be notified. The originator needs to give the event management system enough information that the resulting event can be handled.
- **Caretaker:** The caretaker is the event management system itself. This component generates events but does not know how to handle them. A client is needed for that. When an event has been generated, then the client is notified through the provided client address. The event management system may keep one subscriber or a collection of subscribers, depending on its purpose and design.
- **Client:** The client is the system that responds to and handles events. Since the client is also the party making the initial request for events, the originator and the client are the same component in this scenario.

Queue Management

Queue management, as the name suggests, is the process of a system handling an unpredictable stream of asynchronous events in a first-come, first-serve basis. For example, consider a customer walking up to the pharmacy part of George's store. If no one is currently being served, then the customer can meet with the pharmacist directly. However, when more than one customer is present, a line can be formed. To make this more convenient, each customer is asked to take a number and wait for the "now serving" sign to indicate that the pharmacist is ready. The number dispenser coupled with the now serving sign is the pharmacy's queue management system.

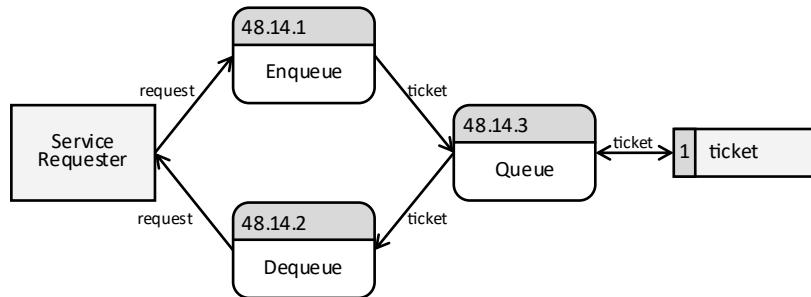


Figure 48.14:
DFD of a queue
management system

In the queue management scenario, the entity is the party wishing to partake of the service. This could be a customer waiting to be served, a document waiting to be printed, or an order waiting to be fulfilled.

- **Token:** Though many tokens can be utilized, the most convenient is to use a counter. The first item added to the queue when the system starts is given number one, and the second is given number two. In most cases, the token is called a ticket.
- **Originator:** The originator is the component accepting requests for service and issuing tickets. This process is known as enqueueing (pronounced "en-Q-ing"). The originator also knows which entity maps to which token. Note that in most queue management scenarios, enqueueing is an enrollment process.
- **Caretaker:** The caretaker is the queue itself, accepting new tokens at the end of the line and removing the oldest token from the front of the line. The caretaker is often called the queue (pronounced "Q").
- **Client:** The client is the component which signifies it is ready for the next entity to be served. This is accomplished by asking the queue for the element in the front of the line. Once this is received, the element is removed from the queue. This is called dequeuing (pronounced "de-Q-ing"). It is very common for the originator and the client to be the same component in queue management scenarios.

State Persistence

Computers were originally calculators, lacking the ability to store data between sessions. As persistent storage devices were developed, it became possible for a program to suspend a session and resume it at a later point in time. Today, users expect most applications to be persistent. They expect to be able to save a game and pick up at a later point in time. They expect for a book reading application to remember the page the reader was last reading. They expect a web browser to retain a history of the pages that were visited. They expect a word processor to remember the last several documents that were opened. Each of these is a form of state persistence.

State persistence is the process of a system remembering the configuration of a system to be recalled at a future point in time. Though a simple state persistence system may only remember one state, some systems require the system to remember many states, any of which can be recalled at the client's discretion.

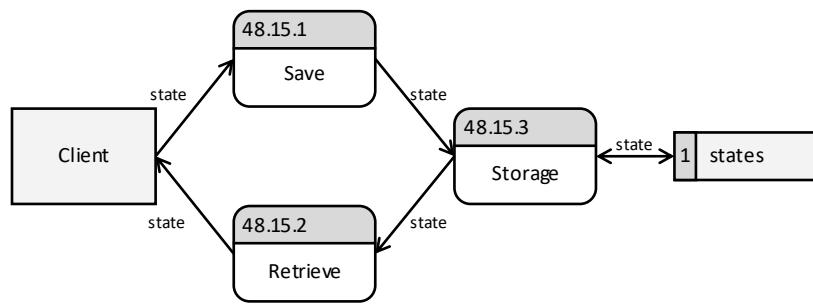


Figure 48.15:
DFD of a state
persistence system

State persistence is entity representation scenario. The entity is the system state or configuration. This can be as simple as a single value such as a page in a book, or it could include a large number of settings such as the state of a video game.

- **Token:** In state persistence scenarios, a token can go by many names. Some of the most common include checkpoint, bookmark, save file, state, and marker. Tokens often store enough information to recreate the state they are meant to represent. A simple configuration such as a location in a book can be represented with a scalar value containing all the information necessary to restore the state. More complex configurations require the originator to store the state values in a separate location and issue a unique ID or name corresponding to the configuration. A file name would be an example of a token fitting this description.
- **Originator:** The originator is the component storing all the configuration information for a state and issuing a token to represent the configuration. This component is often called save.
- **Caretaker:** The caretaker is the system managing the list of collection of states to be persisted. A most recently used (MRU) feature or a bookmark feature would be an example of a caretaker.
- **Client:** The client is the mechanism facilitating the selection of and restoration of a state. This is called the retrieval system.

Token Selection Strategies

In each entity representation scenario, a key design decision is the selection of an appropriate token. There are two broad categories of tokens: thin tokens and thick ones. A thin token is an integer or a short string requiring the originator to decode it to an entity. This requires the originator to maintain a table mapping the entity to the token. A thick token is a token selection strategy where enough information is present in the token to recreate the entity without needing to contact the originator. Thick tokens are distinguished from thin ones in that no token/entity tables are needed.

Incremental Counter

One of the simplest token selection strategies is to use an enumeration, counter, or index. This thin token strategy involves the originator maintaining a list of all the entities currently used by the caretaker. This is called an entity table. During the enrollment process, the originator checks to see if a given entity already exists in the entity table. If so, then the corresponding index in the entity table is used. Otherwise, the entity is appended onto the entity table and a new token is issued.

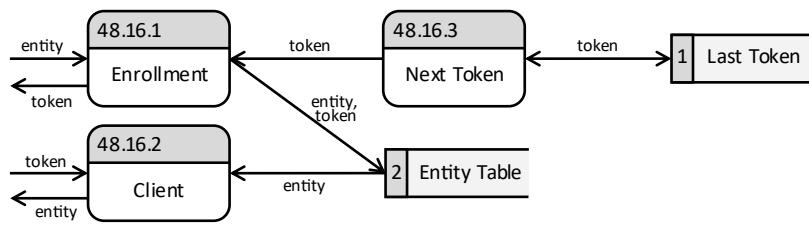


Figure 48.16:
DFD of an incremental counter originator

Incremental counter tokens are commonly used in the physical world around us. The number in a now serving system is an incremental counter token. A raffle ticket at a local fund raiser is also an incremental counter token. A check number in your checkbook is also an incremental counter token. Any time the enrollment process generates tokens in sequence, an incremental counter system is at work. In each of these cases, the originator maintains an entity table of all the entities in the system and tokens are simply indices into this entity table. When a new entity is added to the system, a last token counter is incremented by one and that value is the assigned token to the new entity. Thus, adding new entities or looking up an entity based on the index is O(1). This can be more complicated when the system has multiple enrollment components or when the enrollment component needs to restart, but these can easily be managed.

Best Practice 48.7 Do not use incremental counters if tokens need to keep entities confidential

There are some disadvantages to using incremental counter tokens. First, the value of a token may reveal details of the entity that the owner may wish to keep private. A malicious user eavesdropping on the caretaker can reconstruct the sequence of entities enrolled into the system simply by comparing token values. A malicious user can also determine the total number of entities in the system, a fact the owner may wish to keep confidential.

A more serious security vulnerability may exist if a malicious user alters a token value. The malicious user could retrieve adjacent entities in the entity table by incrementing or decrementing a token value. For example, if a credit card number was created using an incremental counter token, then one could charge a purchase to another person's account by adding one to a given credit card number.

Key

A key is a token selection strategy where the enrollment process generates a random name used to represent an entity. The originator maintains a collection of entities, each retrieved by a key. The most convenient mechanism to store these key-value pairs is a dictionary or a map, though databases are often used as well. These entity collections are, like those of incremental counters, called entity tables.

Key tokens are among the most common token selection strategies in our everyday life. The key can be an account number, an employee ID, a product ID, a model number, or even a username. In some situations, tokens are generated by the originator in the enrollment process. Examples include credit card numbers and social security numbers. In some situations, the user can select a token with the oversight of the originator. Selecting a username or an e-mail address is such an example. There are even situations where the user has the option of token selection or letting the system choose one. A license plate number is such an example.

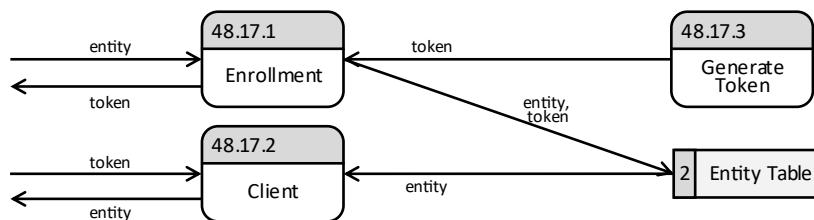


Figure 48.17:
DFD of a key originator

During the enrollment process, it is important that new entities receive a token not previously chosen. If the number of entities in the entity table are large, then this can be an expensive process. Since most entity tables are implemented as a dictionary or as a database, it is $O(\log n)$ to determine if a given token has already been issued. This can become a performance bottleneck when new entity enrollment is a common task.

Best Practice 48.8 Randomly generate token keys

In situations where token keys travel over networks, are persisted in files, or are publicly accessible in any way, token keys should be randomly generated. This makes it more difficult for an eavesdropper to deduce anything about the entity based solely from the token; the originator's entity table is required to make the translation.

Best Practice 48.9 Avoid packing token keys too closely together

Imagine a token represented as an 8-bit number. This makes for 256 possible tokens ($2^8 = 256$). If 128 tokens have been generated, that leaves 128 values available for future tokens. Thus, a randomly generated number has a 50% chance of mapping to a real entity. The more closely packed tokens, the greater the chance that a randomly generated token will map to a real entity. Imagine what would happen if credit card numbers were closely packed. A randomly generated 16-digit number would have a reasonable chance of mapping to someone's actual credit card account! To simplify the process of randomly generating unique token keys and to reduce the chance that a valid token can be guessed, there should be a million or more invalid key values for every valid token key value.

Object Reference

An object reference is a thick token strategy requiring the originator and the caretaker to share the same memory address space. Here, the originator encodes the entity in an object and the object, or a reference to the object, serves as the token. When the client needs to access the entity, it simply dereferences the object and all the needed information is present.

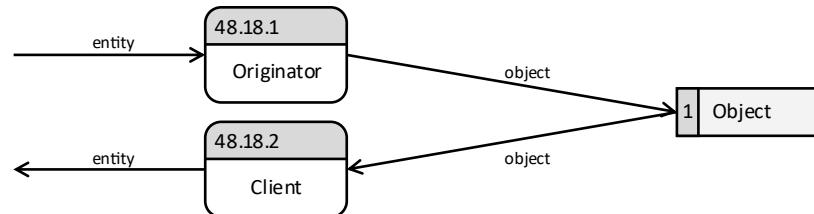


Figure 48.18:
DFD of an object reference
token strategy

There are two variations of object reference tokens. The first involves having a single base class representing all possible tokens and having a distinct derived class, each of which represents an entity. This strategy is useful when the number of entities is known at compile time and when the number of entities is small. However, the inclusion of new entities into the system requires new derived classes to be created. This makes the malleability of the system adjustable at best.

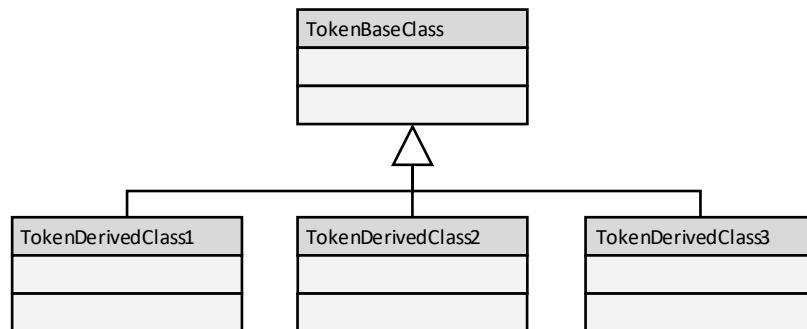


Figure 48.19:
Class diagram of tokens
related by inheritance

The more common variation of object reference tokens is to have sufficient attributes in the class to capture the richness of the various entities to be represented. While it can be difficult for the designer to identify all these attributes, the result is data-driven or configurable malleability.

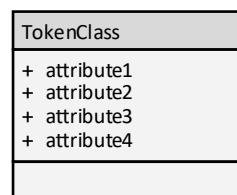


Figure 48.20:
Class diagram of tokens
represented by attributes
in a single class

Configuration File

The final token selection strategy is called a configuration file. Here, all the attributes necessary to represent an entity are captured in a single file. This file can be in the form of a network packet, a JSON entry, or even a traditional file system file.

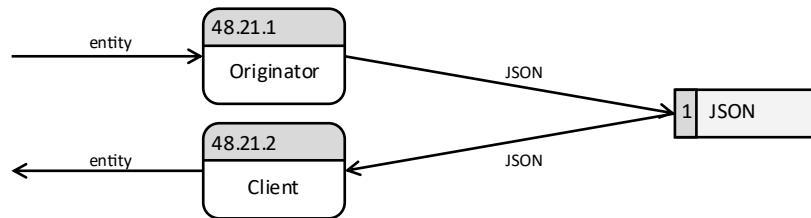


Figure 48.21:
DFD of a configuration
file token strategy

One common form of a configuration file is a cookie. Developed in 1994 by Netscape Communications, a cookie is a token sent to a web browser from a server so the server can maintain session state. Here the server is the originator and the browser is the caretaker. In some cases, the server generates a traditional key token. In other cases, the server loads the cookie with a host of configuration information. This information is sent back to the server with every web request. Though the details of the cookie's context vary according to the client/server relation, it is common to include an identify management token, user preferences, and even shopping cart contents. With a thick cookie, the server can reduce the size of the mapping table or even eliminate the mapping table entirely. The following example could be a cookie for a chess game, allowing the server to remember the state of the game.

```
JSON
{
  "name": "Byrne vs. Fischer, opening",
  "Date": 1956,
  "moves": [ "Nf3", "Nf6",
             "c4", "g6",
             "Nc3", "Bg7" ]
}
```

Figure 48.22:
A JSON token

Best Practice 48.10 Use a data interchange format to encode the entity

While it is possible to use virtually any file format to encode an entity, not all formats parse as easily. There are many data interchange formats that can make this job much easier. Formats such as XML and JSON are supported by numerous libraries in most languages.

Best Practice 48.11 Use a configuration file when passing entities between systems

When an entity needs to pass between components, a configuration file is natural choice. Because thick tokens are self-contained, they can be shared between caretakers without needing originators to mediate the exchange. If the receiving caretaker understands the configuration file format, then sending the configuration file to the recipient is all that is needed to complete the exchange. This is easiest when the configuration file format is standardized or at least well documented.

Examples

Example 48.1: Increment Counter Queue

This example will demonstrate an increment counter token used in a collection management scenario.

Problem

Consider the following scenario:

A restaurant has a collection of table sizes: 2-person, 4-person, and 8-person. When a customer arrives at the restaurant, they are seated at a table if a table is available of the appropriate size. Otherwise, they are given a text message when their table is available.

What would be an appropriate design to manage the items in the game?

Solution

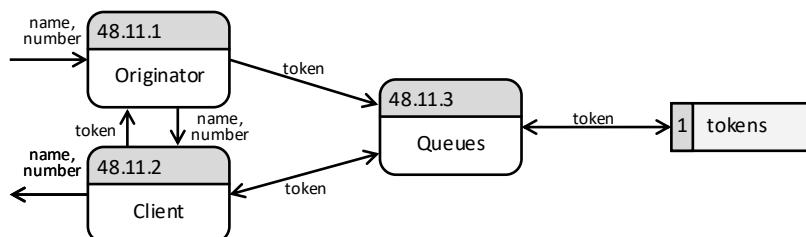
The entity in this scenario is a customer. To avoid duplication, the originator can keep a master list (the entity table) consisting of all the customers who have arrived each day.

The simplest way to refer to these entities are to use the index in the entity table. This utilizes the incremental counter token selection strategy.

The originator is a component accessed by the hostess which enters people onto the waiting list when a table is not available. The enrollment process pairs the customer's name and phone number with a token. The token is then sent to the caretaker.

The caretaker is the system managing the waiting list. This is accomplished with three queues: the 2-person table queue, the 4-person queue, and the 8-person queue.

When a table is cleared, the busboy uses the client to send a notification to the caretaker indicating that a table is available. The caretaker returns with the next token in line. The client then looks up the customer's name and phone number to indicate a table is ready.



Example 48.2: Key Identity Management

This example will demonstrate a key token used in an identity management scenario.

Problem

Consider the following scenario:

A large business would like to create an access control mechanism for all the rooms in the office building. With hundreds of rooms and hundreds of employees, each employee could potentially have access to a unique combination of rooms.

What would be an appropriate design to manage the access control mechanism for this company?

Solution

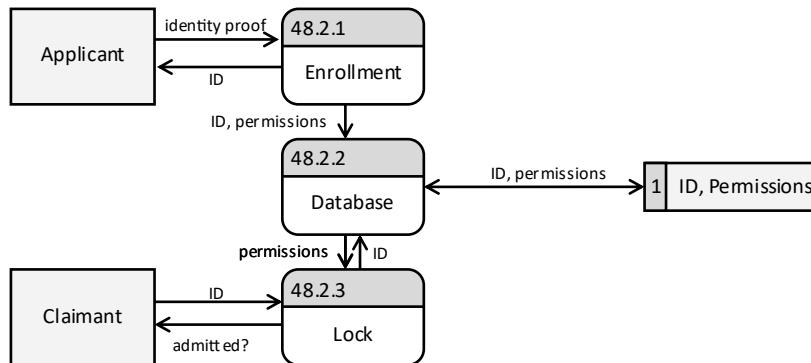
The entity in this scenario is an individual employee.

Though many tokens can be used in this case, the simplest is a key consisting of a randomly generated ID number. With a few hundred employees, a key size of a billion will minimize the chance of someone randomly guessing a valid key. A 32-bit integer would be sufficient for this purpose.

The enrollment process consists of a new employee being given a randomly generated 32-bit integer. This integer will be compared against the known keys in the entity table. If a collision is found, then a new key is generated. This key is then imprinted on the employee's ID card. When an employee requests access to a room, then the card is scanned and the ID number is sent to the identity management system. This scanning process is the originator component.

The caretaker is the database containing all the rooms and the ID numbers that can enter them. The caretaker does not know who is associated with a given ID; it only knows which IDs are allowed into which room.

The client is the component of the lock accepting the admittance/non-admittance signal from the caretaker.



Example 48.3: Object Reference Event

This example will demonstrate an object reference token used in an event notification scenario.

Problem

Consider the following scenario:

A graphics engine can send events to the client using a publisher-subscriber model. The client in this case is a video game wishing to accept keyboard and mouse events from the user. The user's ship accepts mouse events and the game environment accepts keyboard events.

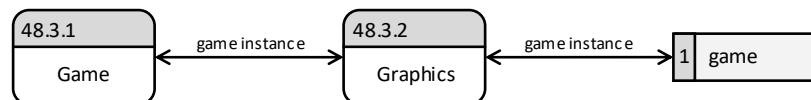
What would be an appropriate design for the graphics engine?

Solution

The entity in this case is the part of the video game wishing to receive input events. The ship object is the entity that subscribes to mouse events and the game environment is the entity that subscribes to keyboard events.

The best token for this scenario would be a reference to the game object itself. This would be a reference to the ship object for the mouse event handler and a reference to the environment object for the keyboard event handler.

The caretaker is the graphics engine, knowing nothing about the application utilizing its services. The graphics engine accepts references to the ship and the environment at initialization time and then sends these objects callbacks when the event transpires.



Example 48.4: Configuration File State

This example will demonstrate a configuration file token used in state persistence scenario.

Problem

Consider the following scenario:

A bible application allows users to retain any number of bookmarks. Each bookmark has a name and the verse in which it refers. The application also allows users to share their bookmarks with other users.

What would be an appropriate design for the bible application?

Solution

The entity is the location in the bible in which the user wishes to return.

A simple configuration file would be a good choice for this token. The file has four components: the bookmark name, the book name, the chapter, and the verse. The JSON data interchange format is a convenient vehicle to encode this token.

<JSON of Proverbs 3:5 with the name “Trust in the Lord”>

The originator uses the current location in the bible to generate a JSON object. This object is then sent to the caretaker.

The caretaker manages the collection of bookmarks and facilitates sharing bookmarks with other users. Note that the caretaker does not look within the JSON object, it just stores and retrieves it.

The client is the mechanism to move the user’s location to the bookmark location.

<DFD>

Exercises

Exercise 48.1: Components

From memory, name and explain the components to entity representation.

Component	Description
Entity	
Token	
Originator	
Caretaker	
Client	

Exercise 48.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification

Exercise 48.3: Token Types

Based on the following description, identify the token selection strategy.

Description	Token Selection Strategy
The social security number (SSN) issued to each American citizen at birth	
Your username when logging into a university's web site	
The now serving ticket you use when waiting in line at the department of motor vehicles (DMV)	
The filename of the document you are using to store your résumé	
The barcode you find on the bottom of your cereal box	
The file used to store your résumé (not the filename, but the actual file)	
Each publication (book, magazine,	

Problems

Problem 48.1: Valet

Identify the token, originator, caretaker, and client in the following scenario. Describe a token strategy which will work best. Include in your description any design details necessary to implement the token generation and retrieval process.

A valet company has been hired to manage parking for a large Porsche Club of America event. About two hundred cars are expected to arrive, each of which needs to be parked when the owner arrives at the event and retrieved when the owner leaves.

Problem 48.2: Restaurant

Identify the token, originator, caretaker, and client in the following scenario. Describe a token strategy which will work best. Include in your description any design details necessary to implement the token generation and retrieval process.

A certain restaurant is very popular, with more potential customers than it can serve. To further complicate matters, customers tend to arrive in a very condensed time. In the past, customers were told to wait in line. Now, the system will develop a new system where customers are notified when their table is ready.

Problem 48.3: Newsletter

Identify the token, originator, caretaker, and client in the following scenario. Describe a token strategy which will work best. Include in your description any design details necessary to implement the token generation and retrieval process.

The local ski resort would like to inform its customers when it receives new snow. To do so, they offer a service whereby interested parties can be informed when new snow arrives. Anyone can subscribe to this service.

Problem 48.4: YouTube

Research how YouTube references individual videos. A video is entirely specified by the uniform resource locator (URL). How does YouTube generate URLs for videos? As a hint, the first ever video is "Me at the zoo."

<https://www.youtube.com/watch?v=jNQXAC9IVRw>

Please answer the following questions:

1. What is the entity?
2. Which entity representation scenario does YouTube follow most closely?
3. What type of token selection strategy does YouTube use?
4. Who is the originator and how does the enrollment process work?
5. Who is the caretaker?
6. Who is the client?
7. Is the token selection strategy appropriate for what it is tasked to do?

Problem 48.5: Social Security

In 1935, President Franklin D. Roosevelt signed the Social Security Act. This law entitles American wage earners to receive benefits based on their contributions. The Social Security Administration (SSA) is tasked with managing the entire process. Please answer the following questions:

1. What is the entity?
2. Which entity representation scenario does the SSA follow most closely?
3. What type of token selection strategy does SSA use?
4. Who is the originator and how does the enrollment process work?
5. Who is the caretaker?
6. Who is the client?
7. Is the token selection strategy appropriate for what it is tasked to do?

Problem 48.6: Bookmark

Bookmarks are a feature present in virtually every web browser since Mosaic introduced them in 1993. This feature enables users to remember a page and the location on a page for later viewing. Please answer the following questions:

1. What is the entity?
2. Which entity representation scenario does a bookmark follow most closely?
3. What type of token selection strategy does bookmarks use in your web browser?
4. Who is the originator and how does the enrollment process work?
5. Who is the caretaker?
6. Who is the client?
7. Is the token selection strategy appropriate for what it is tasked to do?

Challenges

Challenge 48.1: Asteroids

Consider the following video game:

Consider the 1979 Atari game *Asteroids*. You may need to do some “research” to refresh yourself on how this program works. This game consists of several elements: three types of rocks, bullets, a spaceship, and a flying saucer. The game itself has two scores (the current game and the all-time high score) and a counter related to the number of lives remaining.

How would one add the functionality for the user to save a game? Specifically, the user should be able to pause and save any number of games and, on demand, resume a game from that saved state. Please do the following:

1. Identify the entity.
2. Select an appropriate token. Justify your decision.
3. Describe the enrollment process.
4. What will be the task of the caretaker? What tasks will it need to do with the token?

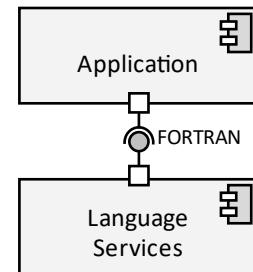
Challenge 48.2:

Layered System Design

Modern applications are created using multiple layers, each layer built on top of the one below.

The first electronic computer was named ENIAC (Electronic Numerical Integrator and Computer), built in the University of Pennsylvania in 1943. ENIAC programmers had full control over the machine; there was no system software or libraries to draw from. Every program was written from scratch in machine language. These programs were painstakingly loaded onto the computer before the computer was turned on. Today, we would call ENIAC a monolithic architecture—a system built from a single component or layer.

In the 1950's, work began on developing tools to facilitate the tedious process of programming computers. Perhaps the most significant outcome of this early effort was the development of the programming language FORTRAN (FORmula TRANslation). FORTRAN represented an important milestone in many regards. One of these was the compiler: a program translating FORTRAN to machine language. The combination of the language and the compiler enabled programmers to work in an abstraction, not needing to be as concerned with the specifics of the hardware on which their programs ran. This represented a two-layer system design with FORTRAN being the communication channel between the layers.



The application utilizes the services in the language

Figure 49.1:
Two-layer system design

Each language provides access to the hardware through APIs supplied by the operating system

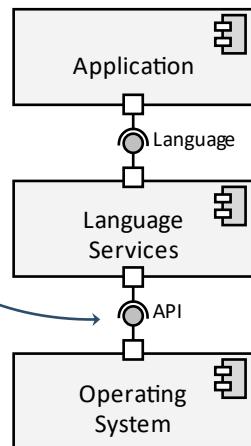


Figure 49.2:
Three-layer system design

In the early days of computing, a programmer would hand a program to an operator who would load the program when the computer was ready. This operator would manually schedule computing time, physically load the punch cards into the computer, and collect the printed output. As computer speed increased and the computer tasks became more complex, software was written to handle the task of the operators. This software was called an operating system, the first of which was GMOS (General Motors Operating System) in 1956. It was not until the IBM OS/360 that an operating system was developed that provided a comprehensive suite of services upon which applications could be built. An application utilizing programming language services that is also built upon operating system services is a three-layered system.

Today, applications are developed with a multi-layer design. Though there are countless ways to design a system, most designs can be categorized into three general architectures:

Architecture	Description
Client-Server	One component makes a request, another responds.
N-Tier	Several components, each built on the one below.
Peer-to-Peer	Many components, each performing the same role.

Client-Server

The server is a component providing services to a client

The original distributed system architecture is client-server, named for the two participating components. A client, otherwise known as a remote processor, front end, presentation layer, or service requester, is a component utilizing system services. In many cases, the client is a user-facing application such as a web browser. In other cases, the client can be a sensor, such as a remote weather station sending data to a server. Either way, the client submits requests to the server and receives data from the server.

The client is a component utilizing the services of a server

A server, also known as a host, back end, data access layer, or service provider, is a component providing functionality to the client. This functionality can be data storage, information processing, communication facilitation, or a combination of these three. A server is frequently labeled by the service it provides. For example, a server hosting a database is called a database server and a server hosting e-mail accounts is called an e-mail server.

In most cases, clients and servers communicate remotely using a transaction-based network protocol. The client packages its request into a message and sends the message to the server. The server synchronously fields the requests and sends a response back to the client from which the request came. This requires each request to have enough information for the server to make the appropriate response. If the client-server communication is privileged or authenticated, then a token or memento is sent to the server identifying the client.



Figure 49.3:
Component diagram of
a client and of a server

Originally, the most common relation between clients and servers was many-to-one; many clients served by a single server. Here, the client was called a terminal and was a special purpose computer whose only job was to connect to the server.

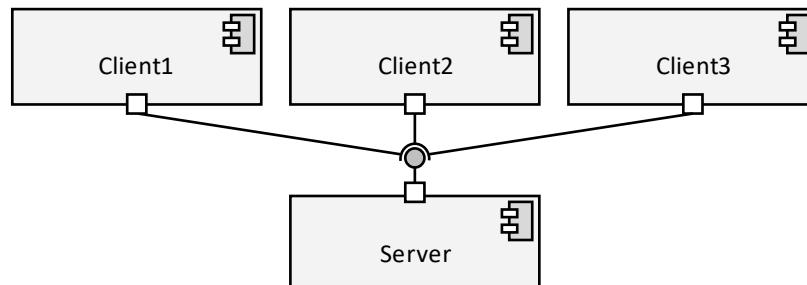
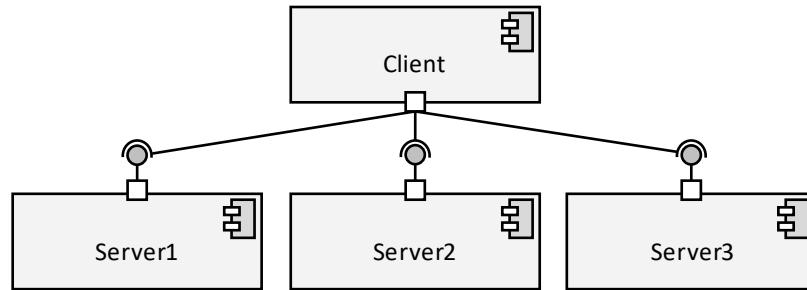


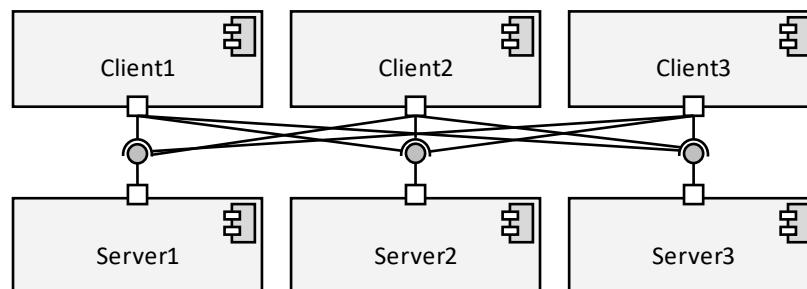
Figure 49.4:
Component diagram of
a many-to-one
client-server architecture

With the advent of the internet, it became more common for a one-to-many relation: one client utilizing several services. Gone are the days when a terminal was dedicated to a single server. Internet users expect to access many websites at the same time.



*Figure 49.5:
Component diagram of
a one-to-many
client-server architecture*

Today, in the highly connected world we live, many-to-many is the most common relation. One mobile device simultaneously maintains a connection with several servers, each of which are connected to several mobile devices.



*Figure 49.6:
Component diagram of
a many-to-many
client-server architecture*

With so many connections possible between clients and servers, it can be easy to create an overly complicated and cumbersome design.

Design considerations

When designing a client-server architecture, three design decisions must be made: where the processing occurs, where the data is stored, and how information is shared between the components.

- **Processing.** All applications perform at least a rudimentary amount of information processing. The following processing questions need to be addressed: Where does the processing take place? Should it be concentrated on the client computer relieving the server of the task? Should it be performed on the server so the client can be without installed code? Should it be somewhere in between?
- **Storage.** All applications utilize data, even if that data is not persisted between sessions. Should the data be stored on the client, on the server, or both? This decision has data accessibility implications in a partially connected environment and where more than one client can modify the shared data.
- **Protocol.** How are the client and the server to communicate? Which network protocols should they utilize and how should data and commands be encoded between them?

Patterns and Best Practices

Though there are countless ways to implement a client-server architecture, there are a few overarching designs which have been proven in a wide range of scenarios. These are the thin client, the thick client, the mirrored client, and the cloud.

Thin Client

A thin client, also known as a dumb terminal or fat server, is a client-server architecture where most of the computational work and data storage occurs on the server. In many scenarios, thin clients do not maintain user data. Any data generated by the client is sent to the server to be saved. It is also common for thin clients to have no application code. This means that any code necessary to interface with the user is supplied by the server and subsequently removed when the task is complete.

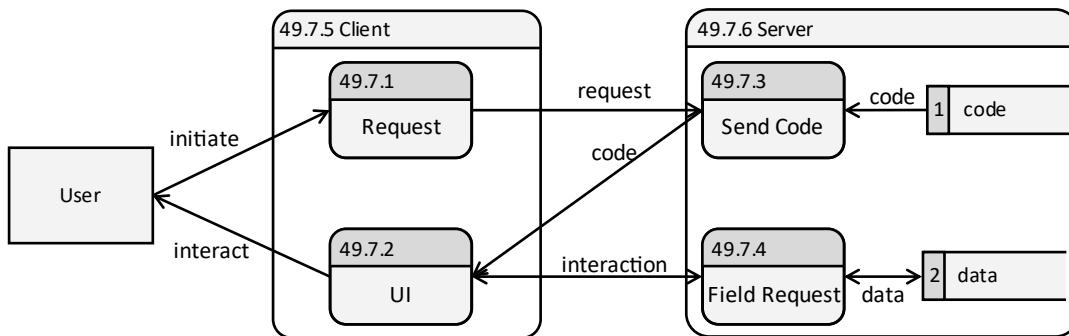


Figure 49.7:
DFD of a thin client

While there are many examples of thin clients in your everyday life, the most common is utilized by the typical e-commerce website. Before the transaction, a client computer has no code from the e-commerce store. When the user initiates a transaction, an initial request is made to the e-commerce server. To facilitate the transaction, the server sends enough code and data to facilitate the transaction. When the transaction is complete, the server code is removed, and the client returns to the initial state. There are several things to notice about this exchange.

- **Recruiting is easy.** The thin client design makes it easy to recruit new participants. Because the client software is minimal, the server only sends a small amount of code and data to the client before the first interaction.
- **Client system requirements can be very light.** Since most of the processing and data storage is handled on the server, very little computing power is required of the client and very little storage space. This facilitates working on less powerful devices as well as enables a single computer to be a client for many systems.
- **Network utilization may be higher than other architectures.** In many thin client scenarios, client code is transmitted over the network. Though this certainly is the source of a lot of bandwidth utilization, it can be minimized or even removed entirely. Another source of network utilization can be harder to remedy. Because the thin client architecture relies on the server for most of the system processing needs, data exchange between the client and server can be high.

Best Practice 49.1 Leverage existing client services when designing a thin client

The functionality and, in some cases, the look and feel of a thin client application can be defined by the existing services on the client. Care should be taken early in the design process to fully understand these capabilities.

Since the server bears most of the computational and storage burden of the application, each new client constitutes an incremental server cost. In other words, the more clients are added to the system, the more work the server must do. These costs must be carefully managed. A client's data footprint must be carefully optimized, and the computational cost of serving a single client request must be carefully optimized.

Thick Client

A thick client, also known as a fat client or a rich client, is a client that performs most of the computation and stores most of the data. The designer of a thick client architecture attempts to do everything possible on the client, connecting to the server only when necessary. This usually involves requiring the client to install and configure software.

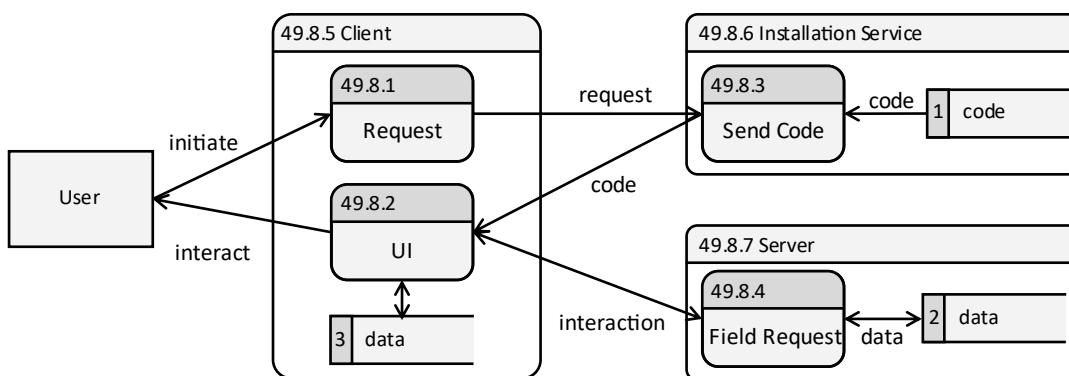


Figure 49.8:
DFD of a thick client

The thick client architecture is the dominant design utilized on mobile technologies today. In most scenarios, the user installation of an application on a mobile device is a separate task from that of using the application. This application does most of the processing and makes periodic exchanges with the server to store or receive data.

In most computing environments, the network is the single largest performance bottleneck. Applications requiring large network throughput or low network latency can feel unresponsive in many scenarios. These interruptions can be more easily controlled when the network traffic is minimized using a thick client architecture.

Per unit cost is the additional load on the server for each new client. This cost can take many forms, but the most common are storage and computing costs. If the number of clients is expected to be very large, then an effort should be made to place as much computational and storage load on the client as possible. This serves to minimize server load, making it possible to accept large increases in the number of clients without requiring substantially more server capability.

In the past, thick clients were not a viable option due to the high cost of computing capability. The past few decades have seen a drastic reduction in computing cost and a proliferation of computing resources. As a result, thick client architecture has become an increasingly attractive design option. We see this today as many of our computing needs are being met by thick mobile applications connected to web services.

Mirrored Client

Many client-server applications are required to operate in an environment where network resources are unreliable. This is commonly called a partially connected environment. In situations like these, the client must be able to function for extended periods without access to the server. The most common solution is to create a proxy on the client which caches important server data and queues information to be sent to the server.

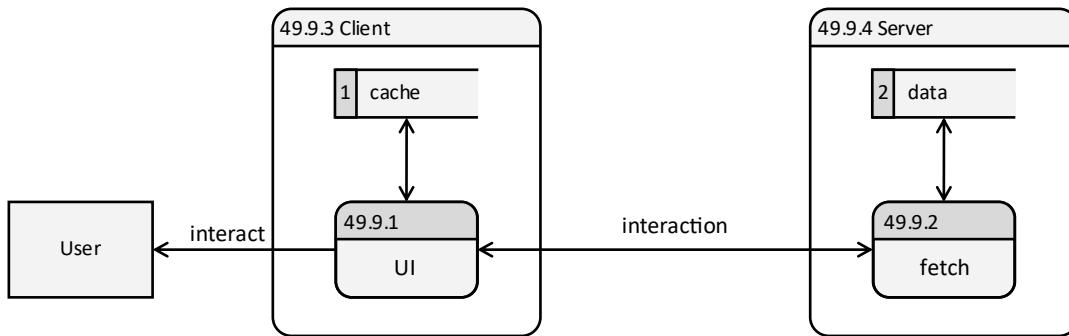


Figure 49.9:
DFD of a mirrored client

If the data interconnection between the client and the server is large, or if the delays between network reconnection are long, caching server data can be complex. The biggest problem deals with reconciliation. What does one do when both the client and the server alter the same data and the changes do not agree? The process of reconciliation in situations like these is called merge conflict resolution.

Consider a calendar system which allows the user to view calendar appointments on a variety of devices. To allow the application to work when no network service is available, the mirrored design is used. This means each client maintains a cached copy of the calendar data, allowing the clients to function even during a network outage. While disconnected, one client changes the location of a meeting to Room 24 and another changes the location to Room 25. When both devices reconnect to the network, which change will win? There are two main strategies for resolving merge conflicts: owner wins and last-write-wins.

- **Owner Wins:** Reconciliation challenges can be minimized if one entity is designated the owner of the data. For example, if there is only one server and the server is designated the owner of the data, then it is always clear which version of the data to use when there are multiple copies. Similarly, it is possible to designate one client as the keeper of the master copy. In each case, merge conflict resolution can be simplified when one entity always wins.
- **Last-Write-Wins:** A common merge conflict resolution strategy is called last-write-wins. In other words, the most recent version of the data is retained in the case of a merge conflict. This requires two things. First, a timestamp of the edit must accompany the data as it travels between the client and the server. This timestamp is based on Universal Time Coordination (UTC), a reference clock of all nodes on the system. Second, all system entities must agree on the time. Clock synchronization can occur through an external source such as the National Institute of Standards and Technology (NIST) internet time service, or through internal sources. There are several strategies for internal clock synchronization, including the network time protocol (NTP), global averaging algorithm, and localized averaging algorithm.

The details of implementing various merge conflict resolution strategies are beyond the scope of this book. However, any system contemplating a mirrored client architecture must develop a reconciliation strategy.

Cloud

The cloud is a client-server model where customers rent computing services rather than buy or build them themselves. In other words, cloud computing is a business model rather than a software architecture model. There are three cloud models:

- **Infrastructure-as-a-Service (IaaS):** IaaS is a model where a customer rents the physical computers used to host servers. From a software architecture perspective, this has one important implication. The thin client architecture may seem unattractive as it places a heavy load on the server as new clients are added. IaaS provides a clear upgrade path so the server can be assumed to have unlimited computational capacity.
- **Platform-as-a-Service (PaaS):** PaaS is a model where the customer rents the physical computers used to host servers as well as components of the server. This could include databases and related tools. As PaaS becomes more common, it introduces pressures to utilize standard components when designing servers.
- **Software-as-a-Service (SaaS):** SaaS is a model where customers rent client applications as well as servers. Venders providing SaaS subscriptions provide installation, update, and support services to customers. Some SaaS architectures utilize thin client strategies (providing customers with web access to common tools) while others use thick client (allowing customers to download applications).

Since cloud architecture is a business model rather than a software model, what software design decisions can best support this system?

Best Practice 49.5 Prefer thin client over thick to support cloud architecture

A key aspect of cloud services is the ability to add and remove functionality according to the business needs of the customer. Complex and time-consuming client installation procedures can make it more difficult to support the subscription model. When faced with multiple technological options, favor thinner clients.

Best Practice 49.6 Manage subscription services carefully

A key component of all cloud services is to provide the customer with the desired level of service in an environment where the level of service can be in flux. Thus, each request for service must be verified against the level of service to which the customer is currently enrolled. This is accomplished with a query to the subscription service. This service must meet the following requirements:

- The subscription service must operate in a partially connected environment so the customer can receive service when the subscription server is unavailable.
- It must be non-intrusive. The customer has little tolerance for service denials when the denial came from the service meant to grant access.
- It must resist attack. Though most customers are willing to pay for the service they consume, some may choose to steal it if the path is easy. For this reason, special security considerations should be made on the subscription service system.

N-Tier

A monolithic design is a design where one component can complete a task in its entirety. While users often think of applications as monolithic, they are rarely built this way. A 2-tier design is a design where one component is built from services provided by another. Notice that the client-server pattern is a 2-tier design on its own. Since virtually all applications make use of operating system services, even “hello world” can be considered a 2-tier design. Applications with greater than two levels of design are called *n*-tier designs. Since operating systems themselves have multiple tiers, it turns out that “hello world” is an *n*-tier design.

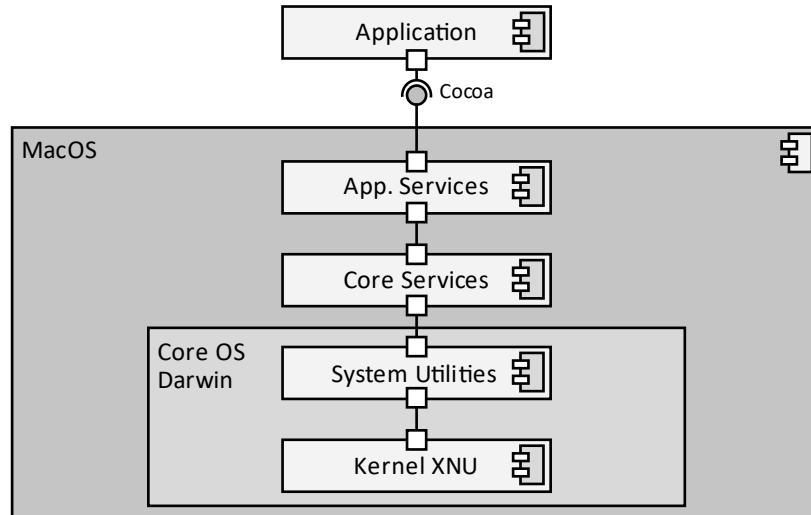


Figure 49.10:
Component diagram
of Mac OS

Notice how there are many layers in the operating system between the program and the hardware, most of which are unseen to the application developer.

Network communication is also built upon an *n*-tier design. There are seven tiers defined by Hubert Zimmermann in 1980. This design is called the Open Systems Interconnection (OSI) reference model.

Application	Originator and consumer of messages
Presentation	Encoding
Session	Conversation
Transport	Message integrity and resource minimization
Network	Routing
Data Link	Point to point communication
Physical	Medium through which data travels

Figure 49.10:
The OSI reference model

Each layer in the OSI reference model communicates only with the layer above and below it in the stack. Additionally, each layer makes no assumptions about how the layer below works, or how the layer above it uses the layer's services.

Characteristics

The Macintosh OS and the OSI reference model have three common characteristics: multiple layers, layers form a cohesive stack, and layers only communicate with neighbors.

Multiple Layers

N-tier designs are characterized by having multiple layers. These layers are also called tiers and components, representing a logical subdivision of the program. As with other multi-component designs, it is not uncommon for the layers in an *n*-tier design to be built with different technology and to communicate through standard protocols. While client-server designs have two layers, *n*-tier designs can have any number of layers between the client-facing interface and the low-level services. The bottommost layer is called layer 1, the topmost is called layer *n*.

Stack

The layers in an *n*-tier design are in an ordered collection called a *stack*. Do not confuse an *n*-tier stack with a stack data structure. Recall that a stack data structure is a first-in last-out collection of items. This certainly does not apply to an *n*-tier stack because tiers are not added or removed from the stack. Instead, an *n*-tier stack is a collection of services laid on top of each other, each tier being built upon the tier below. Note that the term bundle or suite would be more appropriate than stack, but stack is the standard term. For example, the LAMP server stack is built from Linux at the base, with Apache web services sitting on Linux, and MySQL database sitting on the Apache layer, followed by PHP at the top layer.

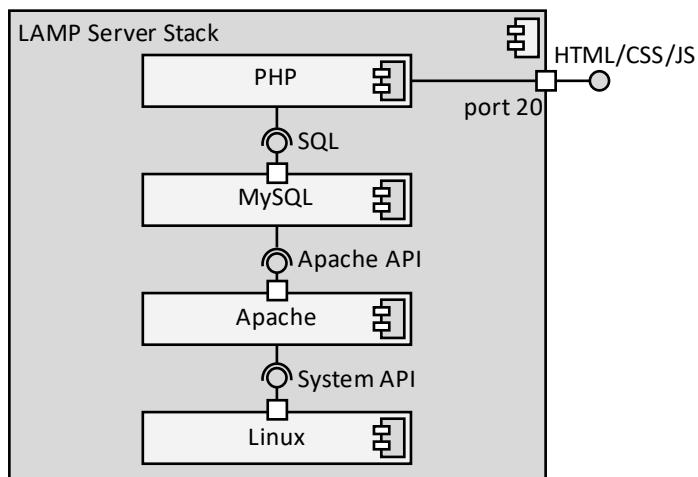


Figure 49.11:
Component diagram
of the LAMP stack

Communicate with Neighbors

The stack tiers are ordered in such a way that each layer only communicates with the tiers above and below it in the stack. In other words, each tier is built on top of the tier below. The third tier, for example, can only access services on the second tier, not the first or forth. In other words, a given tier can have no knowledge of how many tiers lie below it, nor how many tiers are above it in the stack. Some *n*-tier designs relax this constraint. These are called open layer architecture.

These three elements define *n*-tier or multi-tier architecture. *N*-tier architecture is defined as a system where the different functions are physically or logically separated, and each separation is in an ordered stack.

Variations

Though there is only one basic layout of an n -tier design (each layer sitting on top of the one below), there are four variations of communication flow within this design: top-down, bottom-up, partial traverse, and horizontal stack.

Top-Down

Top-down communication flow is characterized by the client initiating a request to the top-most tier (called layer n) and that request percolates through the layers until it reaches layer 1. Each layer m generates one or more request to layer $m-1$. An example of top-down communication is how an application would draw a pixel on the screen. The application initiates the request to the graphics layer (OpenGL in this case). The graphics layer then passes requests to the rendering component of the operating system (Linux's direct rendering manager (DRM) in this case). The operating system then sends several requests to the computer hardware (the graphics processing unit (GPU) in this case).

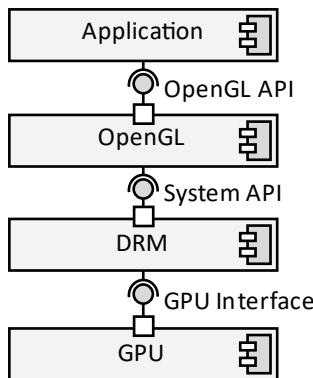


Figure 49.12:
Component diagram
of the OpenGL stack

Bottom-Up

The bottom-up communication flow is also known as the notification pattern. Here, layer 1 receives a notification and it is sent to one or more layer 2 services. These notifications continue to percolate up the stack until high-level messages are sent to the client. An example would be a cellphone which lost its cellular service. The message would percolate up from the cellular driver to the network hardware abstraction layer (HAL). This would be sent up to the secure sockets layer (SSL) platform library. Since the application interfaces directly with the SSL, it would receive the loss of service notification directly from the SSL.

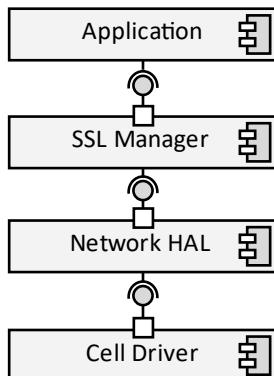


Figure 49.13:
Component diagram
of the Android SSL stack

Partial Traverse

A partial traverse occurs when layer n receives a command from the user and it travels down the stack until it reaches a level that can handle the request. Because the request is handed, it never makes it to the bottom of the stack. An example of this is a cache. An application requests a resource that is located on a remote device. The request is sent to the operating system file manager. The file manager recognizes that the resource is in the cache, so it is returned directly to the application. The request is never sent to the network manager or to the remote server.

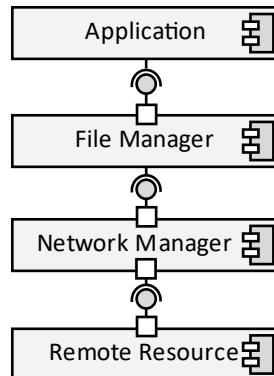


Figure 49.14:
Component diagram
of file caching

A partial traverse can happen both with a top-down communication flow and with a bottom-up notification.

Horizontal

The final design pattern occurs when multiple services exist at a single layer. In cases such as these, complementary services may need to communicate with each other. The classic example is the TCP layer of the OSI reference model. One of the services provided by the TCP layer is the guarantee of packet delivery. This is accomplished by the recipient verifying that all the packets associated with a given message are received. If a packet is lost, then a new request for the missing packet is issued. It may take several interactions between TCP services to facilitate the passage of a single message.

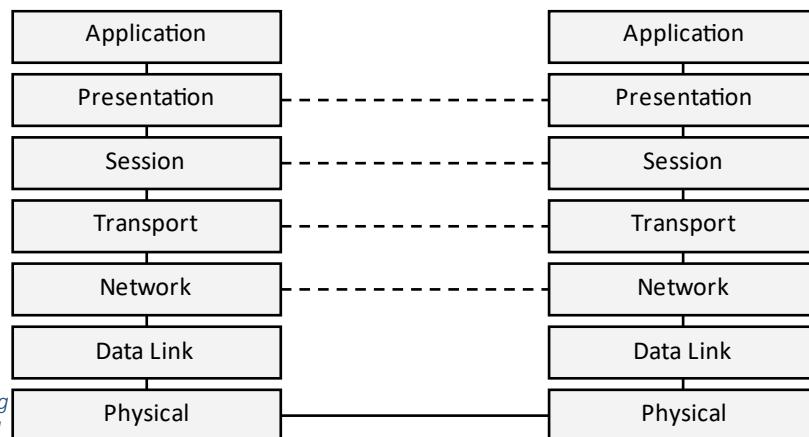


Figure 49.15:
Horizontal message passing
on the OSI reference model

Horizontal message passing can happen at several layers in the OSI model, including the presentation (through exchanging encryption keys), session (through exchanging session keys), transport, and network (exchanging routing information).

Benefits and Challenges

The *n*-tier design is one of the most common large system design patterns, present to some degree in virtually every operating system, large application, and server. The popularity of the *n*-tier design can be attributed to its many recognized benefits.

- **Increased Cohesion:** When each layer is considered as an independent system or service, it is easy to achieve component cohesion. This is also known as separation of concerns.
- **Decreased Coupling:** Since each layer only communicates with the layer above or below it, it is easy to simplify and reduce information interchange between components. This tends to yield looser coupling.
- **Isolated Changes:** Code and design changes made at one layer have minimal or no impact on layers above or below the change. This decreases the testing burden.
- **Interchangeable Components:** Two versions of a component with equivalent functionality should be interchangeable. In other words, a component can be rewritten with a completely different design philosophy and using completely different technology. If the new component honors the same contract as the original, it should be able to exchange the components without impacting the operation of the overall system.
- **Encourages Reuse:** A cohesive layer with a well-defined interface facilitates reuse in other stacks. This can yield lower development cost, lower testing costs, and lower maintenance costs.
- **Encourage Standardization:** Standardization is the process of documenting and publishing interfaces for the purpose of encouraging the interface to be used in other contexts. In short, standardization results in a lot of paperwork. This increased scrutiny on the component interfaces is usually time well spent. It results in more understandable, convenient, and abstract interfaces. Publication of standards also encourages other teams to develop components which can be incorporated in a given design. In other words, standardization facilitates interchangeable components.

Though the *n*-tier design has many benefits, there are several situations in which it may not be best tool for the job. First, not all applications are sufficiently complicated to benefit from multiple layers. In other words, there exists the risk of overengineering. Second, it is difficult to debug across layers. Because each layer is isolated from neighboring layers, it is relatively easy to find and fix bugs within a given layer. However, things can get complicated when bugs span multiple layers. Because each layer can be a separate application utilizing separate technology, a single debugger cannot step across a layer boundary. This problem is called “thunking” and presents one of the most challenging debugging tasks.

Guidelines

A system subdivided into meaningful layers can be much easier to develop and maintain. The converse is also true. A poorly engineered layer system can severely and adversely affect the entire project.

Best Practice 49.7 Carefully consider interface changes

Interfaces between layers should be carefully considered. These should reflect the task the layer is to perform, making no reference to how the task is to be performed. In other words, a layer should be a “black box” to the layer above it, revealing nothing about how it performs its action.

Best Practice 49.8 One task should not cross a component boundary

Each task should be completely assumed by a single component. If a task crosses a component boundary, then perhaps the boundary should be dissolved. Note that it is acceptable for one component to utilize the resources of a lower-level component to complete a task. That is not a problem. However, if the task itself spans multiple components, then the overall system design should be reanalyzed.

Best Practice 49.9 Each component should only communicate with adjacent components

Recall that a defining characteristic of the *n*-tier design is that each layer can only communicate with the layers above it and below it on the stack. If this constraint is violated, then a layer would be made aware of specific implementation details of neighboring layers.

Best Practice 49.10 Use common and recognizable layer names

Though an *n*-tier application can have any number of tiers performing any number of services, most designs select from a small number of services. These services are the following:

- **Presentation Tier:** This is the top-most tier and contains the user interface. The main purpose of the presentation layer is to translate user intent into tasks to be sent to the system and translate system data to be in a format understandable by the user.
- **Logical Tier:** This middle tier, also known as the business layer, business logical layer (BLL), and domain layer, makes calculations, performs validation, and contains the policy of the system. In other words, this layer governs how the system behaves.
- **Data Tier:** This layer, also known as the persistence layer or data access layer, interfaces with the database or any other system which stores the system data.
- **Platform Tier:** This tier hosts fundamental services required by the system. Examples include a file system, database, directory service, hardware interfaces, or network services.

Peer-to-Peer

A peer is defined as an individual of the same rank, status, or social group as another. Peer-to-peer architecture is a network of peers where each member has equal privilege and fulfills the same role. For example, consider a collection of students and an instructor in an academic setting. The instructor and the students are not peers because they fulfill different roles. One could think of them as a client-server network where the instructor is the server and the students are clients.

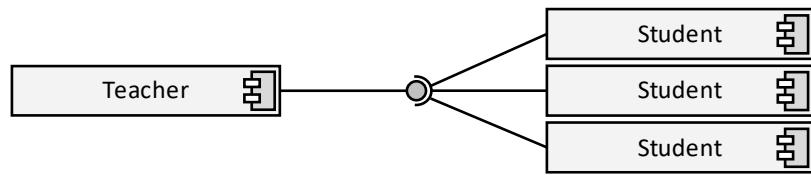


Figure 49.16:
Component diagram of a
client-server design

The collection of students, on the other hand, are peers with each other. If they intercommunicate during class or form a study group, we have a peer-to-peer network. The collection of peers is called a peer group or simply a group.

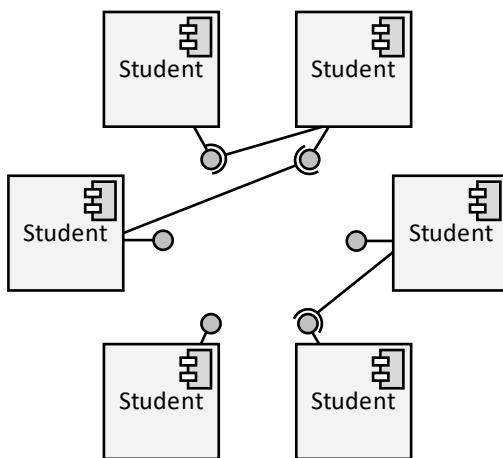


Figure 49.17:
Component diagram of a
peer-to-peer design

Each peer in a peer group operates in the same role. This is called homogeneity. A peer-to-peer system must provide client, server, and routing services.

- **Client:** A client initiates requests for services. Since all peers perform the same function, each peer in a peer group must have the capacity, if not the need, to request for service. If no peer requests service, then no peer will respond and there will be no traffic on the network.
- **Server:** A server responds to requests for service. Each peer must have the capacity, if not the need, to respond to a request. The request can come from any peer in the network.
- **Routing:** Client-server relations have well-defined communication links. The same can be said for n -tier designs. However, peer-to-peer designs do not have well-defined communication links. Commonly, peers can enter and leave the network at any time. Thus, each peer needs to be able to find a peer that would be capable of fulfilling the request. There are several ways this can take place, depending on the design of the peer-to-peer network.

Variations

There are three variations of peer-to-peer designs: ad hoc, orchestrated, and distributed.

Ad-Hoc

The term *ad-hoc* has two definitions. The first is: when necessary. This implies that an ad-hoc action is one that was not previously planned or anticipated. The second is: not generalizable. This means that an ad-hoc solution to one problem cannot necessarily be applied to another. Both definitions apply to ad-hoc peer-to-peer connections.

Ad-hoc communication occurs when two parties need to share data in a way that was not anticipated by the system designers. There may be many peers in the group, and any peer can initiate a connection with any other peer.

It is easy to find examples of ad-hoc designs in our everyday lives. Consider Apple's AirDrop and Android's Nearby Sharing. Both features allow one mobile device to share files with another without going through a centralized server. If Alice would like to send a picture to Bob, then Alice's device operates as the client (initiating the connection) and Bob's as the server (accepting the connection request). Routing occurs as Alice's device broadcasts her desire to share with all the devices in the area and Bob's device responds with an acceptance message. At this point, an ad-hoc connection is established between the two devices.

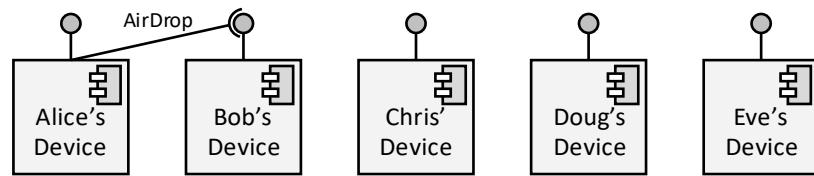


Figure 49.18:
Component diagram of an
ad-hoc design

This design is peer-to-peer because all devices can initiate or receive a sharing request. It is also ad-hoc because the connection is spontaneous.

Orchestrated

Orchestrated peer-to-peer systems occur when many components play the same role (and are thus called peers), but another system performs routing services. In other words, it can be difficult for one peer to find another peer with the needed data or capability. One way to address this difficulty is to have a third party called a conductor. A conductor is an entity aware of the capabilities and possessions of the individual peers in the group. A peer needing data or services contacts the conductor for the address of a peer who can help. When the requesting peer receives this address, then the requesting peer directly contacts the responding peer. Thus, the conductor does not moderate the conversation between the peers. Instead, it only orchestrates the connection.

An example of an orchestrated peer-to-peer system is a chess game allowing two players to share a game. When a player is looking for a partner, he or she will go to a service where potential players gather. Once a partner is found, the two players begin their game. Note that moves are shared directly between the two players, not using the chess services as a middleman.

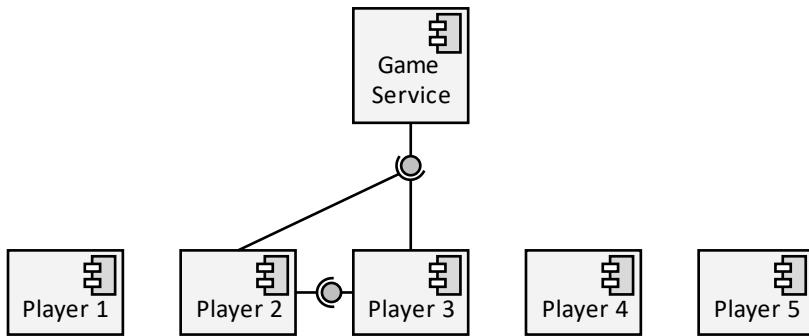


Figure 49.19:
Component diagram of an
orchestrated design

This example is peer to peer because each player has the same role and they communicate without going through a server. It is orchestrated because there is a service connecting players serving as the conductor.

Distributed

A distributed system is a system architecture where different components work on the same problem in a coordinated and often collaborative manner. There are several parts to this definition. First, distributed systems involve many components, each of which fulfills essentially the same role. Therefore, distributed systems are a type of peer-to-peer design. Second, distributed systems work on the same problem. Problems suited to distributed systems are ones that are computationally intensive and can be readily subdivided. This is especially true when the subtasks require minimal or no intercommunication between peers. Finally, distributed systems are often highly coordinated by a conductor. This makes distributed systems a special type of orchestrated design.

One of the earliest examples of a massively distributed computing system was the search for extraterrestrial intelligence (SETI) project in 1999. This project was orchestrated by the University of California, Berkeley. The conductor collected radio telescope observations and packaged them into distinct data sets. Volunteers would install a node called SETI@home which would accept radio data, process it looking for signs of intelligent life, and send the results back to the conductor's servers. In two decades, almost two million volunteers participated in this project.

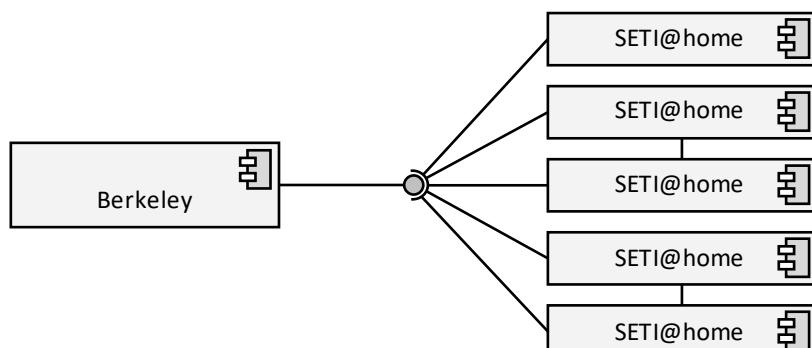


Figure 49.18:
Component diagram of a
distributed design

This design is peer-to-peer because each of the SETI nodes performs the same role and there can be a degree of intercommunication between the nodes. It is distributed because the various nodes are coordinated by the Berkeley servers to work on a single problem.

Examples

Example 49.1: Client-Server

This example will demonstrate how to recognize client-server design.

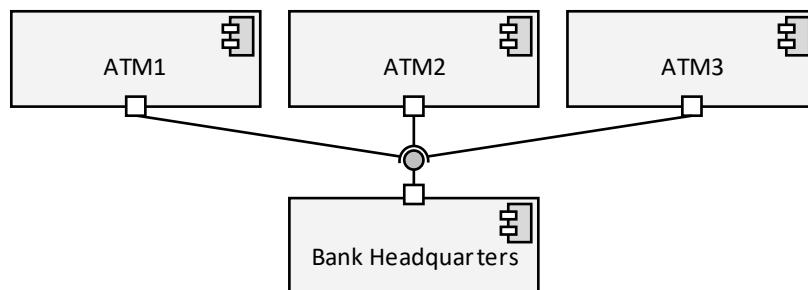
Problem

Describe the system design implied in the following scenario:

A bank's ATM is connected to the headquarters' computers when balance inquiries are made or when a transaction is processed.

Solution

This is an example of a simple client-server design. In this case, the server is the headquarters' computer and the client is the ATM. Note that the bank probably has multiple ATMs in the network. Each ATM has a direct connection to headquarters. This means we have a many-to-one relation between the clients and the server.



From a data standpoint, the ATM represents a thin client. The ATM retains no data once a session is complete; the headquarters computer possesses the sole copy of account data. Note that the ATM cannot function in any capacity in the event of network disconnect.

From a processing standpoint, the ATM represents a thick client. The ATM has installed software designed to moderate the interactions with the user. These interactions are converted into discrete transactions that are easy for the headquarters computer to process.

Example 49.2: N-Tier

This example will demonstrate how to recognize *n*-tier system design.

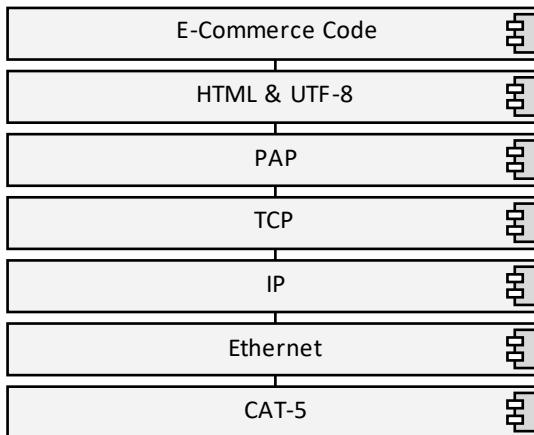
Problem

Describe the system design implied in the following scenario:

An e-commerce server is connected to a customer's web browser using HTML using password authentication protocol (PAP) to handle session state. The client's computer is connected to the internet using a CAT-5 ethernet cable.

Solution

Computer networks use the OSI reference model, which is a seven-layer design. The application layer is the web browser and the e-commerce's client-side code. The presentation layer is HTML and UTF-8. The session layer is PAP. The transport layer is the TCP protocol on the internet and the network protocol is IP or IPv6. There are many data link layers utilized in this transaction as packets travel between the client's computer and the server. However, the problem made specific mention of ethernet. Finally, the physical layer is the ethernet cable itself which is CAT-5, a twisted copper pair cable.



This design utilizes top-down communication flow as the user initiates actions through the shopping process. It probably also utilizes bottom-up as the server sends update information about the status of the order or the purchasing process. There is a small amount of partial traverse as the various internet services perform error checking and other maintenance tasks (such as the various gateway nodes updating their routing tables through the routing information protocol (RIP)). Finally, there is a degree of horizontal communication flow between most of the layers.

Example 49.3: Peer-to-Peer and Client-Server

This example will demonstrate how to recognize peer-to-peer system design.

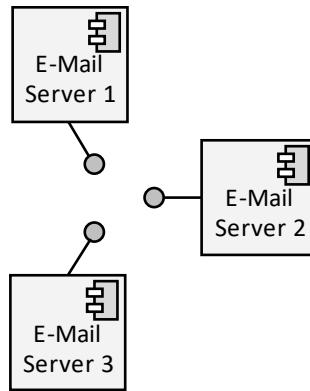
Problem

Describe the system design implied in the following scenario:

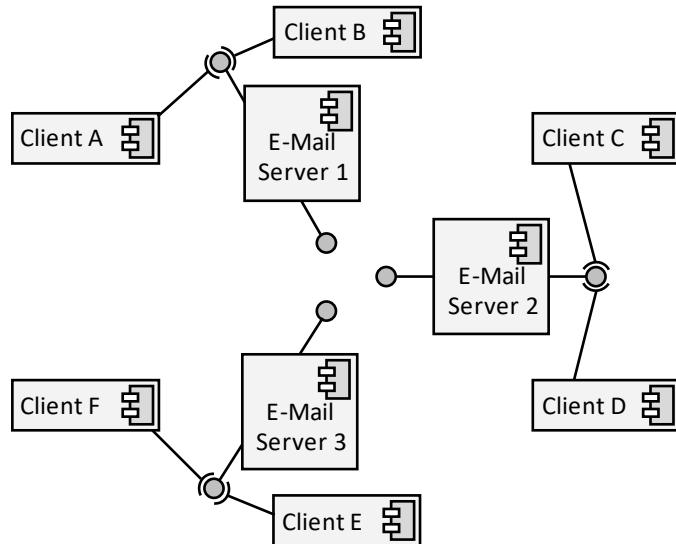
An e-mail is sent from outgoing@originator.com to incoming@recipient.com.

Solution

On the surface, this appears to be a simple ad-hoc peer-to-peer design: any individual can send an e-mail to any other e-mail on the network. Since an e-mail account can both send and receive messages, an account is both a client and a server. Routing occurs as e-mail servers send outgoing messages to the appropriate e-mail server recipient based on the host name embedded in the e-mail address.



Notice that e-mail is usually sent from an e-mail client, not from an e-mail server. This means that a client-server relation also exists.



Exercises

Exercise 49.1: Three Architectures

From memory, name and define the three layer design architectures.

Name	Define

Exercise 49.2: Fact or Fiction

For each of the following, identify whether it is fact or whether it is fiction. For each one, justify your answer in the space provided.

Fact or Fiction	Justification
The service coordinating an orchestrated peer-to-peer network is called the conductor	
A given layer on an <i>n</i> -tier design can only communicate with neighboring layers	
The topmost layer is called layer <i>n</i> , the bottommost layer is called layer 1	
A thick client relies on the server to do most of the processing and data storage	
A client-server design is a type of peer-to-peer design	
Cloud computing is mostly a business distinction, not a software design pattern	
The collection of layers in an <i>n</i> -tiered design is called a stack	
An <i>n</i> -tiered design is a type of client-server design	

Problems

Problem 49.1: Elementary School Files

Describe the system design implied in the following scenario:

The principle at an elementary school would like to provide file backup and remote file access services to all the teachers. To accomplish this, a local area network (LAN) is created that connects each computer. In the principle's office sits a single computer with a large amount of storage capacity. All the teachers are instructed to keep a copy of their school files on this computer.

Problem 49.2: WAMP

Describe the system design implied in the following scenario:

The Microsoft version of the LAMP web services stack is called the WAMP.

Research the WAMP stack, describe it using a component diagram, and describe the type of processing that occurs at each layer.

Problem 49.3: Napster

Describe the system design implied in the following scenario:

Napster was a music sharing service initiated in 1999. It allowed individuals to post their music collection online and download the music owned by others.

Note that some research may be required to solve this problem.

Problem 49.4: Mobile Device

Consider the mobile device you use most often. Characterize the layered model utilized by the network component of your mobile device. Create a component diagram describing how it works. Finally, critique the design decisions according to the best practices and metrics contained in this chapter and elsewhere in this book.

Challenges

Challenge 49.1: Google Documents

Research Google Documents. Characterize the layered model utilized by Google Documents. Create a component diagram describing how it works. Finally, critique the design decisions according to the best practices and metrics contained in this chapter and elsewhere in this book.

Challenge 49.2: Office 365

Research Office 365 documents. Characterize the layered model utilized by Office 365. Create a component diagram describing how it works. Finally, critique the design decisions according to the best practices and metrics contained in this chapter and elsewhere in this book.

Appendix

Glossary

Appendix A

abstract component	One component in the decorator design pattern, the abstract component is the class that contains the task method. This class serves as a link or node in the chain of tasks to be executed
abstract factory	A creational design pattern that is a collection of factory classes. Each derived class is customized to the specific needs of the client
abstract strategy	One component in the strategy design pattern, the abstract strategy is the abstract base class from which all concrete strategies are derived. This class defines the interface for the algorithm to be abstracted
abstract template	One component in the template method design pattern, the abstract template is the base class that contains the template method
abstraction	A software quality metric describing the amount the client needs to know about the implementation details of a class to use it effectively
acceptance test	A type of test (along with system test, integration test, and unit test) which captures what the stakeholder needs the system to do
activity diagram	A graphical tool used to represent algorithms. An activity diagram is closely related to a flowchart
ad hoc	Generally, ad hoc means that something is not planned or anticipated. The term also refers to a specific type of peer-to-peer design where two parties need to share data in a way that was not anticipated by the system designers
ad hoc driver	A type of manual driver where a few lines of code are added to quickly test a function
adaptability	A software quality metric describing the suitability of the base class in an inheritance hierarchy to facilitate the creation of derived classes which can fulfill previously unknown or unanticipated design concerns
adapter	A class exhibiting the interface that the client expects while also containing a class that the client needs
adjustable	A level of the malleability metric of software quality, adjustable means that alterations to the algorithm can be made by changing local elements but not refactoring the existing code
aggregate	An item formed by combining several individual elements. In the context of software, aggregates are most commonly associated with aggregate data types such as collections
aggregation	When the member variable of one class is a reference to an object of another class
agile	A class of development methodologies first described in 2001. Agile favors short development cycles, intimate involvement of the client in development activities, and minimal overhead
algorithm	A sequence of instructions designed to accomplish some task
algorithm abstraction	The process of taking a potentially complicated algorithm and reducing it to its components, allowing the components to be reused and reassembled to create new combinations
algorithmic efficiency	A measure of how program execution time is related to input size. The most common metric is big-O
alignment	A software quality metric describing the degree in which the inheritance tree models relationships in the problem domain
anonymous	An adjective applied to a variable, member variable, function, or other programming construct which usually has a name but, in this instance, does not
alpha testing	A form of acceptance testing conducted by specialized testers who systematically analyze a system
applicant	A role in the authentication process representing an individual who is not currently a subscriber but aspires to be one

application	A software entity like a program. In many contexts, the term application, program, and executable are considered synonyms
application programming interface	Otherwise known as API, this is generic term referring to any interface between components of software. An API could be one or more functions, one or more classes, an RPC interface, or even a port
architect	One who writes computer code, who is experienced and wise, and who is responsible for system-wide design decisions. Most consider an architect as being the most highly skilled programmer
array	An aggregate data type consisting of a collection of homogeneous elements stored in a continuous region of memory facilitating random access
ASCII	Acronym standing for the American Symbolic Code for Information Interchange. This is a standard mapping between integers and letters. The capital letter A, for example, is 65 in the ASCII table.
assembly code	A human-readable version of machine code consisting of register designations and memory locations
assembly connector	A mating between a required and a provided interface on a component diagram
assert	Checks placed in a program representing an assumption the developer thinks is always true
association	When the member variable of one class is a reference to an object of another class
attribute	The data of a class, also known as a member variable
authentication	The process of providing individuals are who they say they are in an identity management system
automation	Types of drivers where a function or program generates test cases and verifies results
back door tests	Another name for white box tests, back door tests have access to implementation details of the unit under test
back matter	The end of a large document such as an SRS or SDD including appendices, the glossary, a list of abbreviations, and anything else that may be needed to explain the intent of the design
base class	A class that others inherit from. It is the basis of inheritance
beta testing	A form of acceptance testing involving exposing the software project to real end-users in their environment
big-O	A performance metric where execution time is a function of input size
binding	The process of tying named programming entities to physical locations in memory
binding table	A table maintained by the compiler (for early binding) or during runtime (for late binding) which maps programming entities to physical locations in memory
black box testing	Black box testing is a validation and verification process where only two aspects are considered: the input into the system and the output received from the system
block	A programming construct meant to delineate or minimize scope. By defining a variable in a block, the programmer is limiting the scope to the bounds of the block
bookmark	A token in a state persistence scenario representing a configuration of the system
Boolean algebra	The process of analyzing and simplifying Boolean expressions
Boolean expression	An equation or expression that always evaluates to true or false
boundary condition	Locations at the border of equivalence classes, boundary conditions represent test cases that should be validated
branch prediction	The process of optimizing a CPU architecture so it executes conditional jumps (IF statements) more efficiently
breakpoint	A debugger tool enabling the programmer to halt execution of the program at a specific moment
bridge	An interface design pattern where the interface is developed specially for the client and the implementation is designed to both fulfill the provider's services and easily integrate with the interface
builder	A creational design pattern used to facilitate construction of object in distinct phases
business logic	The part of an application, program, or class that represents the rules that determine how data is manipulated
by-reference	A method of passing data to a function through a parameter. By-reference parameters can be set by the caller or the callee. In other words, they facilitate two-way data movement

by-value	A method of passing data to a function through a parameter. By-value parameters are only set by the caller, any change by the callee will not be sent back to the caller. In other words, they facilitate one-way data movement
C string	A method for storing text, a C string is defined as an array of characters where the last element is a null-character
call	The process of one function executing another function. This is also called “invoking”
callback	A type of function pointer enabling a programmer to specify which function is to be called at runtime
callee	A function that is invoked by a caller
caller	The entity invoking a given function
call stack	A data structure storing the parameters, local variable, and return addresses of all the active functions in a program at a given moment of execution. The first item on the call stack is main() or whatever the equivalent may be in your programming language of choice. The call stack feature is also a debugger tool enabling the programmer to view the called functions from main at a given moment in the execution of a program
can	A prefix used with a function or method name indicating the function or method is a predicate. The can prefix indicates readiness to perform an action
caretaker	The part of a system which manages collections of tokens
chain of responsibility	A message passing design pattern characterized by a collection of recipients each having the opportunity to handle an event
checkpoint	A token in a state persistence scenario representing a configuration of the system
circular integration	A separation of concerns reassembly strategy where the interface sends messages to the business logic component. The business logic component sends messages to the storage component. Finally, the storage component sends messages back to the interface
claimant	A role in the authentication process representing an individual claims to be a subscriber but has yet to be verified
class	A program entity consisting of data and the operations that act on them
class cohesion	A software design metric comparable to fidelity capturing the degree in which the methods of a class operate on the same data
class diagram	A graphical tool used to represent the attributes and operations associated with a class. It is also used to describe how multiple classes relate to each other. The class diagram is perhaps the most widely used design visualization tool
classifier rectangle	A part of a component diagram representing that which is contained within a component. In many ways, a classifier rectangle in a component diagram is similar to a trust boundary in a DFD
client	The component of a software system that consumes the services provided by another component. A client using a function, a class, a component, or even an entire software system
client-server	A system architecture involving two components: one to make requests and one to respond to requests
cloud	A client-server model where the customers rents computing services rather than build or buy them themselves
code	Instructions written in a programming language that can be compiled into an application. Code is also known as software or source code
codebase	All the code associated with an application
codepath	The path that the program follows through the codebase through an execution. If each line of code were numbered, then the codepath would constitute the sequence of line numbers through execution.
cohesion	A software quality metric describing how well a unit of software represents one concept or performs one task
coincidental	A level of cohesion identified by Larry Constantine where there is no meaningful relationship among the elements in a module. This is similar to the modern level of weak cohesion
collaborator	From the context of a CRC card, a collaborator is a class or component of the system that relates in some capacity to the class under design. A collaborator may be related by inheritance, composition, association, or

any other type of class relation. From the context of a unit test, a collaborator is another unit which may influence the unit under test.

collection	A collection is a group of variables under a single variable name. Commonly used collections include structures, tuples, arrays, lists, and dictionaries
collection-controlled loop	A type of loop where all the elements of a collection are visited
collocate	The process of collecting items into a single location. From a software development perspective, this means collecting statements into functions, variables into classes, etc.
component	A subdivision of a software system representing a collection of units which are independent of the rest of the program. A component is a modular unit designed to be upgradable and replaceable. Frequently, components can be separately compiled, run on different hardware, or even be replaced without affecting the rest of the system
concrete component	One component in the decorator design pattern, the concrete component is a class that derives from the abstract component and implements the task method. This is the base task that serves as the endpoint in the decorator chain of tasks
concrete decorator	One component in the decorator design pattern, the concrete decorator represents a single task in the chain of tasks to be executed by the client.
concrete template	One component in the template method design pattern, the concrete templates are collections of derived classes which implement the various steps of the algorithm being abstracted
closed	A level of the adaptability software quality level that can be identified when no allotments to enhancements through inheritance can be made
command injection	A security vulnerability involving an attacker gaining access to an interpreter where such access is against policy
commandeer driver	A type of manual driver where code is added to main that directly tests a function. Note that the rest of the program is typically not executed
commander	In a command passing scenario, a commander is the component that creates an order to be sent to the executor to be carried out
common	A level of coupling identified by Larry Constantine and later refined by Glenford Myers characterized by a group of modules that reference a global data structure
communicational	A level of cohesion identified by Larry Constantine where elements in a function are related by a reference to the same set of input and or output data
complete	A level of fidelity corresponding to the condition where states of the class completely match the design concern. Complete is also a level of alignment corresponding to when the problem domain and the inheritance tree match
complex	A level of coupling characterized by at least one piece of information being passed between functions that is nontrivial to create, validate, or interpret
component	A part of a distinct part of a system that can be developed, substituted, and even executed independently of the rest of the system
component architecture design	A part of a system design focusing on the design of a single component in a system. Component architecture designs are frequently expressed in terms of component diagrams or DFDs
component diagram	A graphical tool used to represent the components of a system and the protocols they use to communicate with each other. It is particularly useful for large-scale system visualization
composite state	A collection of related states in a state diagram
composition	When the member variable of one class is an object of another class
composition siblings	When two classes are unrelated to each other except that another class has two member variables: objects of the two classes
concrete strategy	One component in the strategy design pattern, the concrete strategy is a single manifestation or variation of an algorithm that is being abstracted

conditional breakpoint	A debugger feature allowing the programmer to specify the location and condition on which execution is to be paused
conductor	A component of an orchestrated peer-to-peer design, a conductor is a party that facilitates peer connections
configurable	A level of the malleability metric of software quality, configurable means that big changes can be made in the system without altering any code
constructor	A method that is guaranteed to be called when an object is instantiated
constant	A level of algorithmic efficiency where performance is unrelated to the amount of input into the algorithm, characterized by $O(1)$
context	One component in the strategy design pattern, the context class, also known as the client, is the part of the program that uses the algorithm being abstracted
continuous	A branch of mathematics (along with discrete) in which values may be subdivided. Real numbers and floating point values are continuous constructs
contract acceptance test	A form of acceptance testing focusing on whether a software system meets the stipulations outlined in a service level agreement
control	A level of coupling identified by Larry Constantine and later refined by Glenford Myers characterized by one module explicitly controlling the logic of another
convenience	A software quality metric describing how easy it is for a client to use a class in an application
conversion constructor	Any constructor that is not of the default, copy, or move variety. Often called a non-default constructor
convoluted	A level of adaptability characterized by adding functionality to a class requires making minor alterations to the base class
cookie	A token generated by a server used to maintain state on a web client
copy constructor	A method called on object creation when the client input is of the same type as the object
correctness	A software quality metric describing the extent in which a given input produces the expected output
counter	A type of variable that increments. If, for example, one were to write a program to display the numbers from 1 to 10, then a counter would be used to represent these values. Usually, counters are integers
counter-controlled loop	A type of loop where code is repeated a fixed number of times. Usually the controlling variable is an integer
coupling	A software quality metric describing the complexity of the interface between units of software
creational design patterns	A collection of design patterns facilitating the creation of objects. These include the factory, the abstract factory, the builder, the prototype, and the singleton
credentials	A collection of information used during the authentication process to establish an individual is who they say they are
critical	A level of redundancy where siblings in an inheritance tree have common attributes or operations
customer	An individual who sponsors or pays for a software package
data	A level of coupling identified by Larry Constantine and later refined by Glenford Myers characterized by modules directly communicating with each other where all interface data items are homogeneous data items
data-driven	A level of the malleability metric of software quality, data-driven means that significant changes to the algorithm can be made by altering only data elements, not logical elements
data flow diagram	A graphical tool used to represent how data flows through a program. Though it is particularly useful at the function level, a data flow diagram (DFD) can also be useful for representing system designs or even the flow of data within a single function
data protection	A collection of techniques used to provide robustness assurances to class clients
data storage	One of the three things that classes tend to do. The other two are to contain business logic and provide public interfaces
data type binding table	A binding table that maps the data type name with its memory size requirements and all other information required by the compiler to use the data type

debugger	A tool that runs a program so the developer can gain insight how the program is functioning
decision tree	A graphical tool used to represent the process of making complex decision
decorator	An algorithm abstraction design pattern allowing the client to add tasks to an algorithm dynamically to they can be applied in any order or combination
deducible	A low level of the understandability software quality metric, code that is deducible is code where all the required information is present. It is not straightforward where everything is clearly stated, but is better than misleading
deep copy	When an object is copied, a deep copy duplicates every member variable. This is important when an object contains a reference. Most clients expect a copy to duplicate the member variables rather than have both objects share a single instance of the member variables
default constructor	A method called on object creation when no parameters are provided
delegate	A function passed to another function as a parameter
delegate invocation	A command passing design pattern where an order is represented by a function passed as a parameter to an executor
dependency	A type of class relation that is not composition, aggregation, nested, or association. Usually, dependency means that a parameter is accepted to a method that is of that data type
dequeue	The process of removing an element from the front of a queue
dereference	The process of retrieving a value from an address. One can dereference a value from a pointer with the dereference operator or one can dereference a value from an index in an array with the square-bracket [] operator
dereference operator	A programming language construct allowing the programmer to request the data corresponding to a pointer
derived class	A class that inherits of another base class
design	One software engineering activity (along with requirements elicitation, development, testing, and others) involving figuring out how the software is to be built. The end result of design activities is typically design representations such as flowcharts, pseudocode, structure charts, DFDs, class diagrams, component diagrams, and others
design concern	An aspect, constraint, requirement, or need that a software system is meant to fulfill
design document	A mixture between a graphical and a textual tool, a design document can be used to completely represent a system design, from the highest possible level down to individual statements. Design documents are aggregate tools, meaning they are the fusion of several visualization tools
design phase	The part of the software development life cycle where the system design and many of the details are specified. The design phase could last a couple minutes or a couple years, depending on the scope of the project and the development methodology utilized by the team
design pattern	A generalized solution to a common problem encountered in programming. Design patterns exist at the algorithm level (such as an iterator), all the way up to the system level (such as the model-view-controller)
developer	One who writes computer code, who consistently makes high-quality software design decisions. Most consider a developer as being above a programmer but somewhat below a software architect
development	This term has two distinct definitions. The first refers to the process of creating a software system or a feature on an existing system. In this sense, “development” is a synonym of “coding”. The second use of the term refers to one of four viewpoints. The development viewpoint refers to looking at the partitions of a system and how the various components are organized
dictionary	An aggregate data type consisting of an ordered collection of homogeneous pairs facilitating fast key-value insertion, removal, and retrieval
direct invocation	A command passing design pattern where an order is represented by a function call to the executor
direct recursion	A desirable form of recursion where a function calls itself. Most recursive designs are direct. The alternative to direct recursion is indirect recursion

director	A component of the builder design pattern, a director determines which options the client desires while constructing an object
discrete	A branch of mathematics (along with continuous) in which values cannot be subdivided. Integers and Boolean values are discrete constructs
display	A name used for a class method which sends data to the user
disassembly	A debugger tool allowing the programmer to view the compiled code that it sent to the CPU
disinheritance	An inheritance type where a method in a base class is not implemented in a derived class
disjoint classes	When two classes are unrelated. They are not related by composition or by inheritance
distinct	A level of redundancy where there are no duplicated attributes or operations in an inheritance tree
distributed system	A type of peer-to-peer architecture where multiple nodes participate in working on a single problem
document	A level of coupling characterized by at least one piece of information being passed between functions that contains a rich language including syntactic and/or semantic rules
domain analysis	A development methodology conducted by the developer to better understand the domain model
double	A unit test tool used to isolate a class for the purpose of unit testing. When a class under test has external dependencies, a double can be used to stand-in for the dependency and thereby isolate the class under test. There are five types of doubles: dummy, fake, stub, spy, and mock
driver	A simple function or program designed to test another function or program by generating input parameters and validating output parameters
driver function	A type of manual driver where permanent code is written to test a specific function. Driver functions usually start out as manual drivers and then get converted to automation later in the development process
dummy	A type of unit test double where an empty class is created containing the same interfaces as the class it is meant to replace. All the public methods contain asserts to verify they are not called
dynamic data type	The capacity provided in some programming languages to specify and even change the data type of a variable at runtime
early binding	A programming construct that is completely resolved at compile time. This means that the compiler has all the information necessary to completely define the construct. All programming constructs are either early binding, late binding, or a combination of the two
efficiency	There are two definitions for efficiency in software engineering. The first is a quality metric describing how algorithm execution time is related to input size. The second is one of three meta-metrics, how much time or effort is required to obtain a measurement
enabling	A level of adaptability characterized by much functionality that can be added through small changes
encapsulated	A level of coupling characterized by parameters being passed between functions which are in a convenient format and are guaranteed to be in a valid state
encapsulated invocation	A command passing design pattern where an order is represented by a class sent to an executor
encapsulation	The process of creating classes to capture component of the system design
end-condition	Part of a test case, the end-condition indicates the output from an algorithm
ENIAC	The first electric, general-purpose computer. ENIAC's name is an acronym: Electronic Numerical Integrator and Computer
enqueue	The process of adding an element onto a queue
enrollment	The process of a new token being generated for a previously unseen entity
entity	A component of the problem domain
entity relationship diagram	A design visualization tool used to represent databases
entity representation	The process of representing components of the problem domain in a computational construct
entity table	A table used to map thin tokens into their corresponding entity

entry point	A pseudo-state in a state diagram representing the entry point to a composite state
equivalence class	A set of input that has the same characteristic. Any member of an equivalence class should cause the system to behave in a similar fashion
event-controlled loop	A type of loop where code is repeated until an event occurs. Usually, that even is represented with a Boolean expression that evaluates to false
execution	The process of running or completing a program statement or a complete program
executor	In a command passing scenario, an executor is the component to receive an order from the commander and then carries it out
exit point	A pseudo-state in a state diagram representing the exit point from a composite state
external transition	A transition crossing a composite state boundary in a state diagram
extraneous	A level of fidelity corresponding to when every state of the design concern is represented in the class, and when some states in the class do not correspond to a design concern
expression	An equation that evaluations to a value
extension	A type of inheritance where one class assumes all the properties and methods of another class and adds a few
external	A level of coupling identified by Larry Constantine and later refined by Glenford Myers characterized by a group of modules are not content or common coupled and they reference a homogeneous global data item
extraneous	A level in the cohesion metric of modularization, characterized by a function where at least one part is not directed towards a single task. However, the principle task is completely represented. Extraneous is also a level in the alignment metric of class relation design, characterized by the existence of some classes in the inheritance tree which do not match the problem domain
façade	A programming technique and design pattern involving simplifying coupling by shielding two functions or subsystems from each other's complexities
factory	A design pattern designed to facilitate the creation of an object that is a member of a polymorphic set. A factory takes an object identifier as input (usually an integer or an enumeration) and returns a new instance of one of the derived classes
fake	A type of unit test double where a simplified version of a class is created that avoids a complex operation (such as file I/O)
fall through	An inheritance type where the derived class method relies on the implementation of the corresponding base class without modification
fallthrough	A mechanism in a switch-case statement allowing control to continue onto the next case label after the previous one is complete
fault tolerance	A software quality metric describing the degree in which the program can continue to function in the face of errors or unexpected input
fidelity	A software quality metric describing the suitability of a class in representing a design concern
filter	A function that produces a new collection from an old one by selecting only those elements that satisfy a given condition. Filter, along with map and reduce, are among the most commonly used functional programming tools
final state	A pseudo-state in a state diagram representing the last state the program is in
find	A name used for a class method which locates an element that is within a collection
fixture	A test fixture is a component of a unit test. It represents everything that needs to be in place for a unit test to run. A test fixture is created in the setup step of a test method
FOR	Pseudocode keyword signifying a counting loop
FORTRAN	The first practical compiled programming language. The name FORTRAN came from FORmula TRANslation, named for a chief design goal: to facilitate translating mathematical equations into executable code
fork	A special transition in a state diagram indicating that two or more states will be simultaneously occupied next

flare	A type of automatic program trace consisting of a simple display statement added to the code indicating that execution has reached a given line
flow arrows	Arrows in a flowchart represent how program flows from one symbol to another. They are also DFD elements representing the movement of data through a system
flow process chart	A graphical tool used to represent algorithms. A flow process chart is a variation of a flowchart where all the symbols are arranged vertically
flowchart	A graphical tool used to represent algorithms, flowcharts are particularly good for communicating designs with non-technical people and for visualizing complex decisions
fragile	A level of quality in the robustness encapsulation design metric capturing when a class was only tested in very restricted circumstances
front door test	Another name for a black box test. Front door tests are tests use only public interfaces and do not have access to implementation details
front matter	The beginning of a large document such as an SRS or SDD used to inform the reader of what is to be built, who sponsored the project, how to know if the project is successful, and any related work that may impact the project
function	A unit of a program containing a collection of statements
function binding table	A binding table mapping a function name with the function signature and the location where the function's compiled code resides in memory
function pointer	The capacity provided by some programming languages to specify a pointer to a function (as opposed to a pointer to data)
function signature	The function name coupled with the number of parameters, the data type of each parameter, and the return type
function under test	The code (usually a function) that a driver is exercising
functional	A level of cohesion identified by Larry Constantine where all the elements in a function are related to the performance of a single task. This is related to the strong level of cohesion used today
functional decomposition	A technique used to understand or debug a multi-function algorithm where the return result of a function is substituted for the function call itself
functional programming	One of three programming paradigms (the others being object-oriented and imperative) characterized by programs being created out of functions where function's behavior is solely dependent on its input, completely independent of system state
GET	Pseudocode keyword signifying receiving input from the user
get	A name used in a class method that retrieves data from an object
getter	A method used to retrieve an attribute associated with a class
global	The term “global” is an adjective referring to the degree of scope of a given programming construct. Global means that any entity in the system can have access to it. Often “global” is used to refer to a global variable, a variable residing in the global scope and therefore visible to the entire program
group	A collection of peers in a peer-to-peer design
guard assert	An assert in a debug function such as a test method. The guard assert replaces a conditional statement in a test with an assertion. This is useful if code will throw an exception or crash if a certain condition is not met. Rather than preventing the exception or crash with an IF statement, an assert is used instead. This ensures the guard assert will fire before the code is executed containing the potential for an exception or crash
hacker	One who writes computer code, typically in an undisciplined or haphazard fashion. Most consider a hacker as being somewhat below a programmer
has	A prefix used with a method name indicating the method is a predicate. Has indicates the object contains a property or that a collection contains an element
has-a	A type of class relation associated with composition or aggregation

head recursion	One of two recursion strategies where the recursive call stack is first built and then the recursive work is done when functions are popped off of the call stack. The other recursive strategy is called tail recursion
high-order function	A function that takes another function as a parameter
homogeneity	A state or condition when multiple components fulfill the same role or perform the same action
IF	Pseudocode keyword signifying a binary decision is being made
Immediate window	A debugger tool allowing the programmer to enter the name of a variable or enter an expression for the purpose of viewing the value at a given moment in the program's execution. The immediate window is also known as "immediate"
immutability	A property of a variable where once a value is assigned, it cannot be changed
imperative	One of three programming paradigms (the others being object-oriented and functional) characterized by programs being created out of ordered statements that control flow and compute values
increment	The process of increasing the value of a variable. The most common form of incrementation is to increase the value of a counter by one. This is so common that many languages have a special constructor for this: <code>++</code>
incremental counter	An entity representation strategy where a token is an enumeration, counter, or index
index	A variable used to indicate which record from a collection is to be utilized. Indices are often integers with the value 0 corresponding to the first element in an array
indirect recursion	An undesirable form of recursion where a function calls another function which, in turn, calls the first function. Few recursive designs are indirect. The alternative to indirect recursion is direct recursion
infinite loop	The process of a loop continuing indefinitely. This usually happens when the controlling expression always evaluates to true
infix	A function notation where the operator is between the operands. An example of infix notation is "four plus five"
inheritance	When one class is defined in terms of another class
inheritance siblings	Two classes that share the same base class
initial state	A pseudo-state in a state diagram indicating where the program begins
input	A name used for a class method that receives input from the user
insert	A name used for a class method that adds a new element into a collection
instantiation	The process of creating an object
instruction pointer	A variable indicating which instruction or command is currently being executed in a program
interface	A method of communication between components
instrumentation	The process of adding counters into an existing algorithm to measure how the algorithm performs
integration testing	A form of testing involving verification of the interplay between the various subsystems in a program
integration test plan	A part of a test plan used to describe how to verify that the units within a component work together
intent-revealing names	An understandability principle revolving around how to make code more obvious. The central theme is to make the name of a variable, function, or class reflect what it is meant to represent
interact	A name used for a class method that includes multiple input and output interactions
interactive	A level of coupling characterized by a communication avenue between units of software involving dialogs, sessions, or interactions
interactor	A DFD element representing an entity that creates and consumes data. These can be users or external systems
interface	The connection between two components. If the components are small, it could be a return parameter from a function. If the components are large and distributed, it could be a HTTPS connection between a client and a server

interface expansion	The process of adding member variables and/or methods to a base class that were previously unutilized by the other derived classes
infrastructure-as-a-service	A cloud model where a customer rents the physical computers used to host servers
interoperability	A software quality metric describing the ability of the system to collaborate with other systems
interpreter	A command passing design pattern where an order is represented as a rich language to be interpreted by an executor
invariant	an attribute or operation that is shared between classes in an inheritance tree
invoker	The invoker is the executor in the encapsulated invocation design pattern
is	A prefix used with a function or method name indicating the function or method is a predicate
is-a	A type of class relation. Is-a is synonymous with inheritance
iterator	Generic looping structures that work the same regardless of the underlying data structure
join	A special transition in a state diagram indicating that several states in a concurrent state machine transition to a single state
JSON	A lightweight data interchange format developed for the JavaScript programming language (hence the name "JavaScript Object Notation") but now commonly used in a wide variety of message passing scenarios
jump table	A mechanism used in implementing SWITCH/CASE statements where the program stores the location of code corresponding to CASE labels in an array. The program then jumps to the corresponding code in constant time
key	A token selection strategy where the token is represented as a value which is used to retrieve the entity in an entity table
lambda	A lambda is an anonymous function: a collection of statements designed to perform a task. The only difference between a lambda and a traditional function is that lambdas are not given a name
lambda calculus	A branch of mathematics based on functional abstraction. Lambda calculus is the basis of functional programming
late binding	A programming construct that is completely resolved at runtime. This means that the compiler is lacking some information necessary to completely define the construct; other information is provided as the program runs. All programming constructs are either early binding, late binding, or a combination of the two
linear	A level of algorithmic efficiency where performance is directly related to the amount of input into the algorithm, characterized by $O(n)$
linear aggregation integration	A separation of concerns reassembly strategy where the interface contains a reference of a business logic class which, in turn, contains a reference of a storage class
linear composition integration	A separation of concerns reassembly strategy where the interface contains an instance of a business logic class which, in turn, contains an instance of a storage class
linked list	An aggregate data type consisting of a collection of nodes connected by pointers. Each node in the sequence points to the next node
Liskov substitution principle	An OO design principles stating that any of the children of a class base class can stand in for the base class
list	An aggregate data type consisting of a collection of homogeneous elements allowing for rapid insertion and removal but providing sequential access. Lists are often implemented with a linked list data structure
local	The term "local" is an adjective referring to the degree of scope of a given programming construct. Local means that any entity in a given function can have access to it, but no entity outside the function. Often "local" is used to refer to a local variable, a variable residing in a given function
local transition	A transition between substates within a composite state in a state diagram
logarithmic	A level of algorithmic efficiency where performance is a log of the amount of input into the algorithm, characterized by $O(\log n)$
logging	The process of recording all that took place on a system during a period of time

logical	This term has three distinct definitions. The first refers to the decision-making process of a program. This is usually represented with Boolean expressions and conditionals. The second use of the term refers to one of four viewpoints. The logical viewpoint refers to looking at how data are stored in a system. The third is a level of cohesion identified by Larry Constantine where elements fall into some general category because they all do the same kind of thing
loop	A programming mechanism allowing for the repeated execution of one or more statements
loop unrolling	The process of replacing a loop with inline code. If a loop were to execute the body four times, then loop unrolling would consist of repeating the code in the body four times and removing the loop mechanism
machine code	A statement consisting of 1s and 0s that a computer's CPU consumes directly. Generally, not human readable
magic numbers	Literals embedded in algorithms with unexplained meaning. They are particularly difficult to manage when there are multiple occurrences of a single magic number in the codebase or when they are distributed throughout the codebase. These are also called hard-coded values
maintainability	A software quality metric describing how much time or effort is required to fix defects or make enhancements in a given piece of software
manual drivers	A type of driver that relies on a user to generate test cases and check output for correctness
map	A function that produces a new collection from an old one based on a passed criterion. Map, along with filter and reduce, are among the most commonly used functional programming tools
marker	A token in a state persistence scenario representing a configuration of the system
mediator	A message passing design pattern characterized by all communication between many program entities passing through a central hub
member function	A function associated with a class, also known as an operation or method
member variable	The data of a class, structure, or a tuple. If, for example, a tuple consists of three values, each value is known as a member variable. Another name for a member variable is an attribute
memory dump	A type of automatic program trace consisting of display code that puts the values of variables on the screen so they can be visually verified by the programmer
memory watch	A debugger tool enabling the programmer to view the contents of memory at a given moment of program execution
meta-metric	A metric from which we measure metrics. They allow us to determine the quality of the metrics we use to measure the quality of our software designs
metaphor	A symbol used to represent an aspect of a program
method	A function associated with a class, also known as a member function or operation
minor	A level of redundancy where any duplication in an inheritance tree exists in non-adjacent classes
mirrored client	A client-server architecture where the client maintains a cache of the server's data
misleading	A low level of the understandability software quality metric, code that is misleading is code that suggests an invalid mental model to the programmer. It does something different than it seems to do
mock	A type of unit test double which is a sophisticated stub that replays fixed answers from expected inputs
modularization	The process of subdividing a program into functions for the purpose of reusing code, simplifying design, and reducing development costs
monolithic architecture	A system built from a single component or layer
move constructor	A form of copy constructor that modifies the client's input by stealing its member variables
multiple recursion	A form of recursion which should generally be avoided. It occurs when there are more than one recursive variable or more than one recursive function calls. This can lead to very inefficient algorithms if not managed carefully
multiplicity	A class diagram indication of the number of instances of one class are contained within another. Multiplicities are commonly used with composition and aggregation relations

mystery guest	A hidden collaborator in a unit test that can make a test more difficult to understand or maintain
n-log-n	A level of algorithmic efficiency where performance is linear times logarithmic, characterized by $O(n \log n)$
n-squared	A level of algorithmic efficiency where performance is a square of the amount of input into the algorithm, characterized by $O(n^2)$
namespace	A program entity similar to a block where collections of related entities are defined. For example, one might wish to indicate that a collection of classes are related by encapsulating them in a namespace
nested class	A class defined within another class
no direct	A level of coupling identified by Larry Constantine and later refined by Glenford Myers characterized by one function calling another but passing no parameters and accepting no return data. This corresponds to the modern definition of trivial coupling
notification breakpoint	A debugger tool allowing the programmer to specify the condition on which execution is to be paused. This is distinct from a conditional breakpoint in that the programmer does not need to know the location in the code where the condition is to be met
NULL	A special address corresponding to the location <code>0x00000000</code> . The NULL address usually means “end,” “uninitialized,” or perhaps “error”
object	An instantiated class. This is a variable whose data type is a class
object reference	A thick token selection strategy where the entity is encoded in an object
observer	A message passing design pattern characterized by many program entities subscribing to an event and, when the event occurs, each receiving a notification
obvious	The highest level of the understandability quality metric, code that is obvious leaves no room for confusion or interpretation
object-oriented	One of three programming paradigms (the others being imperative and functional) characterized by programs being created out of classes
on	A prefix used for a function name signifying that the method is a callback that is to be executed at a pre-specified event
open-closed principle	An OO design principle stating that software entities should be open for extension but closed for modification
operand	An operand is a value or expression used in a function or mathematical expression. In the expression “4 + 5”, both 4 and 5 are operands
operating system	A computer program that provides an interface between the hardware and applications. Operating systems provide a variety of functions, including memory management, interfacing with devices, scheduling processes and tasks, networking with other computers, and interfacing with system users
operation	A function associated with a class, also known as a member function or method
operational acceptance testing	A type of acceptance testing that verifies if the software system can function on the hardware and software platform on which it is designed to run
operator	An operator is a symbol that stands for a function or operation. In the expression “4 + 5”, the plus sign is an operator
operator overloading	The process of using a more convenient and human-readable notation for calling a function
orchestrated	A type of peer-to-peer design where communication between peers is mediated by a third party called a conductor
order	In a command passing scenario, an order is the task sent by the commander to be accomplished by the executor
originator	The component that generates a token from an entity
orthogonal region	Two regions in a graph between which there are no edges. In a state diagram, states in one orthogonal region cannot be reached by those in another region
OSI reference model	A network model with seven tiers defined by Hubert Zimmerman in 1980 and utilized by virtually all networks to this day

outsider	A role in the authentication process representing an individual who is not a rightful user of a system
over specified test	A problem with white box tests which depend too much on implementation details of the unit under test. This results in test code being changed when implementation details of production code is changed
overengineer	The process of building too much functionality, flexibility, or robustness into a system
package	A program entity similar to a namespace where collections of related entities are stored. Many programming languages allow components to be defined by packages, allowing programmers to conveniently include them in his or her application
parameter	The data sent into a function from the caller. This is also known as the function's input
partial	There are two meanings. A level in the cohesion metric of modularization characterized by a function where all aspects are directed towards a single task, but the task is not completely represented by the function. Additionally, a level of fidelity corresponding to when some states of the design concern are missing but there are no extraneous states. Finally, partial is a level of alignment corresponding to the existence of items in the problem domain which are not present in the inheritance tree
peer	One element in a peer-to-peer group
peer-to-peer	A system design where multiple components perform the same role
physical	One of the four model viewpoints, the physical perspective of a system is a description of the hardware
pipelining	A CPU optimization scheme where multiple instructions are executed simultaneously. This is also called instruction-level parallelism
platform-as-a-service	A cloud model where the customer rents the physical computers used to host servers as well as the components of the server. This can include databases and related tools.
point of sale	Point of sale (POS) systems are a suite of applications facilitating a store to manage their inventory, facilitate purchasing, and generate reports
pointer	A variable containing the address of an element rather than the value of an element
polymorphic set	A polymorphic set is the collection of classes that derive from a single base class
polymorphism	An inheritance technique where various derived classes have different implementations but the same interfaces
poor	A level of fidelity, corresponding to when the states of a class poorly represents the design concerns. This is a combination of extraneous and partial fidelity. Poor is also a level of alignment, corresponding to the condition when the inheritance tree poorly represents the design concern
port	An explicit and often formal interface through which all communication travels
pre-processor directive	A programming construct allowing the programmer to specify operations that are to be performed on the source code before it is compiled. The most common pre-processor directives are macros and includes
precision questioning	The process of asking questions that are increasingly specific to get to the heart of an issue
predicate	A type of function that returns a Boolean value
prefix	A function notation where the operator exists before the operands. An example of prefix notation is "add four and five"
primary key	A token used in a database used to uniquely address a given record
private	An access modifier indicating that only member functions can have access to the corresponding member variable or method
procedural	This term has two distinct definitions. The first refers to the style of programming where functions and algorithms are the primary units of design. This is in contrast to object-oriented programming and functional programming. The second definition refers to a level of cohesion identified by Larry Constantine where each element in a function are organized by a sequential relationship that must be honored
process	This term has two distinct definitions. The first refers to one concurrent execution path of a program. A reasonably complex program may have several processes doing different tasks concurrently. The second use of

	the term refers to one of four viewpoints. The process viewpoint refers to looking at the system from the perspective of the verbs or algorithms
processor	A DFD symbol representing program entities that act on data. They can transform data, set data, or move data between elements
program flow	The various paths that an instruction pointer can follow when executing a program
programmer	One who writes computer code. Most consider a programmer as being above a hacker but somewhat below a software developer or a software architect
programmer-specified watch	A debugger feature enabling the programmer to specify the variables to monitor during execution of the program
programming language	A collection of keywords, symbols, and grammatical rules used to instruct a computer to perform a task. Programming languages can be executed by a computer, as opposed to human-languages which typically cannot
progress step	One part of two in every recursive algorithm, the progress step is the process of taking the larger problem and breaking it into a smaller pieces, each of which are closer to the end-condition
prohibitive	A level of the malleability metric of software quality, prohibitive means that it is easier to start over than to update code. The maintainability is so low that it economically cannot be refactored. Prohibitive is a level of the adaptability metric, characterized by the amount of work required to add functionality being greater than the benefit received
PROMPT	Pseudocode keyword signifying the user is asked for information. This is a combination of a PUT and a GET
prompt	A name used for a class method which both displays a request for information to the user and receives the input from the user. A prompt method is the combination of a display and an input
protected	An access modifier indicating that only member functions and member functions of derived classes can have access to the corresponding member variable or method
protection proxy	A type of proxy that is a mechanism to keep unauthorized clients from accessing the real object
protocol	The language utilized in an interface
prototype	A creational design pattern used to facilitate creating duplicate copies of an existing object
proven	A level of quality in the robustness encapsulation design metric capturing the case when a design was formally proven to be robust
provided interface	An interface between components. A provided interface is a service offered to another component
provider	A function, class, component, or system that provides services for a client
proxy	An interface design pattern where one object stands in for a real or genuine object
pseudocode	A textual tool used to represent algorithms. Pseudocode can be very close to a programming language. This makes it easy to translate a design into working code
public	An access modifier indicating that the client can have access to the corresponding attribute or operation
public interfaces	The collection of interfaces in a class that are available for a client to use
pure function	A function whose operation is completely independent of any system or program state. Pure functions, along with immutability, represent the two constraints of functional programming
PUT	Pseudocode keyword signifying output is sent to the user
puzzling	The lowest level the of understandability metric, code that is puzzling requires substantial effort on the part of the programmer to figure it out
quality	A desirable property of a software system. One can say that a system has achieved the desired level of quality when it is fit for the client's use
quality assurance engineer	Also known as a QA engineer, a quality assurance engineer is a software engineer responsible for assessing the quality of various aspects of the software project and software process

quality assurance phase	The part of the software development life cycle where the level of quality of the product is ascertained and where bugs are identified. This is often done in conjunction with other development activities
queue	A data structure analogous to a line where the first one to enter is the first one to leave
queuing	The process of a recipient accepting input that may arrive unpredictably and then handling it in the order it came at a later time
reachability	In a graph, reachability is the condition of one node being reachable from another. Reachability is also an important property of a state diagram, describing whether one state can be reached from another
READ	Pseudocode keyword signifying receiving input from a file, a server, or another program
real	In the proxy design pattern, a real object is the provider that the client may need to invoke
redo	The process of repeating an action one or more times
reduce	A function that produces a singular value that is computed by combining all the values in the input collection. Reduce, along with filter and map, are among the most commonly used functional programming tools
redundancy	A software quality metric describing the extent in which there are common elements in an inheritance tree
redundant	A level of redundancy where there are multiple instances of duplicated attributes or operations in an inheritance tree
refactor	The process of rewriting and restructuring code so it functions the same as before. The intent is that the new code should improve non-functional attributes such as maintainability, performance, or other metrics
refactorable	A level of the malleability metric of software quality, refactorable means that refactoring and redesigns are required for most maintenance tasks
referential transparency	A property of a function where every aspect of a function's behavior is solely dependent on its input, completely independent of system state. Referential transparency is a central concept to functional programming
regulation acceptance test	A type of acceptance testing focusing on whether relevant rules or regulations are met
reimplementation	The process of altering the logic of a function, method, or class without changing the functionality or interface
reliability	There are two definitions for reliability in the software engineering context. The first refers the capacity of a system to produce output to the required level of precision. The second is one of three meta-metrics, the amount of error in a measurement
remote method invocation	Analogous to a remote procedure call, remote method invocation is a message passing technique where two processes can communicate across disjoint address spaces through invoking a method in a class
remote procedure call	A remote procedure call, also known as RPC, is a message passing technique where two process can communicate across disjoint address spaces through invoking a function
remote proxy	A proxy that stands in for an object that exists in a remote location
remove	A name used for a class method that removes an existing item from a collection
required interface	An interface between components. The required interface is an interface needed by the component for it to function properly
requirement	According to the IEEE, a requirement is “a condition or capacity that must be met or possessed by a system to meet an objective”
requirements elicitation	The process of gathering data from a variety of courses for the purpose of discovering what is needed for a given software system to fulfill the needs of the client
resilient	A level of quality in the robustness encapsulation design metric capturing the case that software was thoroughly white box tested
robustness	A software quality metric describing the degree of resistance a class has from being placed in an invalid state or sending invalid data to the client
routing	The process of sending a message or command form the originator to the intended audience. Usually, routing involves the message or command traveling through multiple entities until the intended audience is found

run	A debugger stepping tool that executes the program without pausing or stepping from the current location
run to	A debugger stepping tool that continues execution of the program until a specified statement is reached. Execution is paused at that point
scenario	A simple story describing how the system is used
scope	A programming construct referring to the visibility of a variable or data type. A variable that is defined in a function is only visible or accessible within the function. In this case, the scope of the variable would be the bounds of the function
scoreboard	A type of manual program trace consisting of creating a collection of slots, each of which representing the location in memory where various variables hold their data. As program execution progresses, the programmer updates the slots with the current values
script injection	A security vulnerability involving an attacker utilizing a provided interpreter in a way that is against policy
security	A software quality metric describing the capacity of the system to provide confidentiality, integrity, and availability assurances
sentinel	A guardian or a gatekeeper
sentinel-controlled loop	A type of loop where code is repeated until a gatekeeper called a sentinel indicates the loop is finished. Sentinel-controlled loops are characterized by multiple decision points, each of which could change a Boolean variable. This Boolean variable is the loop control variable indicating when the loop terminates
sequence diagram	A design visualization tool used to represent concurrency challenges
sequential	A level of cohesion identified by Larry Constantine where the output data from one processing element serves as the input data for the next processing element
server	One component of a client-server system, the server is the system that responds to client requests. Common services offered by a server include storing data, processing data, and facilitating communication with other network entities
service level agreement	A contract describing software to be built in exchange for a given compensation
set	A name used on a method that updates an attribute in a class
set next statement	A debugger stepping feature that modifies the regular execution of the program so the specified statement is the next one to be executed
setter	A method used to update a class' attribute(s)
shallow copy	When an object is copied, a shallow copy does not duplicate every member variable. This is bad, especially when an object contains a reference. Most clients expect a copy to duplicate the member variables rather than have both objects share a single instance of the member variables
shortest path first	The shortest path first (SPF) algorithm is a depth-first search through a graph to find the shortest path between two nodes
side effect	A secondary outcome of a program. A medication that has a side effect is one that has an effect on the body different than that which the medicine was meant to accomplish. Side effects are considered undesirable in software; well-designed code should lack side effects
signature	A function signature is the interface between a function and the rest of the program. It consists of the name of the function, its parameters, and the return value(s). For any part of the program to call a given function, it needs to honor the function signature by calling it by the appropriate name and providing the expected parameters
simple	A level of coupling characterized by the information exchange between functions being in a format that is easy to select, interpret, and validate
single responsibility principle	A class design guideline comparable to fidelity. It states “gather together the things that change for the same reason. Separate those things that change for different reasons”
singleton	A creational design pattern used to ensure that only one instance of a class exists in an application
size	A name used for a class method which returns the number of elements in a collection

software	A program or a suite of programs designed to accomplish a task. Software is a logical entity consisting only of information whereas hardware is a physical entity that can be touched
software-as-a-service	A cloud computing model where the customer rents client applications as well as servers
software design description	Commonly known as an SDD, a software design description is a formal design document used to represent the entirety of a system design. The heart of a SDD is a collection of design views, each of which represents one aspect of the overall system design
software development engineer in test	A type of software engineer that specializes in creating test cases, implementing white box tests, and authoring testing automation
software development life cycle	The process a development team follows for a software project to be completed
software requirements specification	Commonly known as an SRS, a software requirement specification is a formal description of all the client expects and needs from a system
SOLID	A collection of OO principles defined by Robert Martin in 2003. The SOLID principles include single-responsibility, open-closed, Liskov substitution, interface segregation, and dependency inversion
source code	Instructions written in a programming language. Source code is often called "code"
spy	A type of unit test double consisting of a stub that also collects information about how it is used
stack	Two definitions. A stack is an abstract data structure characterized by last-in, first-out. Additionally, a stack is a collection of components in an <i>n</i> -tier architecture
stakeholder	An individual who has a vested interest in a project. This could be the user (an individual who uses the application), the customer (an individual who pays for the application), the development team, or those who support the software after it has been released
stamp	A level of coupling identified by Larry Constantine and later refined by Glenford Myers characterized by two functions referring to the same non-global data structure
standard operating procedure	A set of procedures, standards, instructions, or policies identified by an organization which serves to describe how things are to be done. Most companies have documented standard operating procedures (SOP) describing how code is written, how tests are performed, how code is integrated into the project, and how products are released to the customer. All employees should know and follow the SOPs of their organization
start-condition	Part of a test case, the start-condition indicates the input into an algorithm
state	A discrete configuration of a system which describes its behavior
state diagram	A graphical tool used to represent state machines. A state diagram consists of states and transitions
state machine	An algorithm defined by states and their transitions. Also known as a finite-state machine
state space	The collection of all possible states and their transitions in a state machine
state transition	The process of a state machine moving from one state to another
statement	A high-level programming language construct consisting of one or more assembly code elements
static	A variable that can have only one copy within a given instance of the program. Thus, all objects (for member variable statics) or all functions (for local variable statics) share the same instance of the variable
step	A debugger feature allowing the programmer to control execution of the program in discrete increments. A step is also one component in the template method design pattern. This is one aspect of an abstract algorithm that is to be defined in a derived class
step into	A debugger stepping feature allowing the programmer to execute the next statement. If that statement is in a function call, then the debugger will step into that function and pause at the first line therein
step out	A debugger stepping feature that will run to the end of the current function and pause at the point in the caller where the current function is called
step over	A debugger stepping feature allowing the programmer to execute the next statement in the current function. If the next statement is a function call, then the function (and its children) will be executed
stereotype	A label on many UML diagrams. For example, a component diagram may have the stereotype «subsystem» indicating the type of component represented therein

storage	A DFD symbol representing data at rest. This can be a variable, a file, a database, or a data structure
straightforward	The second best level of the understandability quality metric, code that is straightforward is code lacking subtlety: everything is clearly stated. It is also the second-best level of adaptability, when changes can be made solely by adding child classes
strategy	An algorithm abstraction design pattern where there are multiple variations of an algorithm from which the client can choose
strong	To definitions. A level in the cohesion metric of modularization characterized by a function where all aspects are directed to performing a single task and that task is completely represented. Also, a level of quality in the robustness encapsulation design metric capturing when software was rigorously and systematically black box tested
structure	An aggregate data type consisting of a collection of named variables
structure chart	A graphical tool used to represent modularization decisions. A structure chart represents functions and how they call each other
stub	A type of unit test double consisting of a simplified version of the class it is meant to replace that always provide a fixed output
subscriber	A role in the authentication process representing an individual who is permitted to have access to system resources
subject-matter expert	A member of the design team who is an authority or expert in the area in which the product is built. If a team was tasked with making an accounting program, then an accountant would serve as the subject-matter expert
substate	A state within a composite state in a state diagram
superfluous	A level of coupling characterized by at least one piece of data or information being passed between units of software that are unnecessary
supportability	A software quality metric describing the degree of effort required to keep the system working
SWITCH/CASE	Pseudocode keyword signifying a multi-way decision is being made
swimlane flowchart	A graphical tool used to represent algorithms. A swim lane flowchart is a variation of a flowchart where vertical columns are used to represent structural borders
system	A collection of software deliverables that meet the client's needs
system design	An outline of the overall design of a software system. A system design typically describes the various components of a system and how they interact
system test	A form of black box testing that encompasses the entire system. Here, the tester or quality assurance engineer looks at the system as a whole, often from the user's perspective
system test plan	The plan to perform system tests on a system
tail recursion	One of two recursion strategies where the recursive work is done as functions are pushed onto the stack and then the return value is passed back to the caller as functions are popped off the call stack. The other recursive strategy is called head recursion
technical debt	The accumulation of maintainability problems in a codebase which makes it progressively more difficult to work with. Technical debt amasses over time, seldom the result of a single act
template	A function or class that can operate on a wide variety of data types, many of which do not have to be known by the developer of the function or class
template method	An algorithm abstraction design pattern where common steps in an algorithm are defined, and where subclasses are allowed to decide what the common steps mean
temporal	A level of cohesion identified by Larry Constantine where some logical relationship exists in the elements of a module and they are related by time. That is, the temporally bound elements are executed in the same time
terminator	The start or end symbol in a flowchart representing where program flow begins and where it ends
test case	A set of actions used to verify a product, feature, or module behaves as expected. It consists of a start and end condition which informs the programmer of the correctness of an algorithm

test-driven development	A development methodology where unit tests are written and executed before the corresponding code, enabling the developer to both verify that the tests work and that the accompanying code also works
test enumeration	A strategy of implementing a testcase class's run method or a test runner function where each individual test is enumerated explicitly. Another possible strategy is test method discovery
test method	A single function in a unit test case class which represents a single test case. Test methods have four components: setup, exercise, verify, and teardown
test method discovery	A strategy for implementing a testcase class's run method or a test runner function that utilizes the language's reflection feature. This allows the runner to automatically find all the individual tests to be run. Another possible strategy is test enumeration
test plan	A document describing the scope, approach, resources, and schedule of testing activities
test runner	A debug function that exercises all the automation or unit tests in a project
test suite	A type of automation driver where a collection of automation functions is executed with the purpose of obtaining a large amount of code coverage
testcase class	A debug class containing all the test methods needed to validate a single class under test. Every testcase class must inherit off the testcase base class. The testcase class is one component of the xUnit framework
tested	A level of quality in the robustness encapsulation design metric capturing when ad hoc black box testing was performed on a class
testing	One software engineering activity (along with requirements elicitation, design, development, and others) involving finding defects and fixing bugs
thick client	A client-server architecture where most of the computational work and data storage occur on the client
thin client	A client-server architecture where most of the computational work and data storage occurs on the server
thick token	A token selection strategy where the token contains all the information necessary to produce an entity, without the need of an entity table
thin token	A token selection strategy where the token is an index or key into an entity table used to map the token to the entity
ticket	A token used in a queue management system
timing diagram	A design visualization tool used to coordinate asynchronous events
token	A representation of an entity in a computer system
trace	The process of systematically stepping through the code, one statement at a time, to track the values of the variables
trace table	Part of a program trace, a trace table is a table representing the values of variables at various stages of execution
transition	The component on a state machine governing how the system moves from one state to another
transition arrow	The part of a transition indicating the source and destination states in a state diagram
transition condition	The part of a transition indicating the event triggering a state transition in a state diagram
trivial	A level of coupling characterized by no information being exchanged between functions
trust boundary	A virtual wall around a computation asset where elements within the wall have a higher degree of assurances that they are in a valid or safe state
truth table	A simple mathematical proof of a Boolean expression achieved by enumerating every possible input value and computing the resulting output value
tuple	An aggregate data type consisting of a collection of unnamed variables, each referenced by position
UML	Unified Modeling Language (UML) is a collection of tools used to model or visualize various parts of a program design
under engineer	The process of under-delivering on functionality, flexibility, or robustness

understandability	A component of maintainability which, in turn, is a software design metric. Understandability is a measure of how easy it is for a programmer to form a valid mental model of a segment of code
undo	The process of reversing the previously performed action
unit	A small subdivision of a software system that is distinct from the rest of the program. In most cases, a unit is a single function, method, or a class
unit test	White box developer-written automation used to validate a single function, method, or class in isolation of the rest of the program
unit test framework	Debug code written to facilitate integrating unit tests into a project. All unit test frameworks are modeled after JUnit, a framework developed for the Java language. The collection of all the unit test frameworks are collectively called xUnit
universalization	The process of making a single function, class, or type that contains all possible attributes
update	A name used for a class method that changes an element that exists in a collection
usability	A software quality metric describing the amount of effort required of users to learn, understand, prepare input, and interpret system output
use case	A design tool used to represent how external entities interface with the system. It is a written description of the system written from the client's perspective
user	An individual or a collection of individuals who use a software package
user acceptance test	A type of acceptance test focusing on whether the software system meets the needs of the intended users
username	A token used to represent a subscriber or user on a system
user story	A unit of software work akin to a use case or a requirement. User stories can be small (less than four hours of work), medium (about a day), large (several days), or an epic (a week or longer)
V-Model	A SDLC model characterized by the pairing of each development activity with a corresponding quality assurance or testing activity
v-table	Also known as a virtual method table, a v-table is a set of callbacks referring to the virtual functions associated with a given derived class. A v-table is a special type of a function binding table that is maintained at runtime
validity	One of three meta-metrics, validity is the degree in which our metric measures the right thing
variable	A named location in memory where data are stored
variable binding table	A binding table mapping variable names to their associated data type and the location in memory where they reside
variant	An attribute or operation that is unique to a given class in an inheritance tree
viewpoint	Perspectives in the software design process. Most reasonably complex applications require more than one viewpoint to adequately express the design. The types of viewpoints are: process (describing the verbs of the system), developmental (describing how the system is organized), logical (describing the nouns of the system), and physical (describing the physical components)
virtual	A method that can be overwritten by a derived class
virtual method	A method in a class that may be redefined in a derived class
virtual proxy	A type of proxy that stands in for a real object that is expensive to create
visitor	A message passing design pattern characterized by a reporting structure making little demands on the reporting elements
wait	A name used for a class method that pauses until a fixed amount of time has passed or a certain event has transpired
watch	A debugger tool giving the programmer the ability to monitor and adjust the state of variables during program execution
watch local variable	Also known simply as "watch," watch local variable is a debugger tool enabling the programmer the ability to see the values of the local variables at a given moment in program execution

weak	A level in the cohesion metric of modularization characterized by a function where at least one part is not directed towards performing a single task and that single task is not completely represented
WHILE	Pseudocode keyword signifying a non-FOR loop is used in the program
white box testing	White box testing is a validation and verification process utilizing detailed knowledge of how the system is implemented. Here, the testing process is highly coupled with the code being tested. If the developer chooses a different implementation, it is expected that the white box tests will fail and must be rewritten
wrapper	A programming technique where the complexities of an interface are shielded from the client through the user of function, class, or subsystem whose purpose is to translate a complicated interface into a simpler one. This is like the façade design pattern
WRITE	Pseudocode keyword signifying output is sent to a file, a server, or another program
xUnit	A name referring to the collection of unit test frameworks developed for most modern programming language. Each xUnit framework is modeled after the original JUnit framework which was developed for the Java language
YAGNI	A design principles standing for “You ain’t gonna need it.” It can be summed up with this: do not do any more work than is necessary

Further Reading

There are three types of references mentioned here: classics, references, and expounding sources. The classics are the “bookshelf” books that serious engineers should have on their bookshelf and have read at least once. Some of these classics are dated, but served as the genesis of our craft to this day. The reference sources are often handy to have on hand. Seldom does one need to read them, but they are often helpful for answering questions. The final type is the expounding sources. If you would like to go deeper in the topics that are presented here, these sources can take you to the next level.

Chapter 01 Tool: Flowchart

Brooks, F. (1975). *The Mythical Man-Month*, Addison-Wesley Professional.

Classic: This is a wonderful book that all aspiring software engineers should read. Though it is set in a day preceding many of the software advances with which we now take for granted, its teachings remain just as relevant. One particular chapter related to algorithm design is the concept of “essential” and “accidental” difficulties of writing code (Chapter 1: The Mythical Man-Month). This is a must read.

Object Management Group (2007). *Unified Modeling Language*.

Reference: It is often useful to go straight to the source. This is the definitive reference of the UML Activity Diagram.

Miles, R. (2008). *Learning UML 2.0: A Pragmatic Introduction to UML*. O'Reilly.

Reference: Though some of the examples can be a bit puzzling, this book does a good job of presenting activity diagrams, class diagrams, component diagrams, and many other viewpoints that you may need to use.

Chapter 02 Tool: Pseudocode

Robertson, L. (2006). *Simple Program Design*. Cengage Learning.

Expond: This book is easy to understand and provides many examples in pseudocode. While most describe pseudocode in an ad hoc way, Robertson applies a far more methodical and complete description. Pseudocode is introduced in Chapter 2 but is used throughout the book.

Chapter 03 Metric: Efficiency

Bachmann, P. (1894). *Analytische Zahlentheorie*.

Classic: The theoretical underpinnings of Big-O notation of algorithmic efficiency can be traced to Paul Bachmann, a German mathematician. Though this paper is in German, one can see the some of the constructs which are familiar today.

Graham, R., Knuth, D., and Patashnik, O. (1994). *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley.

Expond: For those who are thirsty for a more mathematical approach to Big-O, Ronald Graham and Donald Knuth wrote a surprisingly readable and entertaining book on the subject. At more than 600 pages, this is probably more detail than any software engineer needs.

Chapter 04 Metric: Maintainability

Lientz, Swanson, and Tompkins (1978), *Characteristics of Application Software Maintenance*. Communications of the ACM.

Classic: This study, dating back to 1978, became the foundation for the modern study of maintainability. It is so widely cited in the field that most refer to this paper and associated dataset as “the LST result.”

Chapter 05 Quality: Assert

Turing, A. (1949). *Checking a Large Routine. Report of a Conference on High Speed Automatic Calculating Machines.*

Classic: It is always interesting to go to the source. Alan Turing first conceived of the assert way back in 1949. This is worth a read, consisting of only a few pages.

Chapter 06 Quality: Trace

Fagon, M. (1976). *Design and Code Inspections to Reduce Errors in Program Development*. IBM System Journal.

Classic: Fagon was the first to formally describe the technique we now know as program trace. His technique is overly formal and structured as was the norm for the day. Nevertheless, the paper is informative and worth the read.

Helfrich, J. (2015). *The Effect of Desk Check on Exam Performance*. CSEIT 2015.

Expound: Interesting result which applies directly to computing students: you can expect a 17% improvement in your grade on a test if you apply the trace method to an algorithm. This is not because students who employ the trace method are better in some way (the experiment controlled for that variable), but rather they made less mistakes.

Chapter 07 Strategy: Decisions

Boole, G. (1848). *The Calculus of Logic*. Cambridge and Dublin Mathematical Journal.

Classic: If you are a fan of historical mathematics, then this sentinel work by George Boole is a must read. Even today, more than a century and a half later, the work is surprisingly understandable.

Mittal, S. (2018). *A Survey of Techniques for Dynamic Branch Prediction. Concurrency and Computation Practice and Experience*.

Expound: Though there is an entire branch of scholarly research on the topic of branch prediction, this recent survey provides a good summary and hits upon several of the issues to consider.

Ding, Z. (2012). *Something You May Not Know About the Switch Statement in C/C++*. Code Project.

Expound: This blog provides a wonderful summary of the constraints compiler designer work under, and some of the solutions they implement to support our code.

Chapter 08 Strategy: Collections

Helfrich, J. (2020). *C++ Data Structures*. Kendall Hunt.

Expound: There are many, many excellent data structure textbooks that are currently under publication. C++ Data Structures was designed to complement Software Design.

Josuttis, N. (2012). *The C++ Standard Library*. Addison Wesley

Reference: Every programming language has its own set of supported data structures. While the C++ standard template library is not the most modern or complete, it is typical of what one can expect to find associated with a modern language. If you are going to spend time working deeply in a single programming language, it is worth your while to purchase a book such as this and read every word.

Chapter 10 Tool: Structure Chart

Kruchten, P. (1995). *Architectural Blueprints – The 4+1 View Model of Software Architecture*. IEEE Software.

Classic: With the many viewpoints describing different aspects of software design, Philippe Kruchten organized and categorized them all into his now famous 4+1 model. All serious students of software design should be familiar with this article and the associated model.

Constantine, L. and Yourdon, E. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press.

Classic: This is what is commonly called a “bookshelf book.” One of those sentinel books that you keep on a prominent bookshelf location in your office. Though the examples and technology are very dated, the underlying principles still hold true today. The chapter that introduces structure charts is “4.1 Flowcharts and Structure Charts,” though they are used as examples throughout the work.

Chapter 11 Tool: Data Flow Diagram

Gane, C. and Sarson, T. (1977). *Structured Systems Analysis: Tools and Techniques*. Prentice Hall.

Expound: Chris Gane and Trish Sarson first described DFDs and it is their notation which is most commonly used today. If you want to know more about DFDs, go to the source!

LeBlanc, D. and Howard, M. (2002). *Writing Secure Code (Developer Best Practices)*. Microsoft Press.

Expound: David LeBlanc and Michael Howard were the first to describe how to use DFDs to find and fix security defects. Note that they use the Tom DeMarco notation.

Chapter 12 Metric: Cohesion

Constantine, L. (1968). *The Programming Profession, Programming Theory, and Programming Education*.

Classic: This is the first paper describing cohesion and coupling. A classic that we should all read!

Myers, G. (1978). *Composite/Structured Design*. Litton Educational Publishing, Inc.

Classic: Though it is very dated, it is one of those “bookshelf books.” If you can find a copy of it, it is worth acquiring and reading. There are several examples and quotes that are worth knowing. Chapter 4 “Module Strength,” goes into depth about the classic model of modularization cohesion.

Martin, R. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

Classic: Best Practice 12.1. emphasizes the importance of choosing function names carefully. Robert Martin devotes an entire chapter (“Chapter 2: Meaningful Names” by Tim Ottinger) to this topic. While this chapter is a borderline rant, the point is unmistakable.

Helfrich, J. (2018). *Measurements of Modularization*. CSEIT 2018/

Expond: An update to Constantine’s classic work, written on the 50th anniversary.

Chapter 13 Metric: Coupling

Constantine, L. (1968). *The Programming Profession, Programming Theory, and Programming Education*.

Classic: This is the first paper describing cohesion and coupling. A classic that we should all read!

Myers, G. (1978). *Composite/Structured Design*. Litton Educational Publishing, Inc.

Classic: This is the same book referenced from the previous chapter. Chapter 5 “Module Coupling,” as its name implies, describes the classic model of modularization coupling.

Helfrich, J. (2018). *Measurements of Modularization*. CSEIT 2018.

Expond: An update to Constantine’s classic work, written on the 50th anniversary.

Chapter 14 Quality: Test Case

Patton, R. (2005). *Software Testing*. Sams Publishing.

Expond: There are dozens of books and thousands of blogs dedicated to how to create test cases. This book is a good start but is not the comprehensive source on the subject. If you want to know more about software testing, consider this the first step in your journey.

Bruegge, B. and Dutoit, A. (2010). *Object-Oriented Software Engineering*. Prentice Hall.

Expond: This excellent book is mostly how to organize large teams to solve complex, real-world problems. That being said, “Chapter 11: Testing” is an overview of the topic. It discusses different types and phases of testing, identifying test cases, drivers, and unit tests. Though all these topics are discussed in this book, Object-Oriented Software Engineering provides an excellent second voice.

Chapter 15 Quality: Driver

Axelrod, A. (2018). *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*. Apress

Expond: There are many books written on the topic of test automation and creating drivers. Most, unfortunately, are not written for developers or are of low quality.

Though most developers need only know the basics, this book provides a good introduction to the larger issues surrounding test automation.

Chapter 16 Strategy: Recursion

Dijkstra, E. (1960). *Recursive Programming*. Numerische Mathematik.

Classic: Edsger Dijkstra was the first to describe recursion from a programming perspective, making this early paper a must read.

Chapter 17 Strategy: Top-Down

Dijkstra, E. (1969). *Notes on Structured Programming*. Academic Press Ltd.

Classic: This is perhaps the foundation of procedural programming as we know it. Edsger Dijkstra calls top-down software design “step-wise program composition.”

Chapter 18 Strategy: Bottom-Up

Beck, K. (2002). *Test Driven Development: By Example*. Addison Wesley.

Classic & Expound: Bottom-up development is one component of the larger test-driven development methodology. We will learn more about this in Chapter 26 Quality: Test-Driven.

Chapter 19 Strategy: Functional

Hadak, P. (1989). *Conception, Evolution, and Application of Functional Programming Languages*. ACM Computing

Expound: Paul Hudak does a great job of explaining why we have functional programming languages and provides an excellent history. If you would like to dive deeper into this programming methodology, then it is worthwhile knowing the “why.”

Michaelson, G. (2013). *An Introduction to Functional Programming Through Lambda Calculus*. Dover Books on Mathematics.

Expound: There are hundreds of books on functional programming, most of which are sent in the context of one of the many functional programming languages. Read this book only after you have mastered the exercises and problems contained in this text. It is also a good idea to read Paul Hadak’s article above before diving into this book.

Chapter 20 Tool: Class Diagram I

Object Management Group (2007). *Unified Modeling Language*.

Reference: It is often useful to go straight to the source. This is the definitive reference of the UML Class Diagram.

Miles, R. (2008). *Learning UML 2.0: A Pragmatic Introduction to UML*. O'Reilly.

Reference: Though some of the examples can be a bit puzzling, this book does a good job of presenting activity diagrams, class diagrams, component diagrams, and many other viewpoints that you may need to use.

Chapter 21 Metric: Fidelity

Chidamber, S. and Kemerer, C. (1994). A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering.

Exound: This is a scholarly article which describes a collection of ways to measure the quality of a class design.

Martin, R., (2014). Getting a SOLID Start. From objectmentor.com

Classic: Robert Martin invented the SOLID OO design principles. The single responsibility principle of SOLID is related to the encapsulation metric of fidelity.

Chapter 22 Metric: Robustness

Patton, R. (2005). Software Testing. Sams Publishing

Exound: The essence of robustness is the quality of tests used to validate the class. There are dozens of books and thousands of blogs dedicated to how to create good tests. This book is a good start, but is not the comprehensive source on the subject. If you want to know more about software testing, consider this the first step in your journey.

Cox, B. (1987). Object Oriented Programming: An Evolutionary Approach. Addison Wesley.

Classic: This book is a piece of history, being one of the first widely-read books on OO design. In “Chapter 1: System Building,” Brad Cox equates system design with military defense planning. This short chapter of twelve pages captures the underlying motivation behind the robustness metric and one of the main goals that encapsulation is meant to achieve.

Chapter 24 Metric: Abstraction

Riel, A. (1996). Object-Oriented Design Heuristics. Addison-Wesley.

Classic & Exound: This excellent book is not specific to the metric of abstraction but provides many great design principles to all aspects of OO design. It is a book to read and re-read every few years. Many of Riel’s heuristics apply directly to abstraction, including 2.1, 2.2, and 2.8.

Wick, M., Stevenson, D., and Phillips, A. (2004). Seven Design Rules for Teaching Students Sound Encapsulation and Abstraction of Object Properties and Member Data. SIGCSE’04.

Exound: Michael Wick et al. offer seven excellent design rules that are both very understandable and very applicable to many coding situations. Each one of these will help you maximize your abstraction score on a given class design.

Chapter 25 Quality: Unit Test

IEEE, (1987). IEEE 1008 Standard for Software Unit Testing.

Classic: While there are certainly many technologies supporting unit testing, a better use of your time and effort would be to understand the IEEE standards on the subject.

Martin, R. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

Classic: What reference list on software design would be complete without Clean Code? This is a modern classic, certainly a “bookshelf book.” Though it is heavily Java-centric, the principles apply to all languages. Oh, and the chapter on unit tests

("Chapter 9: "Unit Tests") is very good. While there certainly are specifics, the overall philosophy of creating unit tests alone is worth the purchase.

Meszaros, G. (2007). *XUnit Test Patterns: Refactoring Test Code*. Pearson Education, Inc.

Exponent: This book is huge! Almost 900 pages devoted to unit tests! Despite the enormous size, this book is surprisingly readable. If you are using unit testing in your everyday workflow, it is worthwhile to spend a few days reading this book. It will be worth your time.

Chapter 26 Quality: Test-Driven

Beck, K. (2003). *Test Driven Development: By Example*. Addison Wesley.

Classic & Exponent: Kent Beck was one of the original inventors of test-driven development. Most consider this book to be the ultimate source for all things TDD. This book is very readable, written in an almost conversational way.

Chapter 27 Strategy: Noun Identification

Abbott, R. (1983). *Program Design by Informal English Descriptions*. Communications of the ACM

Classic: The first published article on the noun identification strategy. It is very readable and provides additional insight into the methodology.

Beck, K. and Cunningham, W. (1989). *A Laboratory for Teaching Object-Oriented Thinking*. OOPSLA'90

Classic: The first published article on CRC cards, a contemporary of the noun identification process. Kent Beck and Ward Cunningham are two of the four original authors of the agile development methodology.

Wirfs-Brook, R. and McKean, A. (2002). *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley

Exponent: This book provides a great deal more depth in the noun identification process. If the chapter in this textbook leaf you thirsting for more, this book will certainly satisfy.

Chapter 28 Strategy: Metaphor

C++ Reference, (2020). *Operator Overloading*

Reference: Perhaps C++ offers more operator overloading opportunities than any other language. The C++ Reference site cppreference.com provides a good overview of the different types of operators that can be overloaded, how they are used, and the necessary syntax. Even if your language of choice is not C++, these pages serve as a useful reference.

Chapter 31 Metric: Adaptability

Hitz, M. and Mantazeri, B. (1995). *Measuring Product Attributes of Object-Oriented Systems*. ESEC'95.

Classic: Martin Hitz and Behzad Montazeri wrote perhaps the most frequently reference paper on OO metrics. If you are to read only one scholarly article on the subject, this would be a good choice.

Chapter 32 Metric: Alignment

Armstrong, J. and Mitchell, R. (1994). *Uses and Abuses of Inheritance*. Software Engineering Journal.

Exponent: There are many software engineers who avoid inheritance whenever possible. James Armstrong and Richard Mitchell do a good job of capturing their concerns and offers. They also offer solutions, many of which are the progenitors of the modern alignment metric. This paper is surprisingly readable and useful to this day.

Chapter 33 Metric: Redundancy

Riel, A. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley.

Classic & Exponent: This book was also mentioned as a good supplement to the abstraction metric of encapsulation. Arthur Riel describes several heuristics that help optimize the redundancy metric, including 5.8, 5.9, 5.10, and 5.11.

Chapter 34 Quality: Debugger

Microsoft Corporation (2020). *First Look at the Visual Studio Debugger*.

Reference: Every IDE has extensive documentation for how to use the various debugger tools. If you need to move to a different technology or learn a new programming language, take the time to learn the details about your debugger. This reference is for the Visual Studio debugger if that is what you happen to use. Replace this citation with the technology of your choice.

Chapter 35 Strategy: Polymorphism

Martin, R. (1997). *The Open-Closed Principle*.

Exponent: Though Bertrand Meyer first described the open-closed principle, Robert Martin perhaps described it best in this white paper. This paper is very readable and very detailed.

Martin, R. (2000). *Design Principles and Design Patterns*.

Exponent: This early white paper does an excellent job describing the open-closed principle, the Liskov substitution principle, the interface segregation principle, and others.

Chapter 36 Strategy: Is-a and Has-a

Riel, A. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley.

Classic & Exponent: Arthur Riel has nearly a dozen OO design heuristics devoted to *is-a* and *has-a*. These are all under “Chapter 4: The Relationships Between Classes and Objects.”

Chapter 37 Strategy: Object Creation

Gamma, E. et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley

Classic: This was the first widely circulated book which described many of the design patterns we use today. It certainly qualifies as a “bookshelf book,” it can be hard to understand unless you already possess a good understanding of design patterns.

Chapter 38 Strategy: Algorithm Abstraction

Gamma, E. et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley

Classic: It is difficult to suggest any further reading about design patterns without listing the original work. This book will be cited many times.

Freeman, E. & Freeman, E. (2004). *Head First Design Patterns*. O'Reilly

Expound: The decorator design pattern is particularly well presented here. Unfortunately, they give only a brief introduction to the strategy design pattern.

Chapter 39 Strategy: State

Dijkstra, E. (1959). *A Note on Two Problems in Connexion with Graphs. Numerische Mathematik*

Classic: In this, one of the most famous algorithmic research papers, Dijkstra describes the shortest path first (SPF) algorithm. Though it was written more than fifty years ago, this short paper (only 2½ pages!) describes the most important algorithm in graph theory.

Object Management Group (2007). *Unified Modeling Language*.

Reference: The state diagram is part of the UML family of viewpoints. This is the definitive reference of the UML Activity Diagram.

Gamma, E. et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley

Classic: The state design pattern was initially described by the GoF.

Chapter 40 Tool: Component Diagram

Object Management Group (2007). *Unified Modeling Language*.

Reference: It is often useful to go straight to the source. This is the definitive reference of the UML Component Diagram.

Chapter 41 Tool: Design Description

IEEE, (2009). *IEEE 1016 Information Technology—Systems Design—Software Design Descriptions*.

Reference: This document is quite large and difficult to read. However, it represents the gold standard of the SDD standard. If you ever need to create a formal SDD for legal or contractual reasons, have a physical copy of this document on your desk.

Royce, W, (1970). *Managing the Development of Large Software Systems*.

Classic: There is so much misunderstanding about the waterfall SDLC model! Everyone should read Royce's original paper and form their own opinion.

Chapter 42 Quality: V-Table

IEEE, (2008). *IEEE 829 Standard for Software and System Test Documentation*.

Reference: Though most organizations have a template and guidelines for test documentation, all should be based on the IEEE standard. Certainly, all testers and quality assurance engineers should have a copy of this document. Developers and project managers should also be familiar with its contents as well.

IEEE, (1998). *IEEE 1012 Standard for Software Verification and Validation*.

Reference: A critical standard for all those considering a career as a QA engineer or a SDET. This is also useful for anyone who will spend a career working with them!

Chapter 43 Strategy: Large Inheritance Trees

Linnaeus, C. (1758). *Systema Naturæ*.

Classic: Carl Linnaeus's original work introducing the taxonomy of living things which we use to this day. This is perhaps one of the most important scientific writings, comparable with Darwin's Origin of Species and Newton's *Philosophiæ Naturalis Principia Mathematica*.

Small, E. (1989). *Systematics of Biological Systematics (or, taxonomy of taxonomy)*.

Expound: An excellent article which discusses the considerations that go into making a taxonomy. The constraints on scientific classification that Small discusses are similar to what a software engineer will encounter while designing or modifying a large inheritance tree. A great quote from the paper is "It is good in being a stable, widely used, and at least somewhat reasonable system. It is bad in having been erected subjectively and somewhat arbitrarily, in lacking the flexibility of permitting the incorporation of better organization as it is conceived, and indeed in providing a rather rigid way of consideration."

Chapter 44 Strategy: Message Passing

Gamma, E. et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley

Classic: The mediator, chain of responsibility, observer, and visitor design patterns were first described in this book.

Chapter 45 Strategy: Separation of Concerns

Dijkstra, E., (1982). "On the role of scientific thought". *Selected writings on Computing: A Personal Perspective*

Classic: The concept of separation of concerns was first introduced here. Talk about how separations of concerns came from here. A classic paper that we should all read.

Martin, R. (2002). *Agile Software Development, Principles, Patterns, and Practices*

Expound: The first detailed of the single responsibility principle is described in this book.

Chapter 46 Strategy: Command Passing

Gamma, E. et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley

Classic: The interpreter and command (encapsulated invocation) design patterns were first described in this book.

Chapter 47 Strategy: Interfaces

Gamma, E. et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley

Classic: Again, we are referencing the “Gang of Four” (GoF’s original work describing these design patterns. All four interface design patterns are described in their book.

Freeman, E. & Freeman, E. (2004). *Head First Design Patterns*. O’Reilly

Expound: The style of this book could not be more different than the GoF’s book. For that reason, it is a nice companion. They do a good job describing the adapter, the façade, and the proxy, but their introduction to builder is too brief.

Chapter 49 Strategy: Layered

Bushmann, F. et al. (1996). *Pattern-Oriented Software Architecture*. Wiley

Expound: Though this book is quite dated, it provides an excellent overview of large system and distributed system design. The chapter on layers is particularly relevant to Chapter 49.

Chapter 01 Tool: Flowchart

Best Practice 01.1 Make the flowchart more understandable by making the arrows point in the same general direction

It is easier for a human to understand a flowchart if the arrows are mostly top-to-bottom.

Chapter 04 Metric: Maintainability

Best Practice 04.1 Move algorithm data into constants

One of the easiest ways to make an algorithm data-driven is to move the data component of an algorithm into constants. This makes the algorithm more malleable.

Best Practice 04.2 Avoid duplicate code

If there is more than one copy of code in the codebase, then this is a good indication that there is a maintenance problem.

Best Practice 04.3 Avoid overly complex Boolean expressions

Boolean expressions can be notoriously difficult to create and understand.

Best Practice 04.4 Keep it linear

Whenever possible, the path through the code should be simple to follow. Functions should have one entrance and one exit. The same with loops.

Best Practice 04.5 Thoroughly test the code before refactoring it

When a change needs to be made in code that exhibits refactor malleability, it is necessary to first obtain a detailed understanding of how the code works. As a bare minimum, exercise the code with a wide variety of input. Hopefully the code works as expected without any bugs.

Best Practice 04.6 Only refactor code when there is no other option

Refactoring existing code is an expensive and risky proposition. When we do so, the testing investment is essentially thrown out, requiring everything to be retested.

Best Practice 04.7 Design code to simplify refactoring or to eliminate the need to refactor

Prefer straightforward solutions over clever ones.

Best Practice 04.8 Rewrite only as a last resort

It seldom is a good idea to just rewrite code you don't like. Often wholesale rewrites create more problems than they solve. An incremental approach is far better than throwing everything out and starting over. Prefer refactoring over rewriting.

Best Practice 04.9 Manage technical debt carefully

Technical debt is the accumulation of maintainability problems resulting from engineers pursuing easy solutions rather than doing things "right." The project manager should carefully record those areas that are accumulating technical debt and schedule time to fix problem areas before they become a competitive disadvantage to the team.

Chapter 05 Quality: Assert

Best Practice 05.1 Use asserts for system errors, use IF statements for user errors

A program should not throw an assert when the user makes a mistake. Instead, give the user a friendly message and recover gracefully.

Best Practice 05.2 Use asserts for runtime errors, let the compiler to catch compile time errors

Catch as many errors as possible at compile time through robust type-checking.

Best Practice 05.3 Draft asserts during the design phase

Note assumptions made in the design phase which will later be turned into asserts.

Best Practice 05.4 Turn comments into asserts

If an assumption is stated in a comment, turn it into an assert whenever possible.

Best Practice 05.5 Add a comment next to asserts describing how to fix a bug

Add a few comments describing how the bug could be fixed if this assert ever did fire.

Best Practice 05.6 Document that which should be unreachable with an assert

When writing the code, you may notice that something is unreachable or impossible. This might even be true... right now! After a few rounds of edits and bug fixes, however, it may no longer be true. Document these unreachable points with an assert.

Best Practice 05.7 Add an assert even if there is "no way possible" for it to fire

If a statement is obviously true, don't hesitate to put an assert there. There might be something you have not considered or perhaps the code will change after the last time you looked at it. Just because you can't think of any possible way that the condition is not met does not mean that a way does not exist.

Best Practice 05.8 Validate pre-conditions before every algorithm

No matter how sure you are that the input data to an algorithm is correct, it is worthwhile to write a couple asserts to validate them.

Best Practice 05.9 Write asserts before you refactor code

A great way to validate the program works as you expect is to first add asserts. These asserts represent your best understanding of how everything works.

Best Practice 05.10 Validate post-conditions after every algorithm

Make sure you send valid data the clients of your function.

Chapter 07 Strategy: Decisions

Best Practice 07.1 Simplify every Boolean expression

It is often the case that needlessly complex Boolean expressions find their way into the code because the developer stuck with the first solution that presented itself. When you notice these things, take a moment to untangle and simplify the Boolean expressions.

Best Practice 07.2 Create an experiment to determine the branch prediction algorithm for your CPU

When making performance-related decisions on high-stakes code, it is a good idea to write some instrumentation code to validate your decisions. As a general rule, most performance optimization decisions should be backed up with experimental data.

Best Practice 07.3 Avoid redundancy in Boolean expressions

Never say something twice when you can say it only once.

Best Practice 07.4 Put the most common case in a multi-way IF statement first

It is therefore much more efficient to put the most common cases at the front of a multi-way IF statement and the least common cases at the back.

Best Practice 07.5 Use nested IF statements when there are several options

Nested IF statements are $O(\log n)$ whereas linear IF statements are $O(n)$.

Best Practice 07.6 Avoid empty bodies in IF statements

This often indicates a redundant or unnecessary Boolean expression.

Best Practice 07.7 Use integers in CASE labels whenever possible

In many programming languages, this makes the code execute faster.

Best Practice 07.8 Use integral SWITCH controlling expressions with minimal gaps whenever possible

In many programming languages, this makes the code execute faster.

Chapter 08 Strategy: Collections

Best Practice 08.1 Structures should not contain large collections, collections that grow, or collections that shrink

A structure would not be suitable for storing a collection of pixels representing an image or a collection of transactions representing an account register.

Best Practice 08.2 Structures should map to design concerns

When variables are collected into a structure, make sure the structure should mean something in the problem domain. In other words, structures should not be a random collection of variables. The programmer should be able to readily explain the purpose structure to another programmer or even the project sponsor.

Best Practice 08.3 Name your structures carefully

In most programming languages, structures are defined before they are used and the definition requires a name. Choose your name carefully. A good structure name should be precise, complete, and singular.

Best Practice 08.4 Only define a structure globally if it is to be used by the entire program

Recall that structures need to be defined before they are used. The visibility of the structure definition is called the scope. As a general rule, the scope of a programming construct should be related to the utility of the programming construct.

Best Practice 08.5 Don't use a tuple when you could use a structure

Tuples are useful for temporary groups of related values; if your variable going to be used beyond a single function or two closely related functions, use a structure instead.

Best Practice 08.6 Add a comment when referring to member variables within tuples

Because most languages do not allow the programmer to name or label a tuple member variable, it is almost always helpful to add a comment describing its use.

Best Practice 08.7 Use tuples to get more than one return value from a function

There is a strong analogy between the parameters passed to a function (an ordered set of anonymous values of different data types) and tuples. Note that most programming languages only allow one return value from a function. You can address this asymmetry by returning a tuple from a function.

Chapter 09 Strategy: Loops

Best Practice 09.1 Simplify loop controlling expression as much as possible

Because the controlling Boolean expression is evaluated on every iteration of the loop, the efficiency of execution of that controlling expression is very important.

Best Practice 09.2 Predicates called from loop controlling expressions should be extremely efficient

While performance is very important in all aspects of software design, it is critical in predicates called in the controlling expression of a loop. Every effort should be taken to streamline their evaluation.

Best Practice 09.3 Avoid unnecessary code in a loop body

Because a loop body is executed multiple times, performance is critical. A common mistake for new programmers to make is to put unnecessary code in a loop body.

Best Practice 09.4 Make starting conditions of a counter-controlled loop explicit

If a starting condition is set in a location of the code removed from the loop itself, add a comment indicating this.

Best Practice 09.5 Move complicated code from the end-condition to the start-condition when possible

If a program needs to count from X to Y , it might be that computing X is easy whereas computing Y is difficult. Since the start-condition is executed only once and the end-condition is computed many times, it is much more efficient to count down from Y to X than it is to count up from X to Y .

Best Practice 09.6 Be weary of a counter-controlled loop without an increment statement

If a counter-controlled loop has no increment statement, it usually means that it is an event-controlled loop in disguise. In cases such as these, do not use a FOR loop but a WHILE loop instead.

Best Practice 09.7 Counter-controlled loops should have the increment statement in the FOR loop

This makes it easier to verify that the loop behaves the way the programmer intended.

Best Practice 09.8 Each loop should have exactly one increment statement

Multiple increment statements are difficult to manage and result in many bugs. Avoid them!

Best Practice 09.9 Use a post-condition for a sentinel-controlled loop if that option is available

Because the loop-control logic for a sentinel-controlled loop resides inside the body of the loop, we always need to enter the body of the loop at least once. This means that a post-condition loop is better for sentinel-controlled loops.

Best Practice 09.10 Consistently use true- or false-controlled sentinels in your codebase

Whether your sentinel is true-controlled or false-controlled is up to you. It is usually a good idea to use the same convention throughout, so the code is more easily understood.

Best Practice 09.11 Make the loop control logic in a sentinel-controlled loop obvious

Take care to make the loop control logic obvious in your program. At a minimum, use prominent comments and a very consistent way of setting the sentinel variable so reader of the code can understand how the loop functions.

Best Practice 09.12 Memorize all the collection loops in each programming language you use

Collection-controlled loops are such a common part of programming that we should not have to think of how they work. Each collection loop should be memorized in your programming language of choice so they are used consistently in your code and you never make a mistake using them.

Best Practice 09.13 When working with linked lists, be weary of infinite loops

The other types of container loops offer the assurance that each increment will bring us closer to the end-condition. This assurance is not present with linked lists.

Best Practice 09:14 If your programming language has an iterator, prefer that loop over others

If a language has an iterator implementation for a given collection, it probably has another loop implementation as well. When multiple options are available, use the iterator loop. It is guaranteed to be at least as fast as the other options and is more generic. It also give the author of the collection freedom to alter implementation details without impacting client code.

Chapter 12 Metric: Cohesion

Best Practice 12.1 Choose function names carefully

The name we bestow upon a variable, a function, a class, a file, or indeed any programming construct is vitally important. Not only does it allow us to find the construct amongst a sea of distractors, but it also informs us of its use. We should be very precise and descriptive in our name selection.

Best Practice 12.2 Write accurate and precise comments

This comment should, in just a few words, describe what the block is meant to accomplish.

Chapter 13 Metric: Coupling

Best Practice 13.1 Loose coupling starts with the structure chart; ask coupling questions with every arrow

Coupling errors are easy to identify and fix in a structure chart, difficult in the code base.

Best Practice 13.2 Isolate complex interfaces with a façade

In order to keep protect the quality of our code, we need to shield our program from the complexity of external influences. This can be accomplished with a technique called a façade.

Chapter 14 Quality: Test Case

Best Practice 14.1 Make every test case traceable to a requirement

If you cannot trace a test case directly to a requirement, stakeholder need, or user scenario, then it is likely that test case has validity challenges. The more different it is to make that connection, the more you should be concerned.

Best Practice 14.2 Be very specific about the test case's pre-conditions

Make sure that test case's pre-conditions are specific enough that the test will always be run in the same environment.

Best Practice 14.3 Avoid generalities in test cases

A test-case can have reliability challenges when the instructions say things like “enter some text.” The phrase “some text” may mean the word “hamster” to one tester and the entirety of the Old Testament to another. A more reliable test case should specify exactly the text to be entered.

Best Practice 14.4 Reduce and simplify steps whenever possible

Do not describe in two words what can be expressed with one.

Best Practice 14.5 Automate whenever possible

While it is acceptable to have a few manual tests in your list of test cases, it is better if everything is automated. Write code to do your heavy lifting.

Best Practice 14.6 Beware of false positives, false negatives, unhelpful logs, and verbose logs

If your test cases are buggy, then you might find yourself tracking down many false leads. This can waste your time and cause you to overlook important issues.

Best Practice 14.7 Don’t assume that current behavior is the expected behavior

Look at the system with a critical eye. Just because it has “always been done this way” does not mean that it is right. Do some research to see what the client or the user expects.

Best Practice 14.8 Think BAOE (Basic, Alternate, Options, and Exceptions) when writing test cases

The basic flow is the mainstream scenario, constituting the most common user experience. This must work flawlessly. The alternate flow are variations and nonstandard scenarios. Each of these will probably be explored by the user on occasion. Options represent settings that the user may manipulate. Not only should each option be represented in a test case, but all related combinations should be as well. Finally, exceptions represent errors. Each error condition should be represented with a test case.

Chapter 15 Quality: Driver

Best Practice 15.1 Use manual drivers early in the development process

When the team is early in the development process and perhaps the code is not even completely written, manual drivers are usually the best choice. They are very easy to set up and offer the highest degree of flexibility.

Best Practice 15.2 Use automation after the code is written and the function or program is near code complete

When the code is complete and the team is prepared to run through all these test cases on the function under test, automation is a better choice than manual drivers.

Best Practice 15.3 When possible, put driver code in a separate file.

It is usually a good idea to put driver code in a separate file so there is no chance that it could find its way into customer-facing code

Chapter 16 Strategy: Recursion

Best Practice 16.1 Do not use recursion in performance-critical areas of the application

When pursuing a recursive solution, make sure the number of recursive calls is carefully managed. $O(\log n)$ applications (such as binary search functions or recursive sorting algorithms) are typically acceptable. Algorithms making $O(n)$ function calls should not use recursion unless the size is n is very small.

Best Practice 16.2 Avoid multiple-recursion

Multiple-recursion is when there are more than one recursive variables or there is more than one recursive function call. When not managed carefully, this can lead to $O(2^n)$ algorithmic efficiency which is intractable in all but the simplest cases.

Best Practice 16.3 Avoid indirect recursion

Indirect recursion is where **FunctionA()** calls **FunctionB()** which, in turn, calls **FunctionA()**. In these cases, it is not always apparent that recursion is at play. They are difficult to understand and difficult to debug.

Best Practice 16.4 Make progress towards the end-condition with every iteration

If your algorithm does not work towards the end-condition with every iteration, you will have an infinite loop. If possible, write debug code to ensure that the problem size is smaller with every successive call.

Best Practice 16.5 Be wary of unnecessary or unintentional recursion

Make sure each use of recursion is intentional, with a well-defined progress step and a well-defined end-condition.

Chapter 17 Strategy: Top-Down

Best Practice 17.1 When you find yourself painted into a corner, back out and start again

We often paint ourselves into a corner because we did not understand the problem well enough to when we started. This is expected and a normal part of the development process. You are never as ignorant about a project as you are right now. When we recognize that we are in this case, back out one or two levels in the design process until the source of the problem is discovered.

Best Practice 17.2 Top-down is for design activities, bottom-up is for development activities

The design process should be top-down, working from the general to the specific. The process of writing the code, however, should be bottom-up.

Best Practice 17.3 Large projects should be built iteratively with heavy involvement from the client

Do not go too far down the design path without consulting the client. This helps identify false assumptions and other mistakes which could lead to costly redesigns. The closer we work with the client, the better we understand and can meet the client's needs.

Best Practice 17.4 Always have a master plan, and be prepared to adjust it

The project should always have a master plan as to what the final code will look like. It is understood that this master plan is a draft, subject to change. When the inevitable design change is required, the master plan needs to be updated so it always reflects the latest thinking on the part of the design team.

Chapter 18 Strategy: Bottom-Up

Best Practice 18.1 Use top-down or bottom-up elicitation depending on the needs of the client.

A good requirements engineer is skilled at both top-down requirements elicitation and bottom-up. Choose whichever is best for the client.

Best Practice 18.2 Top-down design is better than bottom-up in most scenarios

Top-down designs tend to be elegant and cohesive. Bottom-up designs tend to be overengineered or underengineered.

Best Practice 18.3 Bottom-up development is easier, has less wasted effort, and takes less time

Bottom-up development begins with the simplest and most independent function. This function is written and tested before working on the next simplest and most independent. As a rule, bottom-up is the recommended path for the development activity.

Best Practice 18.4 Implement bottom-up testing in most software development processes

Bottom-up testing involves starting with the smallest unit of software (typically a function) and generally working up to larger components. Thus, the last thing tested is the holistic system, done only after each individual component has been verified. All indicators suggest that bottom-up testing is superior to top-down testing.

Best Practice 18.5 Keep the structure chart up to date so it faithfully represents the project

Whenever the structure chart is updated, make sure to add the corresponding stubs to the project. Also, when a function is changed, update the structure chart so it is a faithful representation of the project.

Best Practice 18.6 Never leave the code in a state when it does not compile

Do not let yourself be interrupted from your work when the code is in a state where it is not compiling. Mostly this means that you should make sure you have time to finish a function before starting to write code. If this absolutely cannot be avoided, comment out the incomplete code and begin the process anew the next day.

Chapter 19 Strategy: Functional

Best Practice 19.1 Variable names should be nouns because they describe things

Variables represent things. Nouns are for persons, places, or things. Therefore, variables should be nouns.

Best Practice 19.2 Function names should be verbs because they describe actions

Functions represent algorithms which do things. Verbs represent action. Therefore, functions should be verbs.

Best Practice 19.3 Classes should be nouns because they describe things

A class is a new data type from which objects (variables) are created. Therefore they should be nouns.

Best Practice 19.4 Make a function pure whenever possible; it simplifies coupling

As a general rule, a programmer should strive to make functions pure. This reduces complexity and simplifies integration with the rest of the program.

Best Practice 19.5 Make functions pure whenever possible

All functions in your design should be pure with few exceptions. Each exception should have a very good reason for breaking such a rule. Common examples include file I/O, user I/O, and random number generators. If a function is not pure and does not fall into one of these categories, then careful consideration should be taken as to why that is the case.

Best Practice 19.6 Favor iteration over recursion

As a rule, iterative solutions outperform recursive solution. There are some exceptions to this rule. One example is iterating over a binary tree or similar complex data structures. There the recursive depth is small and the iterative equivalent is complex. For most other solutions, iteration is superior.

Best Practice 19.7 Master recursion

Functional programming languages are convenient and powerful tools to help you practice recursion. Many a programmer avoids an elegant recursive solution for a cumbersome iterative one due to their lack of familiarity with recursive. Let this not be you! Recursion is a potent tool that should be in every programmer's tool bag.

Chapter 22 Metric: Robustness

Best Practice 22.1 Start the test plan early

It is generally a good idea to make your first draft of the test plan at the very beginning of the project. The more you think through the test cases, the more likely it will be that you will handle them in the first draft of your algorithm rather than at the end.

Best Practice 22.2 Produce well-defined output, but accept poorly formed input

The data flowing out of a class should always be in a well-defined state. It should be predictable and reliable. However, do not expect the input into a class to be the same. Your class should work reliably regardless of what it is given. This best practice should be expressed as: "have higher expectations for yourself than you do for others." Perhaps this best practice also applies to life in general.

Best Practice 22.3 Maintain the test plan as you go

As you learn more about the project and how your code will be used, you naturally discover test cases. Rather than trying to remember them, add them to your test plan and codify them into the automation.

Best Practice 22.4 Design white box tests as you implement the code

The best time to write code that validates the data structures of your class is when you implement the data structures of your class. At that moment, all the nuances are in your head. Don't let them fade away; codify them!

Best Practice 22.5 Use formal representations whenever possible

It is easy to use layman's terms to describe how an algorithm or class works. It is also easy to gloss over important details. The more precise our language, the less wiggle room we leave for ambiguity.

Chapter 23 Metric: Convenience

Best Practice 23.1 Ask the client many detailed questions

The deeper the provider understands of the client's needs, the more convenient the interface can be. It is impossible to create a truly convenient interface when nothing is known about how the client will use the class..

Best Practice 23.2 Monitor how the class is used

When the client starts to utilize the class in the application, the provider should carefully study how it is used. If the client starts to use the class in a different way than was initially discussed, be ready to change the interface to better meet the client's needs.

Best Practice 23.3 Try to be overly helpful

The provider should constantly be on the lookout for tasks the client can do but would be more appropriate if handled in the class. The more complexity can be removed from the application into the class, the better!

Chapter 24 Metric: Abstraction

Best Practice 24.1 Design public interfaces before considering implementation details

This is related to the property of convenience where public interfaces are designed to be as useful as possible to the client. This should be the first step. Next, design the private member variables with an eye towards fidelity. If we do it the other way around, then a key implementation detail (how we store the data) is likely to be revealed to the client.

Best Practice 24.2 Be mindful of edge conditions and error states

It is often our corner cases that betray us. Usually these are easier to handle early in the design process and more difficult late into implementation. In other words, an inexperienced class designer can paint himself or herself into a corner with error states and edge conditions if they are not considered early in the development process.

Best Practice 24.3 Be wary of performance traps

It is easy to get fixated on efficiency and performance. This is not a bad thing; performance often has a huge impact on the practicality and usability of our code. Over-fixation, on the other hand, can be a problem.

Chapter 25 Quality: Unit Test

Best Practice 25.1 The scope of a unit test must be a single function or method

It should be the goal of every unit test author to isolate the class or function under test to the greatest extent possible.

Best Practice 25.2 Tests should depend on unit requirements, not on implementation details

The purpose of unit tests is to verify that the unit under test functions how it is meant to operate. In other words, the unit tests are to verify the unit's contract, not the implementation details. If we rely too much on back door methods, then we can have over specified tests.

Best Practice 25.3 Try to test every codepath in the unit under test

A test suite with good coverage ensures that at least one unit test exercises every line of code. This increases the chance that a mistake in the unit under test will be caught by a unit test.

Best Practice 25.4 Isolate the class under test from the rest of the system

Each collaborator should be identified and, when possible, replaced with a double to minimize interaction between units in the unit test.

Best Practice 25.5 Minimize test overlap

Duplicate tests simply increase the cost of maintaining all that test code. Ideally, there should be one unit test per test case, and one test case per feature or condition in the unit under test. This means that changing a line of production code primarily impacts only one unit test, and each unit test maps to a single part of the unit under test.

Best Practice 25.6 Unit tests must be written by the developer who wrote the code to be tested

Since unit tests are designed to be white box, only the developer can craft tests that correctly validate the inner workings of the system. It is therefore the developer's job to create unit tests in conjunction with the software that is under test, and to maintain the tests when the design changes.

Best Practice 25.7 Unit tests must be reliable, accurately reporting whether there is a defect in the code

Every effort should be made to avoid false positives because they are a distraction to the development team. False negatives are the most dangerous. They give the development team a false sense of confidence that there are no bugs in the system.

Best Practice 25.8 Make the unit tests obvious

Obvious test code decreases the cost of maintaining the test code and reduces the amount of effort required to fix a bug in production code.

Best Practice 25.9 Test methods must not have loops

Every test method must have a completely linear algorithm, progressing from the setup step to the exercise, and on through to the verify and teardown.

Best Practice 25.10 Test methods must not have conditional logic

Since the fixture is completely predictable, there should be no IF statements or other conditional logic.

Best Practice 25.11 Unit tests must be executed frequently

To get the most out of your investment in unit tests, they should be run frequently. In most cases, this means that every unit test is executed every time the system is run.

Best Practice 25.12 Unit tests must require no effort or user intervention to run

If it is difficult to run unit tests, then it is unlikely that the developers will run them as often as they should. It is good if it is easy to run unit tests. It is better if unit tests are run every time the application is run. The best possible solution is to make it difficult to not run the tests with every execution.

Best Practice 25.13 Unit tests must run quickly

The longer it takes to run unit tests, the less often they will be run. Therefore, a key aspect of a project's unit test strategy is to keep execution time of unit tests short.

Best Practice 25.14 Unit test should exist in the codebase alongside the code it is meant to verify

When test code is collocated with the production code to be tested, then it is easier to maintain both.

Best Practice 25.15 Make the production code easy to test

Easily tested code tends to be higher quality code.

Chapter 26 Quality: Test-Driven

Best Practice 25.1 Keep It Simple (KISS)

The KISS design principle states that simple designs tend to work better than complex ones.

Best Practice 25.2 Before refactoring working code, make sure you have a backup

Refactor bugs can be subtle and difficult to fix. Attempting to refactor code without a robust set of unit tests is risky. Simple code inspections rarely reveal the problem. However, doing so with strong unit tests brings a certain level of confidence.

Best Practice 25.3 It is easier to re-refactor than it is to fix refactoring bugs

If you find that your refactoring caused a unit test to fail, it is most often easier to back out of the refactor and start again. It will probably take less time and certainly less angst.

Chapter 27 Strategy: Noun Identification

Best Practice 27.1 Before settling on a data type, enumerate the possibilities

We are usually better at selecting an option from a list of possibilities than we are to come up with the best answer “out of the blue.” It is helpful to jot down a few ideas and perhaps listing pros and cons before settling on an answer.

Best Practice 27.2 Longevity benefits from formality

When a variable’s lifespan exists completely within a single function and is never shared with a collaborator, then a lightweight solution is probably better. This is especially true if it is used in the program only once. Here, it is best to favor built-in data types such as simple variables, tuples, or arrays. However, when a variable exists within a class, when it is passed between functions, or when it is utilized many times throughout the application, then a higher degree of formality is probably better. This is especially true if more than one developer will use it. In cases such as these, favor a custom data type such as a class.

Best Practice 27.3 Only use a class when there is no better alternative

Many new object-oriented designers create more classes than is necessary. This results in an overly complicated design, poor performance, and slow development time. The inclusion of a class in the design should improve the design, and the onus is on every class to prove its worth. In other words, do not overengineer.

Best Practice 27.4 Classes represent things, not collections of verbs

A class should always represent a thing. It should never represent a collection of verbs. Every single class should have member variables.

Best Practice 27.5 Identify the data type of variables and objects

When a variable is discovered in the requirements elicitation process, it is probably an important one. We need to have a crisp understanding of its data type and/or any related class.

Best Practice 27.6 Create a DFD for important variables

If a variable important enough to be discovered in a requirement, then a thorough understanding of its life cycle is needed. We need to know how the associated data entered the program, the various transformations they went through, where they were stored, and how they were used. A DFD is a great tool to discover these answers.

Best Practice 27.7 When an implied noun is discovered, make it explicit

When the wording of a requirement is such that the noun associated with an adjective is not directly stated, then rewrite the requirement so it is abundantly clear. Remember that requirements are not meant to be works of literary art; they are meant to be unambiguous.

Best Practice 27.8 Enumerate the alternatives

Adjectives usually represent a broader class of possible values. Ask the client to enumerate the possible values and codify them in the design once you discover them.

Best Practice 27.9 If an adjective can be changed, discover how

If a changeable attribute is discovered, its entire life cycle needs to be understood just as it would be for a proper noun. This probably means a DFD will need to be created to document this life cycle.

Best Practice 27.10 Ask cohesion questions with every action verb

As soon as an action verb is discovered, immediately start asking cohesion questions. Begin the process of identifying subprocesses using a structure chart, making every function strongly cohesive.

Best Practice 27.11 Identify all the proper nouns and common nouns associated with every action verb

In many cases, the input and output for a function is well understood, utilizing previously described data types and variables. Oftentimes, however, an action verb will refer to something new. Be on the lookout for hidden parameters; many a class is discovered because it is needed to fulfill an action verb found in a requirement.

Best Practice 27.12 Methods can be distinguished from functions when there is a strongly associated noun

Determine whether the verb is closely associated with a noun or whether it is standalone. When there is an associated noun, then we are probably talking about a method. Hopefully, at this point the corresponding class has already been discovered and described. When there is no associated noun, this verb is probably a function. Note that there are many cases when it could go either way.

Best Practice 27.12 Speak with more than one client whenever possible

Aside from the simplest problems, it is unusual for a single individual to completely understand the problem domain. Interviewing more than one client allows for different perspectives on the system and more specialized areas of expertise.

Best Practice 27.13 Adapt, don't adopt

As with all design methodologies, the purpose of the noun identification process is to help the designer channel his or her creativity. Adapt development methodologies to meet your specific needs; do not adopt those of another.

Best Practice 27.14 Continually validate your requirements list with the client

Use cases from the client are often in vague and general terms. The developer's job is to remove ambiguity at every step in the process, as there is no room for ambiguity in the coding phase. Every requirement should be written with an eye to turning it into code.

Best Practice 27.15 Be as specific as possible

When converting a use case into a requirement, be mindful of implied items. When the client and/or the developer have a simplistic view of the system, that simplistic view often fails to capture the complexity that the user may require.

Best Practice 27.16 Prefer reusing existing functionality over building new functionality

With every new component added to the design, ask yourself if that functionality already exists in the system.

Chapter 28 Strategy: Metaphor

Best Practice 28.1 Use the prefix “set” for any method that updates a class’ attribute(s)

If a class has only one setter or if the intent is unambiguous, then the single word “`set()`” is appropriate for the method name. However, if more than one setter is needed, then “`set`” is commonly used as the prefix to the full method name: `setRow()`.

Best Practice 28.2 Use the prefix “get” for any method that retrieves an attribute from a class

If a class has only one getter or if the intent is unambiguous, then the single word “`get()`” is appropriate for the method name. However, if more than one getter is needed, then “`get`” is commonly used as the prefix to the full method name: `getRow()`.

Best Practice 28.3 Input and output parameters should be in a format convenient to the client

To maximize the metric of convenience, input parameters from getters should be in a format that is fitting and appropriate for the client.

Best Practice 28.4 The name of the callback should reflect the type of event it is responding to

Because the way in which a callback is invoked is more subtle than that of a normal function or method, it is a good idea to reflect its purpose in the name.

Best Practice 28.5 The name of a predicates must indicate what `TRUE` and `FALSE` means

When a predicate function returns `TRUE`, that value must mean something. The name of the function needs to unambiguously describe what that means.

Best Practice 28.6 Use the assignment operator whenever an object can be copied

Each instance of a copy constructor should be paired with an instance of the assignment operator. Whenever the client may wish to create a copy of an object, provide an explicit assignment operator even when the default one created by the compiler does what you expect.

Best Practice 28.7 Make sure that `x += y` works the same as `x = x + y`

When using a compound metaphor such as `+=`, make sure it works exactly the same as its components.

Best Practice 28.8 If you define greater-than, then you should define less-than

Since greater-than is a reflexive operation, `X > Y` should work the same as `Y < X`.

Best Practice 28.9 Define one comparison operator in terms of another whenever possible

Less code and less redundancy result from defining one operator in terms of another. This should be done whenever possible.

Best Practice 28.10 Use the access operator only when the function is O(log n) or better

Most programmers know that referencing an element out of an array is O(1). If the access operator is O(n) or worse, then this would be an inappropriate metaphor application.

Best Practice 28.11 Avoid arithmetic operator overloading unless the metaphor is unambiguous and clear

Because the arithmetic operators can mean different things to different people, it is generally a good idea to avoid them. The exception is when working with a class that represents a number: currency, measurements, distance, and things like that. Here, there is little opportunity for confusion.

Chapter 29 Strategy: Data Protection

Best Practice 29.1 Always provide a constructor for a class

If your programming language has a constructor (as virtually all do), then every single class definition should include one. This is true regardless of whether the compiler will create one for you if it is not done explicitly.

Best Practice 29.2 Always create a default constructor

Even if the compiler-generated default constructor does what you want it to do, it is a good practice to write one anyway. This makes the code easier to understand and makes future enhancements easier to implement.

Best Practice 29.3 Assert on the validity of input parameters for copy constructors

A wise programmer would validate that every input object to a copy constructor is in a valid state. Otherwise, bugs introduced in the input object will be transferred to the newly created object where they will be much more difficult to find.

Best Practice 29.4 Always create a copy constructor

Generally, there are few good reasons for not creating a copy constructor. They reduce ambiguity and make it more difficult to introduce bugs.

Best Practice 29.5 Use a move constructor to avoid unnecessary copies

If your programming language has a move constructor, then significant performance gains can be made when the data contained within a class are large. Note that the result of a move is that the input class is returned to the default empty state.

Best Practice 29.6 The input object to a move constructor should be returned a valid state

After the data from a move constructor have been moved to the new class, the original class must remain in a good state. Usually this entails reverting the host class to the same state that results from the default constructor.

Best Practice 29.7 Conversion constructors must validate their input

Conversion constructors must validate each input parameter to make sure that no invalid data makes it into the member variables.

Best Practice 29.8 Call public mutators whenever possible

In most cases, the data passed to a conversion constructor can also be passed to a mutator. If this is the case, do not duplicate the code – just call the mutator directly from the conversion constructor.

Best Practice 29.9 Fail gracefully when invalid data are sent

Each constructor has the obligation to leaving the newly created object in a valid state regardless of the provided input. In the case of a conversion constructor, invalid data should result in an object with the default value.

Best Practice 29.10 Avoid public member variables; use accessors and mutators instead

As a rule, it is dangerous to allow public access to a class' attributes. When doing so, we are relying on the client to perform data validation. This makes it hard to be sure the member variables are in a good state. There are a few exceptions to this rule. If a member variable is a Boolean, it might be very difficult to make it invalid!

Best Practice 29.11 All client-provided input should be validated

Treat all client-provided input with suspicion. We do not do this because all clients are malicious (though some might be) or ignorant (which some are). Instead, we do this to alleviate the client from the burden of performing data validation.

Best Practice 29.12 Accept client data in the most convenient format for the client

When the client's view of the data is different from the object's internal representation, always make public interfaces work in the client's format.

Best Practice 29.13 Accessors should not change class state

Recall that mutators change class state whereas accessors observe class state. It is possible to have a method which does both, of course. However, if a method is specifically designed to be an accessor, then it should not change state.

Best Practice 29.14 Use asserts in accessors to validate class state

Assert that the class attributes are in a valid state before returning data to the client. This provides an additional assurance that the method is always returning valid data.

accessors brings the developer one step closer to finding bugs.

Best Practice 29.15 Give the client data that are well-formed and in the most convenient format

Though it is most convenient for the object to return the class state in the form of the internal representation, a class author should be more concerned about what is most convenient for the client.

Chapter 31 Metric: Adaptability

Best Practice 31.1 Avoid overengineering

Designs should reflect what is needed to meet the stakeholder's needs and reliably anticipated changes/augmentation. Do not add functionality when there is no readily apparent need.

Best Practice 31.2 Avoid universalization

Prefer abstraction over universalization.

Best Practice 31.3 Be very careful about changing a base class

The further up the inheritance tree a class resides, the more difficult it becomes to validate it. Therefore, be very careful about making any changes to a base class.

Chapter 33 Metric: Redundancy

Best Practice 33.1 Percolate variants down in the inheritance hierarchy

If a property or attribute is unique to a single class, it should exist only in that derived class. Such things do not belong in a base class.

Best Practice 33.2 Percolate invariants up the inheritance hierarchy

The base class should include all the properties and attributes that are common among the derived classes beneath it. If two derived classes contain the same attribute or property, consider moving it to the common base class.

Chapter 34 Quality: Debugger

Best Practice 34.1 Memorize the shortcut keys to each of the stepping features

The more fluid you are in using your debugger, the more you will be able to focus on the code and less on the tool. Memorizing the shortcut keys will make you a more efficient programmer.

Best Practice 34.2 Use "run" when you expect the program to crash

Because debuggers can catch program crashes, you can easily move to the location of the bug by selecting the "run" command in the debugger.

Best Practice 34.3 Use "run" when you want to verify the normal operation of the program

When debugging a program, it is often the case when we have isolated the bug to a few lines of code. After you have stepped over those lines, select the "run" command to see if other bugs exist on the other side.

Best Practice 34.4 Use step over to better understand how a single function works

The step over feature in a debugger keeps execution in a single function, stepping over the other subroutines. This helps the programmer focus on a single function without getting distracted by unimportant details.

Best Practice 34.5 Use step into to view every single statement as it is executed, skipping nothing

If you are unsure of how the code works or do not want to miss any detail when debugging a program, the step into feature will enable you to do this.

Best Practice 34.6 Use step out when you want to return to the caller

Whenever we want to return to a caller, the step out feature will send execution to that location.

Best Practice 34.7 Use run to in a loop to quickly see what has changed with each iteration

With the cursor set on a loop, run to is an easy way to see how the loop behaves. When the loop body consists of a dozen or more lines of code, then step over can be very tedious. Repeated run to commands will pause once per loop iteration.

Best Practice 34.8 To quickly skip over a block of code or even a loop, use run to

Usually only a few statements are relevant when fixing a bug. Run to makes it easy to skip to these statements without getting bogged down in irrelevant details.

Best Practice 34.9 Use set next statement to explore “what-if” scenarios

Set next statement gives the programmer great power to see how code will behave in a wide variety of situations without having to stop and recompile the code.

Best Practice 34.10 Set breakpoints at all the points of interest before executing the code

When debugging an object-oriented program, it is often the case that the bug can be localized to a single class. The problem is that you don't know who is using the class or what methods are being called. To discover these things, you can set breakpoints at the beginning of all the public methods in the class. When the program is executed, this will tell you exactly who is using the class and in what way.

Best Practice 34.11 Save valuable breakpoint sets

Most debuggers allow programmers to name breakpoints, disable them, and save them. If you have just created a particularly valuable breakpoint set, take a moment to preserve it.

Best Practice 34.12 Use the call stack in conjunction with the watch widow for preliminary debugging investigation

When a program crashes, when an assert fires, or when a breakpoint is hit, the debugger halts execution at the offending line of code. Usually the first thing the programmer does is to check the watch window to see the state of the local variables.

Best Practice 34.13 Use the disassembly view to see exactly how a given programming construct is compiled

With high-level programming languages, it can be difficult to see how much assembly code is generated from a seemingly innocuous line of code. Disassembly reveals such details.

Chapter 35 Strategy: Polymorphism

Best Practice 35.1 All members of a polymorphic collection must derive from the same base class

The first step in every polymorphic design scenario is to create an appropriate inheritance tree. The definition of the base class is critical because it will define the shared interfaces which all derived classes will need to implement.

Best Practice 35.2 Every derived class should be able to stand in for its base class

If an object is made from a base class, then another object made from a derived class should work in the application.

Best Practice 35.3 A derived class should expect no more of the client than the base class

Ideally, the preconditions of the derived class should be looser than that of the base class. This means it can accept a wider range of input. In other words, the preconditions of the derived class should constitute a superset of those of the base class.

Best Practice 35.4 A derived class should deliver no less to the client than the base class

It means that any method the base class provides, the derived class provides it as well. In other words, disinheritance (when a derived class explicitly does not implement a method of the base class or decides to not inherit it) is less than ideal.

Best Practice 35.5 Make sure the interfaces in the base class are enough

The interfaces in a base class should be enough for all the derived classes and meaningful for all the derived classes.

Best Practice 35.6 Extend a base class, but do not change it

We should work to make base classes available to adaptation in a derived class but seek to avoid modifying the base class itself. As a rule, the higher up a class is in the inheritance hierarchy, the more careful we should be in making changes.

Best Practice 35.7 Design for extension in the face of uncertainty

When you know that the client's requirements are in flux or are poorly understood, then design for extension. When you know that there are many implementation details which are not yet worked out, then design for extension.

Best Practice 35.8 Avoid unnecessary abstractions when the project is well understood

When the client's requirements are well understood, when many of the implementation details are known and there seems to be little left to discover, then do not go through the extra work of making the class unnecessarily versatile and generic.

Best Practice 35.9 Most derived class methods should utilize reimplemention inheritance

In most polymorphic scenarios, derived class methods should use reimplemention inheritance rather than fall through, extension, or disinheritance. They should use the same interface as provided by the base class but an implementation appropriate to that which the derived class is designed to represent

Best Practice 35.10 Fall through inheritance should be used to minimize redundancy

In many cases, the base class or a parent class in the inheritance hierarchy provides an implementation enough for one or more derived classes. This is appropriate, especially in scenarios when multiple distinct derived classes have common characteristics.

Best Practice 35.11 Use protocol extension only for private methods and in restricted circumstances

Protocol extension is useful when a private method is added to a derived class to simplify its implementation. When a public method is added, then there is a good chance that something went amiss.

Best Practice 35.12 Avoid disinheritance in all polymorphic scenarios

The most dangerous type of inheritance relation in polymorphic scenarios is disinheritance.

Chapter 36 Strategy: Is-a and Has-a

Best Practice 36.1 Use a nested class when the class only makes sense in the context of the enclosing class

The modularization concept of coupling captures the notion of scope minimization: that programs are less complex when the lifespan of variables and the flow of information is tightly controlled. The same also applies to functions and classes. If a function should only be used within a class, it should be private. If a class should only be used in conjunction with another class, it should be nested.

Best Practice 36.2 Make a nested class private if only the enclosing class uses it

When a nested class is used as a utility to support the enclosing class, then it probably has no use outside the enclosing class. In situations like these, make the nested class private. Note that if the nested class could be used by others besides the enclosing class, then we must question why the class is nested at all.

Best Practice 36.3 Make a nested class public if it supports the mission of the enclosed class to the client

Consider the iterator, a class enabling the client to iterate through all the members of a collection. Iterators only make sense in the context of the collection they work with. It is therefore a common practice to make iterators a public nested class of the container. If we made the iterator a standalone class, it would be incomplete by itself. This is because it only makes sense in the context of the container it is meant to serve. A better design is to nest the iterator in the container class.

Best Practice 36.4 Use composition in conjunction with nesting when the nested class is part of the enclosing class's identity

It is common for a nested class to also be a composition class. In other words, a nested class can be utilized as a member variable in a class. In this case, *has-a* would be an appropriate description.

Best Practice 36.5 Composition siblings should have no knowledge of each other

As a rule, composition siblings should have no knowledge of each other. This means they demonstrate no knowledge of each other's implementation details and do not contain references of each other. To do so would violate the encapsulation principle of abstraction.

Best Practice 36.6 Treat composition siblings as disjoint classes

The principle of encapsulation states that classes should be as independent from the rest of the program as possible. In other words, the client should be shielded from any implementation details and be presented with a convenient interface. This is principle applies equally to when the client is a function utilizing two classes for local variables or when the client is a class utilizing two classes for member variables. The principle is the same.

Best Practice 36.7 Do not force *is-a* or *has-a* relations; keep them closely aligned with design concerns

Whenever our class design strays from the design concern, unintended consequences such as these seem to follow.

Best Practice 36.8 When more than one class must share the same interface, use polymorphism

Any programming problem where more than one class must share the same interface is a good candidate of polymorphism and, by extension, inheritance.

Best Practice 36.9 Class relation designs should strive to achieve complete alignment

In most cases, one design will align better with design concerns than another. Usually it is not clear which is better until a draft of each design is sketched out. In times like these, it is a good idea to consult with the subject matter expert. Oftentimes, he or she can offer insight as to which design better models the system. Designs better aligned to the design concerns typically require less refactoring when the inevitable alterations are required.

Best Practice 36.10 Prefer composition over inheritance

In those cases when no clear design is better, prefer *has-a* over *is-a*. Most software developers find composition easier to understand and bugs are easier to find. From a maintainability standpoint, composition is superior to inheritance.

Chapter 37 Strategy: Object Creation

Best Practice 37.1 Use simple object creation when there is only one variation

Use simple object creation when there is only one variation of an object to be created. In cases like this, abstracting or encapsulating the object creation process serves no purpose other than adding complexity to an otherwise simple process.

Best Practice 37.2 Use simple object creation when the class is not part of a polymorphic set

Instantiating an object that is a member of a polymorphic set often requires a more complex object creation strategy. For most other cases, namely for classes that are not part of an inheritance hierarchy, then simple object creation is the best strategy.

Best Practice 37.3 Use a factory function when the client needs to select from a variety of different classes

When there are multiple versions of a class that the client may wish to select and when those versions can be easily enumerated, then the factory function design pattern is probably the best option.

Best Practice 37.4 Use a factory method when the clients of a class are restricted to a family of classes

When all the clients of a given class exist in a single inheritance tree, then the factory function should be a member function within that tree.

Best Practice 37.5 Use an abstract factory when the client requires several variations of a factory

To be truly useful, the client needs to have several variations of a factory representing distinct categories of objects to be created.

Best Practice 37.6 Use a builder when object creation can be subdivided into several components or phases

When object creation is a multi-step process, then the builder design pattern is a good fit. This is true even when there are just two or three steps.

Best Practice 37.7 Whenever a client needs to copy an object, provide prototype functionality

Even when a class is trivial and utilizes simple member variables, we should provide a `clone()` method. This way, if the class needs to change to have more complex functionality, then no client code needs to change.

Best Practice 37.8 Use the assignment operator and copy constructor if your programming language supports it

Many programming languages allow a class to define the assignment operator and the copy constructor. If this is true with your language, use it to include the clone functionality.

Best Practice 37.9 Use a singleton for logging

Many applications need to contain logging functionality so transactions can be audited, or events can be reconstructed. Since logging code might need to be added in almost any function or method in a program, the scope of a logging variable needs to be immense. Rather than pass a logging instance to every class and function in the application, the coupling of the program can be reduced with a singleton.

Best Practice 37.10 Use a singleton for external interfaces

While there certainly are other concurrency mechanisms which offer similar assurances (such as semaphores and mutexes), singletons are often part of the solution.

Best Practice 37.11 Use a singleton to remove a global variable

Global variables are harmful programming constructs. While singletons have global scope like that of a global variable, they have an advantage – it is easy to add getters and setters which facilitate answering these two critical questions.

Chapter 38 Strategy: Algorithm Abstraction

Best Practice 38.1 Use a factory to instantiate a concrete strategy to reduce coupling

The tight coupling between the context and the strategy classes can be reduced through the use of a factory method.

Best Practice 38.2 Use the strategy design pattern when the algorithm variation is selected at runtime

The strategy design pattern combined with a factory makes it possible to select the algorithm variation at runtime.

Best Practice 38.3 Use the template method pattern when an algorithm has several changeable steps

When an algorithm has just one component that varies, then the strategy design pattern is a better choice. However, when there are several steps which can be defined at various levels in an inheritance hierarchy, then the template method is a better fit.

Best Practice 38.4 Avoid redundancy by percolating common tasks higher in the inheritance hierarchy

Template methods often have deep inheritance hierarchies, allowing common steps to be used by many concrete classes. Place common steps high in the inheritance hierarchy and unique steps low in the inheritance hierarchy.

Best Practice 38.4 Verify your decorator implementation early and often

When it is possible to create long and complex task chains such as these, it is a good idea to write unit tests to verify it works as it should. Task chain bugs can be difficult to find and even more difficult to fix. They are much easier to find and fix early in the development process.

Chapter 39 Strategy: State

Best Practice 39.1 State labels are adjectives, nouns, or progressive verbs

Because a state can represent a configuration of the system, it can have an adjective label. Here, the label should refer to the name of the configuration. A state can also represent a noun. A car can be in first gear, so “first” is an appropriate label for a transmission state. Finally, a state can represent a status of the system or what the system is doing. These are best captured with a progressive verb label.

Best Practice 39.2 Do not label a state with an action verb

Be cautious when a state is given an action verb name. Action verbs usually refer to a process and are more appropriate for function or method names. When designers start using action verb labels, their state diagrams begin to resemble flowcharts or structure charts.

Best Practice 39.3 Use an initial state symbol only when the system starts in a given state

When an initial state has a label, it is usually because it indicates a connection with another state diagram. Similarly, a state diagram may use labeled initial and final states to connect long transitions to reduce clutter in a state diagram.

Best Practice 39.4 Use composite states to add clarity

A composite state should be used to group a collection of states when the group has semantic meaning in the context of the application domain. By grouping these states into composite states, it becomes clearer what these substates represent and how they are related.

Best Practice 39.5 Composite states can be used to subdivide large state machines into multiple state diagrams

Though most state diagrams consist of less than a dozen states, there are some that may contain several dozen. These quickly become too busy to be easily understood. Composite states can address this problem by allowing the designer to describe subsystems in separate state diagrams.

Best Practice 39.6 States tend to be located where the system pauses

Though it is possible for a meaningful state to exist where the system remains for only an instant, most states capture a condition in which the system dwells.

Best Practice 39.7 States should represent unique conditions of the system

When a system has multiple conditions under which the system behaves essentially the same, then one state is probably better to model the conditions than several.

Best Practice 39.8 Avoid combining unrelated state machines

If you notice that two sides of a state diagram look very similar, then there might be some redundancy. This is typically because the design would be better served with two independent state machines rather than one.

Best Practice 39.9 Delineate orthogonal regions with composite states

It can be the case that the existence of orthogonal regions is a desirable characteristic of a state diagram. This occurs when there should be no overlap between subsystems. In cases such as these, it is usually a good idea to encapsulate the subsystems in a composite state.

Best Practice 39.10 Verify that orthogonal regions are desirable properties of the state design

Usually, the existence of orthogonal regions is an indication of a defect in the design. If a program has a set of states which cannot be reached from the initial state, then why are these states in the design at all?

Best Practice 39.11 Use an enumeration to name your states

If your programming language supports enumerations, use them to name your states. This makes the code easier to read and more difficult to confuse states.

Chapter 40 Tool: Component Diagram

Best Practice 40.1 Maximize reusability

Design components to be as useful as possible to other parts of the system. This may mean making interfaces more complete or generic than is strictly necessary for the system we are building.

Best Practice 40.2 Make components replaceable

Remember, components to be reused and replaced. You must ask yourself, if someone were to swap out this component with another, how difficult would that be?

Best Practice 40.3 Maximize cohesion

Strive to make every component do one thing and one thing only. This relates directly to the concepts of cohesion for modularization and fidelity for encapsulation. As with

functions and classes, if you cannot come up with a good name for your component, then you probably have a cohesion problem.

Best Practice 40.4 Minimize coupling

Strive to make the interfaces as simple as possible between components and with the outside world. A well-designed interface does not betray any implementation details of the component, is easy for another component to honor, and is easy to extend.

Best Practice 40.5 Enclose appropriately

If several components have somewhat tight coupling between themselves and reveal more about the inner workings of the system than you would prefer, consider enclosing them in a larger component. The goal here, of course, is to minimize system complexity, not to add a new level of complexity to the system.

Best Practice 40.6 Consider compatibility

All changes to interfaces must be backwards compatible. This means that old components should work with new versions of the interface. Most changes to interfaces should be forward compatible as well. This means they should allow for extension and elaboration to accommodate future functionality that one may wish to add to the system. This greatly decreases maintenance costs of the system.

Chapter 41 Tool: Design Description

Best Practice 41.1 Start with the front matter

Before one begins any project (large or small), one needs to understand the scope and what is needed. Many skip this step, thinking they know the project well enough, only later to discover that there is a miscommunication or misunderstanding. Taking a few minutes to write down your understanding of the project can clarify things and reduce the opportunity for mistakes.

Best Practice 41.2 With every major alteration to the front matter, review the entire design description

As often happens on medium- to large-scale projects, we discover that mistakes were made. Sometimes we realize that our understanding of the scope of the project is different than the client's understanding of the scope. As one updates the front matter to reflect the latest understanding of the project, it is a best practice to review the rest of the document to identify places where these misconceptions may be manifest.

Best Practice 41.3 Involve the entire team in the first few views of the design

The first few views in the design description are usually created collaboratively. The manager or head designer calls a meeting with the entire team and different ideas are hashed out on a white board. Here it is important to encourage the team members to share their ideas and concerns. It might take an hour or two to work through all the options and arrive at a final decision.

Best Practice 41.4 The entire team should fully understand and have buy-in on the first several views

After the team has debated many possible design possibilities, it is important that everyone is unified behind a single design. This means that many members of the

team might need to forget about discarded design alternatives—even the ones that had desirable properties. It is more important that everyone is on the same page than that the team is following the best possible course.

Best Practice 41.5 Use a component diagram or a DFD for View #1

The first view of a design description needs to be the most general view, capable of capturing the overall system architecture. In most cases, this is a component diagram, though some designs are best described with a DFD. Here, the design team takes special care to identify the parts of the system which are mostly independent of each other.

Best Practice 41.6 Carefully negotiate interfaces between components

If the interfaces between components are well understood, then the components themselves can be designed and built independent of the rest of the system. This serves to reduce the complexity of the overall system and simplify testing. Unfortunately, identifying and negotiating interfaces can be a very challenging task.

Best Practice 41.7 Use the design information table to record everything that is known about a given view

The most important part of the design information table is the enumeration of design elements described in the design representation. At all points in the design process, the design team records all that is known about each element in the elements row of the table.

Best Practice 41.8 Use a multilevel numbering system to name views

When the number of views is greater than a couple dozen, it is useful to use a multilevel numbering system. This ensures that related views are next to each other in the large document. Another benefit is that each component can add or remove views without impacting the number of other views in the document.

Best Practice 41.10 Use headings and links intentionally

A design description is not read linearly as one would read a novel. Instead, one constantly jumps between views to get more information or to understand the context in which a design element resides. Most word processors will bookmark headings and create a document map to facilitate intra-document navigation. Leveraging these navigation tools is essential to making large documents such as a design description readable.

Chapter 42 Quality:

Best Practice 42.1 Contract, regulation, and operational requirements should use alpha testing

Alpha testing is particularly effective for CAT because it often takes a trained QA engineer to decipher the language used to describe contractual requirements. Subtle differences in word selection can determine whether the contract is satisfied or whether it is not. Alpha testing is also effective in RAT because legally stipulated regulations can be complex and have subtle implications. Finally, alpha testing is effective in OAT. Many platforms are very complicated, requiring extensive domain knowledge and experience to understand their idiosyncrasies.

Best Practice 42.2 Acceptance testing of user requirements should have an alpha testing component

Alpha testing can and should be used for user requirements as well. Unlike CAT, RAT, and OAT where the challenge is specifying tester requirements, the challenge with UAT alpha testing plans is to list the test cases necessary to adequately verify the system. User stories and use cases can be useful tools in this regard.

Best Practice 42.3 Building a network of beta testers should be an ongoing process

Because beta testers are such a valuable acceptance test asset, the QA team should constantly be working to fill the ranks of potential beta testers. Recruitment efforts should be made at each customer, user, and stakeholder interaction

Chapter 43 Strategy: Large Inheritance Trees

Best Practice: 43.1 Every base class should map to a design concern

Software developers create classes to represent design concerns. The challenge is to create meaningful divisions, so real distinctions and similarities are captured in the inheritance design.

Best Practice 43.2 Do not worry about balancing an inheritance tree

When designing an inheritance hierarchy, put no thought into how balanced the tree will be. It does not matter if one branch contains a hundred derived classes and another contains only two. What does matter is that each division is meaningful.

Best Practice 43.3 Each level in an inheritance hierarchy should mean something

A well-designed inheritance tree should seek to make level distinctions consistent across branches and map to design concerns.

Best Practice 43.4 Carefully place new derived classes in the inheritance tree

When the inevitable addition needs to be made to an existing inheritance tree, great care needs to be taken to place the new class in the right spot. While it might be easiest to just throw it into the most convenient spot, this could result in illogical combinations that might be difficult to remedy in the future. Be wary of this temptation! It will almost always create more work for the entire team.

Best Practice 43.5 When new understandings invalidate old assumptions, move derived classes as needed

As changing software requirements are revealed or implementation details are discovered, it often becomes apparent that assumptions are incorrect under which a given inheritance tree was built. When this happens, refactor the hierarchy so it is better aligned to the design concerns.

Best Practice 43.5 Refactor the base classes when needed

Though we seek to avoid modifying base classes, it occasionally becomes apparent that a given design was made without a complete understanding of the system the inheritance tree was meant to model. In cases such as these, it is better to refactor a base class than to tolerate a poorly aligned inheritance tree.

Best Practice 43.6 The higher the rank of the change, the greater the caution that should be exercised

When refactoring a base class containing a small number of derived classes, a certain amount of caution should be exercised. However, when refactoring a base class high

in the inheritance tree, great care should be taken. As a rule, an order of magnitude more caution should be exercised for each level up the inheritance hierarchy.

Best Practice 43.7 Keep the depth of the inheritance tree reasonable

While computers can deal with huge inheritance trees, humans cannot. An upper limit as to the practical depth of a class inheritance hierarchy is about seven. Most applications should be no deeper than three or four.

Best Practice 43.8 Unambiguously communicate the meaning of each division in an inheritance tree

Every single node in the inheritance tree should be unambiguously defined and communicated to all who use it.

Best Practice 43.9 Choose base class names carefully

When choosing a name for a base class, make sure it honors any established naming conventions in your organization and make sure it captures the defining characteristic of the group of derived classes it is meant to represent.

Chapter 46 Strategy: Command Passing

Best Practice 46.1 Use direct invocation when there are a small number of orders

If there are a small number of orders for the executor to carry out, then direct invocation can be implemented with minimal lines of code. This makes the direct invocation design pattern ideal for small and simple problems.

Best Practice 46.2 Use direct invocation when quality assurance is an overriding concern in the system design

Direct invocation has the benefit of allowing for a relatively small test burden. Adding new connections has minimum impact on the rest of the system. If the new connection does not interfere with an existing one, the other systems do not need to be verified when a new connection is added.

Best Practice 46.3 Avoid direct invocation when it is likely that new interfaces will need to be added to the design

While it is easy to create a single connection between the commander and the executor using direct invocation, the second connection costs as much as the first. Thus, adding capabilities to the system can be prohibitively expensive.

Best Practice 46.4 Avoid direct invocation when system-wide features might need to be added to the design

Adding system-wide services such as security (to prevent eavesdropping) or logging (recording all the orders relayed to the executor) can be complex and time consuming. Direct invocation requires each connection to be independently studied and designed. When the size of the API is large, this can be prohibitively expensive.

Best Practice 46.5 Maximize uniformity when designing a direct invocation API

Most of the disadvantages of direct invocation can be mitigated by utilizing consistent API names, behaviors, and data types. Little things such as using the same representation for commonly used constructs goes a long way towards making the interface more maintainable.

Best Practice 46.6 Specify the delegate function signature carefully

The author of the delegate interface needs to be extremely careful when specifying the delegate function signature. The signature must be rich enough to support all the commander's needs. It also must not require any information that the commander is not able to provide.

Best Practice 46.7 The executor should know nothing about the delegate

To decouple the commander from the executor, it is important that the executor makes no assumptions about the commander. The recipient is only responsible for accepting the callback and executing it at the appointed time.

Best Practice 46.8 The commander should know nothing about how the executor will execute the callback

Once the commander hands the executor the callback, it is up to the executor to decide when and how to call the callback. Any assumption the commander makes about the executor serves to increase the coupling between the two components.

Best Practice 46.9 Make it easy to configure a command object to represent the action to be performed

Unlike delegate invocation where the commander simply selects the delegate to represent the action that needs to be performed, command objects may contain the data that accompanies the order. Concrete command authors should create constructor and mutators to make this process as easy as possible. The convenience and fidelity metrics of encapsulation design should be carefully considered.

Best Practice 46.10 If commander context is needed, add it to the command object rather than having the executor manage it

It is often the case that an order needs information from the commander to function properly. Using delegate invocation, this can only happen by asking the executor to pass commander context onto the order. The encapsulate invocation design pattern can make this easier on the executor. Because classes have member variables, it is easy to add any needed commander context to the command object.

Best Practice 46.11 The executor should not need to know anything about the command object it invokes

Aside from the name of the abstract command base class and the name of the execute method, the executor should not need to know anything about the commands that are executed. This serves to give the commander great latitude in the type of commands which are created. It also gives the executor latitude to implement a variety of order invocation strategies such as queuing and logging.

Best Practice 46.12 Carefully design the context to facilitate make the expression classes easier to implement

The commander needs to work with the context to create an order matching the system needs. The executor needs to work with the context to understand the actions contained therein. If you choose an overly general context (such as a string), then it may be very difficult to parse.

Best Practice 46.13 Use a data interchange format to encode the document format

While it is possible to use virtually any file format to encode commands, not all formats parse equally easily. There are many data interchange formats that can make this job much easier. Formats such as XML and JSON are supported by numerous libraries in most languages.

Best Practice 46.14 Mitigate against command and script injection attacks

When designing a system to use the interpreter design pattern, be aware of command and script injection attacks. Make sure that malicious users cannot abuse the power that is contained in this interface.

Chapter 47 Strategy: Interfaces

Best Practice 47.1 Be wary of performance problems in adapter methods

When the client uses adapter interfaces, there is an expectation that things will run as smoothly as if the provider interface was directly queried. Therefore, be extra cautious of performance problems in adapter interfaces. One poorly written adapter interface doing more work than necessary can bring down the performance of the entire system.

Best Practice 47.2 Use a bridge pattern when there are multiple clients or multiple providers

The bridge pattern eliminates any duplicate code when there are several clients requiring different interfaces or when there are many providers with different interfaces. This design has the distinct level of redundancy.

Best Practice 47.3 Use the bridge pattern when you can control the implementation interface

Many situations involve integrating components with predetermined interfaces. The adapter is often the best solution in those cases. However, when the provider interface is negotiable, then the bridge pattern often yields more malleable and efficient solutions.

Best Practice 47.4 Use the façade pattern when the underlying provider is large or complex

Though façades can be vast and difficult to create, they can shield the client from having to understand complex subsystems. In times like these, a façade can greatly increase the convenience of an otherwise prohibitive API.

Chapter 48 Strategy: Tokens and Entities

Best Practice 48.1 Each token should be the same size

The size of a token is the amount of memory required to store it on a computer system. A byte token, for example, would be eight bits in size. When each token on the system is the same size, it is easier to manage entity collections.

Best Practice 48.2 Token should be easy to move, duplicate, and delete

The caretaker should be able to easily manipulate the tokens it stores. This means that common operations such as moving, duplicating, and deleting should be fast and easy to accomplish. When a token is a built-in data type such as an integer, this is easy to do. When something more complicated is used, then work should be done to ensure that it is efficient and easy to move, duplicate, and delete the token.

Best Practice 48.3 Each token should map to a single entity

The originator should ensure that tokens are not reused or repurposed. If more than one entity corresponds to a single token, then it can difficult or impossible to determine what the token stands for. While it may be unavoidable in some situations, care should be taken to minimize the instances when a single token maps to more than one entity.

Best Practice 48.4 Each entity should map to a single token

It is desirable but not required that the entire problem domain be represented in the set of possible tokens. This may not be possible when there are an infinite number of entities. However, when each entity has exactly one corresponding token, then it becomes possible for the system to determine if two entities are the same through comparison of their tokens.

Best Practice 48.5 The caretaker must be blind to the contents of the token

The caretaker must not depend on the implementation details of any aspect of the token except for its size. In other words, if the originator chose to change the implementation of the token, the caretaker should not require any design alterations to accommodate this change.

Best Practice 48.6 Only one component should handle translation between a token and an entity

The goal here is to minimize or eliminate redundancy in the system. If any change is made in the entity representation strategy of the system, then only one component should have to change.

Best Practice 48.7 Do not use incremental counters if tokens need to keep entities confidential

The value of a token may reveal details of the entity that the owner may wish to keep private. A malicious user eavesdropping on the caretaker can reconstruct the sequence of entities enrolled into the system simply by comparing token values. A malicious user can also determine the total number of entities in the system, a fact the owner may wish to keep confidential. Also, a malicious user could retrieve adjacent entities in the entity table by incrementing or decrementing a token value.

Best Practice 48.8 Randomly generate token keys

In situations where token keys travel over networks, are persisted in files, or are publicly accessible in any way, token keys should be randomly generated. This makes it more difficult for an eavesdropper to deduce anything about the entity based solely from the token; the originator's entity table is required to make the translation.

Best Practice 48.9 Avoid packing token keys too closely together

The more closely packed tokens, the greater the chance that a randomly generated token will map to a real entity. To simplify the process of randomly generating unique token keys and to reduce the chance that a valid token can be guessed, there should be a million or more invalid key values for every valid token key value.

Best Practice 48.10 Use a data interchange format to encode the entity

While it is possible to use virtually any file format to encode an entity, not all formats parse as easily. There are many data interchange formats that can make this job much easier. Formats such as XML and JSON are supported by numerous libraries in most languages.

Best Practice 48.11 Use a configuration file when passing entities between systems

When an entity needs to pass between components, a configuration file is natural choice. Because thick tokens are self-contained, they can be shared between caretakers without needing originators to mediate the exchange. If the receiving caretaker understands the configuration file format, then sending the configuration file to the recipient is all that is needed to complete the exchange. This is easiest when the configuration file format is standardized or at least well documented.

Chapter 49 Strategy: Layered System Design

Best Practice 49.1 Leverage existing client services when designing a thin client

The functionality and, in some cases, the look and feel of a thin client application can be defined by the existing services on the client. Care should be taken early in the design process to fully understand these capabilities.

Best Practice 49.2 Be cautious of server load

A client's data footprint must be carefully optimized, and the computational cost of serving a single client request must be carefully optimized.

Best Practice 49.3 Utilize thick client to optimize user experience

In most computing environments, the network is the single largest performance bottleneck. Applications requiring large network throughput or low network latency can feel unresponsive in many scenarios. These interruptions can be more easily controlled when the network traffic is minimized using a thick client architecture.

Best Practice 49.4 Utilize thick client to minimize server cost

Per unit cost is the additional load on the server for each new client. This cost can take many forms, but the most common are storage and computing costs. If the number of clients is expected to be very large, then an effort should be made to place as much computational and storage load on the client as possible. This serves to minimize server load, making it possible to accept large increases in the number of clients without requiring substantially more server capability.

Best Practice 49.5 Prefer thin client over thick to support cloud architecture

A key aspect of cloud services is the ability to add and remove functionality according to the business needs of the customer. Complex and time-consuming client installation procedures can make it more difficult to support the subscription model. When faced with multiple technological options, favor thinner clients.

Best Practice 49.6 Manage subscription services carefully

A key component of all cloud services is to provide the customer with the desired level of service in an environment when the level of service can be in flux. Thus, each

request for service must be verified against the level of service to which the customer is currently enrolled.

Best Practice 49.7 Carefully consider interface changes

Interfaces between layers should be carefully considered. These should reflect the task the layer is to perform, making no reference to how the task is to be performed. In other words, a layer should be a “black box” to the layer above it, revealing nothing about how it performs its action.

Best Practice 49.8 One task should not cross a component boundary

Each task should be completely assumed by a single component. If a task crosses a component boundary, then perhaps the boundary should be dissolved. Note that it is acceptable for one component to utilize the resources of a lower-level component to complete a task. That is not a problem. However, if the task itself spans multiple components, then the overall system design should be reanalyzed.

Best Practice 49.9 Each component should only communicate with adjacent components

Recall that a defining characteristic of the *n*-tier design is that each layer can only communicate with the layers above it and below it on the stack. If this constraint is violated, then a layer would be made aware of specific implementation details of neighboring layers.

Best Practice 49.10 Use common and recognizable layer names

Though an *n*-tier application can have any number of tiers performing any number of services, most designs select from a small number of services.

Index

Appendix D

2

2-to-the-*n* 61, 339

A

absorption 159
abstract component 758
abstract factory 727, 733
abstract strategy 752
abstract template 755
abstraction 10, 477, 518, 560, 730, 757
acceptance test 493, 835, 840
acceptance test plan 842
accessor *See* getter
activity diagram 26
ad hoc driver 316, 317
adaptability 10, 615, 736
adapter 943, 944
add 569
ad-hoc 1002
adjustable 97, 100
aggregation 543, 595, 598
agile 549, 814, 824
algorithm 3, 14
algorithm abstraction 751
algorithmic efficiency 61
alignment 10, 636
alpha tester 843
alpha testing 843
ambassador *See* remote proxy
and 159
annulment 159
anonymous 180
API *See* application programming interface
applicant 971
application programming interface 915, 952
architect 1
array 178, 181, 694
assembly code 3
assembly connection 802
assembly connector 800
assembly language 24
assert 11, 114, 581, 584, 679, 1063
association 6, 543, 595, 600
associative 159
associative array *See* dictionary
attribute 5, 390, 414, 560
authentication 971

automation 316, 320, 492, 494

B

back end *See server*
back matter 818
backtrack 19
BAOE 304, 1050
base class 6, 855
BDD *See behavior driven development*
behavior-driven development *See test-driven development*
beta testing 843
Big-O 61
binding 692, 698
binding table 692
 data type 693
 function 693, 696
 variable 692
black box test 493
black box testing 443
block 179
bookmark 974
Boolean algebra 159
Boolean expression 158, 159, 199
bottom-up development 369, 372, 1052
boundary condition 304
breakpoint 668, 677, 684
 conditional 678, 682, 684
 notification 678, 683
bridge 943, 947
builder 727, 735
business logic 895, 898
by-reference 285, 391
by-value 285

C

cache *See remove proxy*
call stack 337, 338, 392, 668, 679, 684
callback 694, 697, 919
callee 226
caller 226
can 564
caretaker 966, 968
CAT *See contract acceptance test*
chain of responsibility 868, 873
checkpoint 974
circuit diagram 817
circular integration 903
claimant 971
class 5, 179, 390
class cohesion 431
class diagram 9, 225, 408, 594, 817
class under test 444, 446
classifier rectangle 801

client.....	356, 461, 578, 755, 944, 969, 989, 1001
client-server.....	988, 989, 995
closed	615, 616
cloud.....	994
code coverage.....	<i>See</i> test coverage
code review.....	442
codepath	499
cohesion.....	10, 261, 278, 426, 431, 542, 641, 657, 801, 806, 896, 1069
coincidental.....	266
collaborator.....	500, 547, 548
collection-controlled loop	198, 208, 1048
collections	178
command design pattern.....	<i>See</i> delegate invocation
command injection.....	932
commandeer driver	316, 318
commander.....	914
common	286
communicational	266
commutative.....	159
compile time	691
compile time binding.....	<i>See</i> early binding
complement.....	159
complete	426, 427, 477, 478, 636, 637, 757
complex.....	278, 282, 352
component.....	7, 798, 800, 801, 837
component architecture design	837
component diagram	9, 225, 227, 798, 817
composing functions	<i>See</i> high-order function
composite	<i>See</i> composite state
composite state	778
composition	6, 543, 595, 597, 710
composition siblings	713
concrete component	758
concrete decorator	758
concrete strategy	752
concrete template	755
conductor	1002
configurable	97, 98
constant	61, 63, 79
constructor.....	580
context	752
continuous	771
contract acceptance test.....	842
control	286
controlling expression	199, 201
convenience.....	10, 461, 518, 521, 560, 584, 585, 624, 730, 899, 953
conversion constructor	579, 583
convoluted	461, 463, 518, 615, 619
cookie	978
copy constructor	579, 581
correctness.....	299
counter	201
counter-controlled loop	198, 201
coupling.....	10, 278, 461, 518, 584, 657, 951

CRC card	548
credentials.....	971
critical	477, 481, 652, 655
customer test.....	<i>See</i> acceptance test

D

data.....	286
data access layer.....	<i>See</i> server
data flow diagram....	9, 225, 241, 242, 409, 540, 541, 578, 676, 799, 817
data hiding	413
data protection	577
data storage	895, 896
data-driven.....	97, 99, 100
De Morgan	159
debugger	11, 668
decision tree	27
decomposition	<i>See</i> top-down design
decorator	751, 758
deducible.....	91, 94
deep copy.....	581
default constructor.....	579, 580
delegate	918, 919
delegate invocation	914, 918
dependency	543, 595, 601
dequeue	973
dereference.....	208
derived	413
derived class.....	6, 699
design concern.....	5, 426, 461, 538, 559, 615, 636, 855, 896, 898
design document	9
design phase	298, 371
desk check.....	<i>See</i> trace
destructor	579
developer	1
development.....	225, 369, 372, 409
DFD	<i>See</i> data flow diagram
dictionary	178, 185
direct invocation	914, 915
direct recursion.....	339
director.....	737
disassembly	668, 681
discrete.....	771
disinheritance	703
disjoint classes	711
display	562
distinct.....	652, 653, 736
distributed system	1003
distributive	159
divide and conquer	<i>See</i> top-down design
document.....	278, 283, 352, 951
domain analysis	549
double	500
double negative	159
driver	11, 315, 380, 444

driver function	316
dumb terminal	<i>See thin client</i>
dummies.....	500, 511
dynamic binding.....	<i>See late binding</i>
dynamic data type	695, 696

E

early binding	179, 180, 692
easy.....	461, 465
efficiency	10, 299, 301
enabling.....	615, 621, 736
encapsulated.....	278, 280, 352
encapsulated invocation	914, 924
encapsulation.....	5, 407
end-condition.....	138, 331, 332
ENIAC.....	988
enqueue	973
enrollment	967
entity	965
entity identification	547
entity relationship diagram.....	9, 225, 817
entity representation	965
entity table	975
entry point	773, 779
equivalence class	303
event-controlled loop.....	198, 199
executor	914
exit point	773, 779
extension.....	6, 703
external	286
external transition	779
extraneous	261, 263, 426, 428, 636, 638, 654, 657, 711, 736

F

façade	288, 462, 943, 950, 1049
port.....	951
single function	951
factory	727, 729, 754
factory design pattern.....	<i>See factory</i>
factory function	<i>See factory</i>
fakes	500, 511
fall through.....	703
false negative	502
false positive	502
fat client	<i>See thick client</i>
fat server	<i>See thin client</i>
fault tolerance.....	299
fidelity.....	10, 426, 642, 736, 801, 896
field testing	<i>See beta testing</i>
filter	395
final state.....	775, 779
find.....	561, 896
finite state machine.....	780

first-order function	393
fixture	498
flare	145
flow	243
flow arrows	19
flow process chart	24
flowchart	9, 15, 43, 225, 242, 409, 817
FOR	43, 49
fork	773, 777
FORTRAN	988
fragile.....	440, 441
front end	<i>See</i> client
front matter	815, 841
FSM.....	<i>See</i> finite state machine
function	4, 179
function pointers	<i>See</i> callback
function signature	923
function under test.....	316, 317
functional	266
functional decomposition	336
functional programming	12, 95, 389, 391
functions under test	521

G

get.....	560, 561, 896
GET	43, 46
getter.....	560, 578, 579, 585
global.....	285, 391
group	1001
guard assert	504

H

hacker	1
has	564
has-a	<i>See</i> composition
head recursion	334
high-order function	393, 919
homogeneity.....	1001
host.....	<i>See</i> server

I

IaaS	<i>See</i> infrastructure-as-a-service
idempotent	159
identity	159
IF	43, 48
immediate	<i>See</i> watch:immediate
immutability	392
imperative	389, 390
IMS.....	<i>See</i> inventory management system
incremental counter	975
index	181, 208
indirect recursion.....	339
infinite loop	203

infix	565
infrastructure-as-a-service	994
inheritance	6, 543, 595, 596, 710
inheritance siblings	714
initial state	773, 775, 779
input	562
insert	561, 896
instantiation	5
instruction pointer	16
instrumentation	74, 500
integration test	373, 493, 839
integration test plan	837, 845
intent-revealing names	92
interact	562
interactive	278, 284, 352
interactor	243
interface	7, 800, 802
provided	800
required	800, 802
interface expansion	619
interfaces		
provided	802
interoperability	299
interpreter	914, 928
invariant	658
inventory management system	965
inverse inequality	159
is	564
<i>is-a</i>	<i>See inheritance</i>
iterate	896
Iterator	211

J

join	773, 777
JSON	700, 951, 953, 978
jump table	169

K

key	976
-----	-------	-----

L

lag	518
lambda	394
lambda calculus	391
late binding	691, 694
LCOM	<i>See class cohesion</i>
linear	61, 67, 80
linear aggregation integration	902
linear composition integration	901
linked list	184, 210
Liskov substitution principle	700, 703
list	178, 184
local	285

local transition	779
logarithmic	61, 65
logging	926
logical	225, 266, 409
log-n.....	61, 65
loop.....	198
loop unrolling.....	503

M

machine code.....	3
maintainability	10, 90, 615, 657
malleability.....	97, 623, 730, 754
manual drivers	316
map.....	396, <i>See dictionary</i>
marker	974
mediator.....	868, 870
member variable.....	178, 180
memento.....	<i>See token</i>
memory dump	146
memory table.....	<i>See trace</i>
meta-metric	300
metaphors.....	559
method under test.....	446, 497, 498, 521
minor	652, 654
mirrored client	993
misleading	91, 95
mocks	500, 511
model view controller	905
modularization.....	4, 261
monolithic architecture.....	988, 995
most recently used	974
move constructor	579, 582, 588
MRU.....	<i>See most recently used</i>
multiple-recursion	339, 1051
multiplicities.....	597
multi-tier	<i>See n-tier</i>
mutator	<i>See setter</i>
mutual recursion	<i>See indirect recursion</i>
MVC.....	<i>See model view controller</i>
mystery guest	503

N

namespace	179
nested class.....	6, 543, 595, 599
n-log-n	61, 69, 70, 82
no direct	286
non-default constructor	<i>See conversion constructor</i>
not	159
noun identification	12, 538
n-squared	61, 71, 81
n-tier	988, 995
NULL	210

O

- OAT *See operational acceptance test*
object 5, 426, 578
object reference 977
object-oriented 5, 389, 390
observer 868, 877
obvious 91, 92
on 563
opaque 477, 479
open-closed principle 702
operand 565
operating system 988
operation 5, 390, 415
operational acceptance test 842
operator 407, 565, 569
operator overloading 565
or 159
orchestrated 1002
order 914
originator 966, 967
orthogonal region 782
OSI reference model 995
outsider 971
over specified tests 499
overengineer 371, 517, 622, 702

P

- PaaS *See platform-as-a-service*
package 179, 413
partial 261, 264, 426, 429, 636, 639
peer 1001
peer group *See group*
peer-to-peer 988, 1001
physical 225, 409
pipeline 162
pipelining 161
platform-as-a-service 994
pointer 209
polymorphic set 699, 728
polymorphism 6, 12, 691, 696, 699, 716
poor 426, 430, 636, 640
porous 477, 480, 518
port 800, 803
precision questioning 370, 467
pre-compiler directives 122
predicate 200, 395, 564
prefix 565
pre-processor directives 121
presentation layer *See client*
primary key 970
private 411, 413, 578
procedural 266
procedural programming *See imperative*

process	225, 409
processor.....	243
program flow	16
programmer.....	1
programming language	42
progress step	331, 332
prohibitive.....	97, 102, 461, 462, 615, 618
prompt	562
PROMPT	47
protected	411, 413
protection proxy	956
protocol.....	7
prototype	727, 738
proven	440, 448
provider.....	461, 944
proxy.....	943, 954
pseudocode.....	9, 42, 225, 242, 409, 817
public.....	411, 413, 578
public interface	895, 899
publisher-subscriber.....	<i>See</i> observer
pure function	391
PUT	43
puzzling	91, 96

Q

quality.....	299
quality assurance engineer	835
queue	922, 973
queuing	922

R

RAT	<i>See</i> regulation acceptance test
reachability.....	782
READ	43, 46
real.....	954
recursion	331, 392, 398, <i>See</i> recursion
recursive.....	350
redo	922
reduce	397
reductionism	<i>See</i> top-down design
redundancy	10, 652, 734, 736
redundant	652, 656
refactor.....	101, 126, 524
refactorable	97, 100, 101
refactoring.....	547
referential transparency.....	389, 391
regulation acceptance test.....	842
reimplementation.....	619, 703
reliability.....	299, 301
remote method invocation	916
remote procedure call.....	916, 951
remote processor	<i>See</i> client
remote proxy	955

remove	561, 896
repetitive control structure.....	<i>See loop</i>
requirement.....	546
requirements elicitation.....	298, 299, 369, 370, 835
resilient.....	440, 446
rich client.....	<i>See thick client</i>
RMI	<i>See remote method invocation</i>
robustness.....	10, 440, 560, 624
routing.....	1001
RPC	<i>See remote procedure call</i>
runtime.....	696
runtime binding	<i>See late binding</i>

S

SaaS	<i>See software-as-a-service</i>
scenario	303
schema	817
scope	179, 1047
scoreboard	147
script injection	932
SDD	<i>See software design description</i>
SDET.....	<i>See software development engineer in test</i>
SDLC.....	<i>See software development life cycle</i>
seamless.....	461, 466
security.....	299
sentinel.....	204
sentinel-controlled loop	198, 204
separation of concerns.....	895
sequence diagram	9, 225, 817
sequential.....	266
server.....	989, 1001
service level agreement	842
service provider	<i>See server</i>
service requester	<i>See client</i>
set.....	560
setter	560, 578, 579, 584
shallow copy	581
shortest path first	781, 782
side effect.....	93, 95, 389, 398
signature	230
simple	278, 281, 352
simple object creation.....	727
simplification.....	159
single responsibility principle.....	431
singleton.....	727, 739
size.....	561, 896
SLA.....	<i>See service level agreement</i>
SME.....	<i>See subject-matter expert</i>
SoC.....	<i>See separation of concerns</i>
software crisis	517
software design description.....	814, 836, 841
software development engineer in test	493, 836
software development life cycle.....	834
software requirement	<i>See requirement</i>

software requirements specification	370, 814, 835
software-as-a-service	994
SOLID	431, 700, 702, 1038
SOP	<i>See</i> standard operating procedure
specification-based test	<i>See</i> system test
SPF	<i>See</i> shortest path first
spies	500, 511
SRP	<i>See</i> single responsibility principle
SRS	<i>See</i> software requirements specification
stack	996
stakeholder	30
stamp	286
standard operating procedure	92
start-condition	137
state	771, 773, 774
state chart	<i>See</i> state diagram
state diagram	772, 817
state machine	9, 772
state space	771
state transition	776
state transition diagram	<i>See</i> state diagram
state-chart diagram	<i>See</i> state diagram
static	411
static binding	<i>See</i> early binding
step	668, 675, 755
run	669
run to	669, 674
set next statement	669, 674
step into	669, 671, 674, 682
step out	669, 672, 674
step over	669, 670, 674, 682
step-wise design	<i>See</i> top-down design
step-wise refinement	<i>See</i> top-down design
stereotype	804, <i>See</i> template
storage	243
straightforward	91, 93, 461, 464, 615, 620, 657
strategy	751, 752
strategy design pattern	<i>See</i> strategy
strong	261, 262, 440, 445
structure	178
structure chart	9, 224, 225, 242, 409, 799, 817
structure-based test	<i>See</i> system test
structured programming	<i>See</i> top-down design, <i>See</i> imperative
stub	376
stubs	500, 511
subject-matter expert	636, 637, 639
subscriber	971, 972
substate	779
superfluous	285, 352, 391
supportability	299
swimlane flowchart	25
SWITCH/CASE	43, 48
symbol table	<i>See</i> dictionary
system	7

system design	836
system test.....	373, 493, 836, 840
system test plan.....	836, 844
systems engineering vee	<i>See</i> V-Model

T

tail recursion	334
task	758
TDD	<i>See</i> test-driven development
technical debt	96, 102, 519, 1045
template	411, 412
template method.....	751, 755
temporal.....	266
terminators	18
test case	11, 137, 298, 380, 445
test coverage	517
test enumeration	496
test method	497
<i>test method discovery</i>	496
test plan	11, 298, 445, 814, 841
test runner	321, 495
test suite.....	316
testcase class	496
test-driven development.....	11, 378, 516, 838
tested	440, 443
testing.....	369, 373
thick client.....	992
thick token	975
thin client	991
thin token	975
ticket.....	973
tight	518
timing diagram.....	9, 225, 817
token	965, 966
top-down design.....	350, 371
trace	11, 136, 337
trace table	140
trace-through.....	<i>See</i> trace
transition	771, 773, 776
transition arrow	776
transition condition	776
trivial.....	278, 279, 352
true negative.....	502
true positive	502
trust boundary	250, 578
truth table	160
tuple	178, 180

U

UAT	<i>See</i> user acceptance test
UML	<i>See</i> unified modeling language
underengineer	371, 702
understandability.....	91

undo	926
unified modeling language.....	26
unit	493, 838
unit test	11, 373, 492, 521, 838
unit test framework.....	495
universal product code.....	966
universalization.....	622
UPC	<i>See</i> universal product code
update	561, 896
usability	299
use case	9, 303, 545, 716
user acceptance test	842
user story	549
username	971, 976

V

validity	300
variable.....	3
variant	658
viewpoint.....	224, 225, 241, 772, 817
virtual	411
virtual method table	<i>See</i> v-table
virtual methods.....	696
virtual proxy	956
visitor.....	868, 880
V-Model.....	834
v-table	696, 697

W

wait	562
watch.....	668, 675
immediate	676
locals.....	675
memory	680, 683
programmer-specified.....	676
weak	261, 265
WHILE	49
white box testing	446, 492
wrapper.....	335, <i>See</i> adapter
WRITE	43

X

xor	159
xUnit	495, 496

Y

YAGNI	702
-------------	-----