Principles of Software: The Textbook

Dr. Konstantin Kuzmin and Alice Bibaud

January 10, 2023

Contents

1 Introduction **7**
- 1.1 Correctness . . . 7
- 1.2 Maintainability . . . 8
- 1.3 Complexity . . . 9
- 1.4 Software Engineering . . . 10
- 1.5 Tools . . . 10
- 1.6 JUnit testing framework . . . 11
- 1.7 Test-Driven Development (TDD) . . . 12
- 1.8 Object Oriented Programming (OOP) . . . 12

2 Specifications **14**
- 2.1 Why not Just Read Code? . . . 14
- 2.2 Complexity in Specs . . . 14
- 2.3 Conventions . . . 14
- 2.4 Autoboxing and Unboxing . . . 17
- 2.5 Specifications and Dynamic Languages . . . 17
- 2.6 Comparing Specifications . . . 17
- 2.7 Satisfaction of Specifications . . . 18
- 2.8 Converting PSoft Specs into Logical Formulas . . . 18
- 2.9 Review . . . 18

3 Reasoning **20**
- 3.1 Forward Reasoning . . . 20
- 3.2 Backward Reasoning . . . 20
- 3.3 Reasoning about Loops . . . 21
  - 3.3.1 Loop Invariant . . . 21
- 3.4 Induction . . . 21
- 3.5 Hoare Logic . . . 23

4 Abstraction **24**
4.1 Abstract Data Types (ADTs) . . . 24
4.2 Control Abstraction (Procedural Abstraction) . . . 24
4.3 Data Abstraction . . . 25
4.4 ADT Methods . . . 25
4.5 ADT Java Language Features . . . 25
4.6 Domain . . . 26
4.7 Representation Invariants . . . 26
4.8 Reasoning About ADTs . . . 27

5 Java **28**
5.1 Variables . . . 28
5.2 Models for Variables . . . 28
5.3 Differences between Java and C++ . . . 29
5.4 Types and Type Checking . . . 30
5.5 Equality . . . 31
5.6 Inheritance from Object Class . . . 31
5.7 Exceptions . . . 32
5.8 Compilation vs Interpretation . . . 32
5.9 Specification Fields in Java . . . 33
5.10 ADTs in Java . . . 34
5.11 Review . . . 36

6 Dafny **37**

7 Testing **40**
7.1 Quality Assurance (QA) . . . 40
7.2 Testing Scope . . . 41
7.3 Testing Strategies . . . 41
7.4 Equivalence Partitioning . . . 42
7.5 Control-Flow-Based Testing . . . 43
7.6 Coverage . . . 45
7.7 Definition-Use (DU) Pairs . . . 45

8 Identity Equality **47**
8.1 Properties of Equality . . . 47
8.2 Hash Function . . . 47
8.3 Overloading vs. Overriding . . . 48
8.4 Implementation . . . 49
8.5 Abstract Classes . . . 49

9 Parametric Polymorphism: Generics **50**
9.1 How does a method call execute? . . . 51
9.2 Parametric Polymorphism . . . 52
9.3 Using Generics . . . 53
9.4 Bounded Types . . . 53

9.5 Java Wildcards . . . 54
9.6 Contravariance . . . 55
9.7 Review . . . 56

10 Design Patterns **57**
10.1 Unified Modeling Language (UML) . . . 57
10.2 Creational Patterns . . . 60
10.2.1 Abstract Factory . . . 60
10.2.2 Builder . . . 63
10.2.3 Factory Method . . . 70
10.2.4 Prototype . . . 73
10.2.5 Singleton . . . 75
10.3 Structural Patterns . . . 76
10.3.1 Adapter . . . 76
10.3.2 Bridge . . . 78
10.3.3 Composite . . . 80
10.3.4 Decorator . . . 83
10.3.5 Facade . . . 85
10.3.6 Flyweight . . . 88
10.3.7 Proxy . . . 90
10.4 Behavioral Patterns . . . 91
10.4.1 Chain of Responsibility . . . 91
10.4.2 Command Pattern . . . 95
10.4.3 Interpreter . . . 97
10.4.4 Iterator . . . 103
10.4.5 Mediator . . . 106
10.4.6 Memento . . . 109
10.4.7 Observer . . . 111
10.4.8 State . . . 116
10.4.9 Strategy or Policy . . . 117
10.4.10 Template . . . 120
10.4.11 Visitor . . . 120
10.5 Interning Pattern . . . 123
10.6 Wrappers . . . 125
10.7 Strategy / Policy Pattern . . . 125

11 Refactoring **126**
11.1 Code Smells . . . 127
11.2 Extract Method . . . 128
11.3 Move Method . . . 128
11.4 Replacing Conditional Logic . . . 128
11.5 Replace Temp with Query Refactoring . . . 129
11.6 Replacing Type Code with State or Strategy . . . 129
11.7 Replace Conditional with Polymorphism Refactoring . . . 130
11.8 Template Method Design Pattern and Form Template Method Refactoring . . . 130

12 Usability **131**
12.1 Learnability and Memorability 131
12.2 Perception 132
12.3 Safety 133
12.4 Simplicity and Satisfaction 134
12.5 Prototyping and Testing 134
12.6 Review 135

13 Graphical User Interface (GUI) Programming **136**
13.1 Terminology 136
13.2 JFrame 136
13.3 JPanel 137
13.4 Layout Managers 138
13.5 Graphing, Painting, and other things 138
13.6 Event-Driven Programming 139
13.7 Serialization 142
13.8 Java GUI Libraries 143
13.9 Nested Classes 143
13.10 Program Thread and UI Thread 144
13.11 Components, Events, Listeners, and the Observer Pattern 145

14 Software Processes **147**
14.1 Software Life cycle and Artifacts 147
14.2 Steps 147
14.2.1 Step 1: Requirements Analysis and the Use-Case Model . 147
14.2.2 Step 2: Design Process 148
14.2.3 Step 3: Implementation 149
14.2.4 Step 4: Testing 150
14.2.5 Step 5: Maintenance 150
14.3 Requirements 151
14.4 Software Development Processes 152
14.4.1 Code-And-Fix or Ad-Hoc Process 152
14.4.2 Waterfall Process 153
14.4.3 Iterative Process 154
14.4.4 Staged Delivery 155
14.4.5 Open Source Development 156

15 Appendix **157**
15.1 Halting Problem 157
15.2 Rice's Theorem 157
15.3 Russel's Paradox 158
15.4 Java ADTs 158
15.5 Dafny 158
15.6 OpenJML 159
15.7 Design Patterns 159
15.8 Gang of Four 159

15.9 Software Errors 159
15.10 Induction 160
15.11 Testing 160
15.12 Github 160

Thanks

The design and spirit of this textbook is based on the RCOS Handbook, the document for which another RPI class, Rensselaer Center for Open Source (RCOS), is based off of. That document is maintained and updated by CS Department Faculty and the student coordinators who make it all happen.

This textbook would also not be possible without:

Dr. Ana Milanova,

Dr. David Goldschmidt,

Dr. Carlos Varela,

Dr. Michael Ernst (of University of Washington),

Professor Thompson,

All of Principles of Software's TAs and mentors for their contributions of student grading and course advice,

Dr. Barb Cutler and the Submitty team for making grading Principles of Software possible,

and Dr. Konstantin Kuzmin, who oversaw, edited, and guided the author in the creation of this publication.

This book should not be considered an original work - it is a collage of expertise woven together by an over-ambitious undergraduate at Rensselaer Polytechnic Institute. All resources, sources, quotes, and some extra ideas are dully cited in the Appendix. If you should desire to learn more about any idea in this book, consult the Appendix or contact Alice Bibaud at alice.bibaud@gmail.com.

1 Introduction

This course is about writing correct code. Correct in this case describes code that is error free, maintainable, and does what we want it to do. To that end, we must ask a few questions about writing correct code:

1. How can we describe the code we intend to write in an enforceable way?
2. How can we guarantee that our code matches our specifications?

The remainder of this book and the course that goes along with it will illuminate the answers. Though they may seem logically intuitive initially, these concepts are essential in industry and academia.

1.1 Correctness

Correct code is error-free code that does what we want it to do. That seems simple, right? No. Correct code also implies:

1. maintainability, which we'll get to later
2. that the code adequately matches its specifications.

Programmers achieve correctness through principled design and development, abstraction and modularity, and documentation. We will outline specific ways to go about doing all of this later. Additionally, readers can find information about abstraction in the ADT sections, modularity in the Design Pattern and Refactoring sections, and documentation in the Java Doc section. Principled design and development refer to programming with the best practices outlined in this book.

We are also interested in verifying correctness. Students will do that through reasoning about code, testing, and automated verification. The Halting Problem and Rice's Theorem together prove that these are necessary concepts worthy of attention. Luckily, correctness is provable in specific cases, and the usefulness of doing so is immense:

> Almost all (92%) of catastrophic system failures result from incorrect handling of non-fatal errors explicitly signaled in software... In 58% of the catastrophic failures, the underlying faults could easily have been detected through simple testing of error handling code.
>
> - Ding Yuan et al., Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems, 2014

We care about correctness because caring, paired with adequate designing and testing, stops catastrophic failures from occurring often and ultimately saves

money and lives. Programming errors cost the U.S. economy $60 billion annually in lost efficiency and real harm. Here are a few examples of such failures:

1. **The Mariner 1 Spacecraft, 1962**: before NASA sent a person to space, they sent an unmanned space ship to fly past Venus. An omitted hyphen caused the craft to veer off course and, fearing a devastating crash-landing moments after launch, the ship self-destructed. Total present-day cost of failure: $169 million.

2. **The Morris Worm, 1988**: Then-graduate student Robert Tappan Morris is credited with creating the first software virus, which he sent through an early version of the internet. Dr. Tappan is now a professor at MIT, and is known for exposing an early digital vulnerability through helpful, chaotic mischief. Total present-day cost of failure: $25 million.

3. **Bitcoin Hack, Mt. Gox, 2011**: Mt. Gox was the biggest bitcoin exchange in the world in the 2010's, until they were hit by a software error that ultimately proved fatal. The glitch led to the exchange creating transactions that could never be fully redeemed, costing up to Total present-day cost of failure : $2 million.

4. **NASA's Mars Climate Orbiter, 1998**: The Mars Climate Orbiter burned up after getting too close to the surface of Mars because there was a software error in converting imperial units to metric. Total present-day cost of failure: $580 million.

5. **Y2K, 2000**: At the turn of the twenty-first century, software users were concerned that computer systems around the world would not be able to cope with dates after December 31, 1999, because most computers and operating systems only used two digits to represent the year. Software systems handled the situation well, though governments and citizens alike around the world spent a lot of money preparing for the inevitable collapse. Total present-day cost of distrust in the technology: $172 billion.

The list goes on and on, but hopefully this is enough to convince even the most wary computing students that paying attention to what your software does in the world is very, very important.

1.2 Maintainability

What does code require to be maintainable? The code base should be well-organized, consistent, well-documented, understandable (this is up for interpretation, but can also very importantly refer to variable naming conventions), and follow the Open-Closed Principle:

Be open for modification, but closed for extension.

For example, suppose we have an editor that manipulates two-dimensional shapes. We have created Square and Circle objects, and have a program that moves these shapes around the user's screen. Adding a Triangle (open for extension) should be easy without any changes to the existing code that manipulates shapes (closed for modification). Modularity depends on the open-closed principle; good programmers do well to remember it.

We can achieve maintainability, like correctness, through principled design and development, abstraction and modularity, and documentation, with the addition of correct object-oriented approaches. These are the best practices for software development. It is vital to take these concepts to heart in this course.

1.3 Complexity

So... why is software so hard? Software is not very aptly named, is it? Writing software is hard because of complexity, and by extension, the symptoms of complexity. At its core, programming well is about managing complexity - anything related to a software system's structure that makes it hard to understand and modify.

We manage complexity through modular design. Software is composed of a collection of relatively independent modules. A class, a subsystem, a service, or even a method in a script, can all be considered modules. In an ideal world, modules act entirely independently of one another. However, modules always have to depend on one another, something we call a dependency. Managing these dependencies manages the complexity of the overall system.

One way to manage complexity is to separate the interface from the implementation: the user interacts with a module through an interface, where everything the module does is abstracted away from the user.

How does complexity arise in the first place? Complexity is sneaky; it creeps into code over time, and if its causes go untreated, it can be fatal to the affected systems. Dependencies typically cause complexity. Dependencies exist when pieces of code cannot be understood without one another. Obscurity in code is the existence of dependencies, and is the opposite of the open-closed principle. When multiple files need modification during debugging or feature creation, dependencies are present. Luckily, some amount of dependencies are unavoidable - systems with multiple parts require dependencies to link all their modules together. However, generally avoiding excessive dependency reduces cognitive load during programming.

What are some symptoms of complexity? Change amplification, where simple changes in one module require disproportional amounts of changes in other

files, is a telling sign of complexity. Good design's primary goal is to reduce change amplification in the system, reducing cognitive load on the programmer. Cognitive load is the amount of knowledge a system developer needs to contribute to a code base. How will programmers fix bugs and meaningfully contribute if it is unclear what their code is doing? Best case, each new feature will require excessive code modification, because developers spend more time acquiring enough information to make changes safely. Worst case, programmers lack the necessary information and break the system. The bottom line is that complexity makes modifying an existing code base difficult and risky.

1.4 Software Engineering

Building correct software is complicated. Large software systems are enormously complex; they consist of many moving, perpetually-changing parts. Large systems often face problems scaling up due to poor specification quality, which can cause code smells, bugs, and misuse of the resulting product.

Software engineering is about mitigating complexity, managing change, and handling software failures.

1.5 Tools

Principles are more important than tools, but good tools are important for acting on principles. In RPI's version of this class, we use:

- Java: an industry-standard object-oriented programming language
- Eclipse: powerful integrated development environment (IDE)
- Git: version control (more on this coming up)
- JUnit 4: a testing framework for Java
- Submitty: a homework submission server developed by RCOS (Rensselaer Center for Open Source) at RPI. Submitty is a system for submitting assignments, auto-testing code submissions, and assigning grades.

Version control records changes to files over time. With version control, developers may:

- Checkout (download) files from a centralized repository into a local machine
- Add local files to the centralized repository, where other users can see, obtain, and modify them
- Commit changes by adding file changes to the repository
- Update local files relative to the repository

With version control, programmers can also merge files. Merges occur when multiple users have modified the same file in a repository. Hopefully, the changes do not conflict, though sometimes they do. When changes cannot automatically merge, programmers manually resolve conflicts as they occur.

1.6 JUnit testing framework

JUnit is a unit testing framework for Java that supports writing and running unit tests. Unit testing is a testing class whose scope is just one module. In object-oriented programming, the smallest testable unit is the class. Therefore, unit testing can also be called class testing.

JUnit uses annotations to specify test runs:

- `@BeforeClass` is the static method that configures the test run. The JUnit framework runs this method before all tests and creates an instance of the module being tested
- `@Test` marks a method as a test method; it is run in JUnit as a test
- `assertEqual` takes in an expression and its expected result, and displays a message if the two arguments match.
  Syntax for this is: `assertEquals( message, expected result, actual expression )`
- `assertTrue` takes in a boolean expression and displays a message if the expression resolves to `true`.
  Syntax for this is: `assertTrue(message, boolean_expression)`

For example:

```
public class SaleTest {
    private static Sale sale = null;
    private static double ITEM_PRICE = 2.5;

    @BeforeClass
    public static void setupBeforeTests() { sale = new Sale(); }

    @Test
    public void testGetTotal() {
        sale.makeLineItem( "item1", 1, ITEM_PRICE );
        sale.makeLineItem( "item2", 2, ITEM_PRICE );
        assertEquals( "sale.getTotal()", 7.5, sale.getTotal() );
    }
```

1.7 Test-Driven Development (TDD)

Modern software development methodologies, such as Extreme Programming (XP), Agile, and the Unified Process (UP), significantly emphasize unit testing. They advocate for Test-Driven Development (TDD), or test-first development. The idea is to write all unit tests before writing software and after designing. Test-first development allows the software engineer to take care of inadequacies in the code base before they even exist. TDD is also helpful because:

- Unit tests are always written
- Programmers are satisfied with their work, which leads to more consistent test writing
- Correct interface behavior is clarified, so code is easier to write
- Programmers are forced to think about correct inputs, outputs, and edge case handling
- Repeatable, automated verification is built into the code base

1.8 Object Oriented Programming (OOP)

An object is a piece of software that can store values and perform operations. Here, value storage is called "state," and operations are called "methods." Objects are integral to objecty-oriented programming, a standard of organization many modern programming languages adhere to. The four object-oriented programming concepts are:

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism

Object Oriented Programming requires discipline. Luckily, we have codified that discipline into a series of design principles:

- Single Responsibility Principle: a class should only have one responsibility
- Open-Closed Principle: software entities should be open for extension, but closed for modification
- Interface Segregation Principle: many client-specific interfaces are better than one general-purpose interface
- Dependency Inversion Principle: one should depend on abstractions instead of concrete implementations

Though it is the only one covered here, object-oriented programming is only one of many ways to think about software. Object-oriented programming is a superset of imperative programming, a model which executes a series of steps in a specific order. Procedural programming groups instructions into procedures. In declarative programming, the programmer declares properties of the desired result, but not how to compute it. In functional programming, the result is declared as the value of a series of function applications. In logical programming, the result is written as the answer to a question about a system of facts and rules. Lastly, in mathematical programming, the result is the solution to an optimization problem. Studying functional, logical, and mathematical programming outside this class is encouraged, but not required. These concepts are widely used in industry and higher-level computing courses at RPI. This class focuses on object-oriented programming as applied to principles of software.

2 Specifications

A **Precondition** is a condition that must be true before method execution, and a **Postcondition** is a

2.1 Why not Just Read Code?

Why not just read the code to see what is happening in the system? Code is complicated; large systems contain more detail than anyone trying to parse the abstractions needs, and understanding code can be a sizeable cognitive burden. Code is also ambiguous - it is often unclear what is essential and incidental in the code base. **Incidental Code** is dead code that comes about as the result of optimization. Most importantly, the client needs to know what the code does, not how it works! Even in mathematical and scientific software, the user can only know *some* implementation details.

What about comments? Those are important, but not sufficient, for proper specification. They are often informal, ambiguous, or misleading, so they cannot stand in for formal specs. Comments must also be updated in tandem with the code to avoid problematic misunderstandings, which does not happen often enough in the real world. Unfortunately, most code lacks specification; programmers guess what it does by skimming through or running it. Lack of documentation results in bugs and complex code with undefined behavior. It would be cool to generate code from specifications; this is currently in development and available in some spaces.

2.2 Complexity in Specs

A complex specification is a red flag; it is better to simplify design and code rather than try to describe complex behavior! Over-complicated specifications indicate that something needs to be fixed or the code needs to be simplified. Software designers constantly battle complexity, so good specifications help reduce the programmer's cognitive burden.

2.3 Conventions

There are standard specification conventions that make writing readable code easier. There are many to choose from.

JavaDoc specifications are simple text descriptions of the method's behavior that reside in a method's type signature:

```
\*
  * parameters: text description of what gets passed
  * returns: text description of what gets returned
```

```
  * throws: list of possible exceptions
  *\
  public String methodName() { ... }
```

Principles of Software (PSoft) specification conventions are also included in the code as comments. The precondition is the requires clause, and the postcondition is comprised of all other clauses:

```
\*
  * requires: clause spells out constraints on client code
  * modifies: lists objects that may be modified by the method.
  * Any object not listed under this clause is guaranteed untouched.
  * effects: describes the final state of modified objects
  * throws: lists possible exceptions
  * returns: describes the return value of the method
  *\
  public String methodName() { ... }
```

A few additional comments describing the function help increase readability for the programmer.

When using PSoft specifications, all tags must be assigned. Use `none` when fields need to be left blank. Method signatures are part of the spec but do not need to be written elsewhere.

Sometimes it can be hard to capture all potentially relevant details in the spec. If required, add more to the specification to make it more specific - just not at the cost of efficiency!

The `requires` clause is critical in PSoft Specifications. If the client fails to provide preconditions, then the method can do anything it wants – even throw an exception or return an object of the wrong type. It is important to double-check the `requires` clause in method implementation.

Checking preconditions makes implementation more robust, provides feedback to the client, and avoids silent errors. It is not always necessary to check the preconditions of private methods since those are not client-facing and should be compliant with specifications by default. However, if the method is public, preconditions should be checked unless such a check is computationally expensive.

Java Modeling Language (JML) is another important specification convention; Javadocs and PSoft specifications are not "machine-checkable" languages, whereas JML is. JML is a formal specification language built on Hoare logic. The end goal of JML is automatic verification. Here is an example of a binary search implementation with JML specs, adapted from the OpenJML Tutorial:

```
public class BinarySearch {
//@ requires sortedArray != null && 0 < sortedArray.length
    //  < Integer.MAX_VALUE;
//@ requires \forall int i; 0 <= i < sortedArray.length;
    //  \forall int j; i < j < sortedArray.length;
    //  sortedArray[i] <= sortedArray[j];
//@ old boolean containsValue =
    //  (\exists int i; 0 <= i < sortedArray.length;
    //  sortedArray[i] == value);
//@ ensures containsValue <==> 0 <= \result < sortedArray.length;
//@ ensures !containsValue <==> \result == -1;
//@ pure
public static int search(int[] sortedArray, int value) {
     //@ ghost boolean containsValue = (\exists int i; 0 <=
        //  i < sortedArray.length; sortedArray[i] == value);
     if (value < sortedArray[0]) return -1;
     if (value > sortedArray[sortedArray.length-1]) return -1;
     int low = 0;
     int high = sortedArray.length-1;
     //@ loop_invariant 0 <= low < sortedArray.length &&
        //  0 <= high < sortedArray.length;
     //@ loop_invariant containsValue ==> sortedArray[lo] <=
        //  value <= sortedArray[high];
     //@ loop_invariant
        //  \forall int i; 0 <= i < low; sortedArray[i] < value;
     //@ loop_invariant \forall int i; high < i < sortedArray.length;
        //  value < sortedArray[i];
     //@ loop_decreases high - low;
     while (low <= high) {
         int mid = low + (high - low) / 2;
         if (sortedArray[mid] == value) {
             return mid;
         } else if (sortedArray[mid] < value) {
             low = mid + 1;
         } else {
             high = mid - 1;
         }
     }
     return -1;
}
}
```

2.4 Autoboxing and Unboxing

A **Primitive Data Type** in Java is an object whose value is intrinsic and does not have the properties of an object. However, **Boxing**, the practice of wrapping primitive values in Java objects, can impart object properties onto primitives. Boxing is useful when comparing Java objects to primitives. Java has a feature called **Autoboxing**, automatically Boxing primitives where needed. **Unboxing** automatically converts Java objects (boxed types) to primitive types when required.

Here is an example of Boxing, Autoboxing, and Unboxing:

```
int a = 3;   // regular primitive data type
Integer b = 3;  // AutoBoxing: 3 is automatically converted to Integer,
             //  int's Java object "wrapper" class
int c = b;   // Unboxing: Integer 3 to int 3
```

2.5 Specifications and Dynamic Languages

A type signature is a form of specification in statically typed languages like C/C++ and Java. In dynamically typed languages, there is no type signature! So how do we write specs for them? A method is called for its side effects (`effects` clause) or its return value (`return` clause); it is bad style to have both effects and return blocks. There are exceptions, though; the main point of a specification is to be helpful. Overly formal specifications are not helpful, while being too informal does not help either.

2.6 Comparing Specifications

Sometimes, we need to compare specifications. Specifications can be compared in two ways:

1. by hand, examining each clause
2. with logical formulas representing the spec

Use whichever is most convenient; comparing specs enables reasoning about substitutability. For instance, "P is stronger than Q" means that, for every implementation Q, Q satisfies $P \rightarrow Q$, which satisfies P. If the implementation satisfies the stronger spec (P), it also satisfies the weaker (Q). However, the opposite is not necessarily true. In general, we want substantial specifications. A larger world of implementations satisfies the weaker spec Q, then the stronger spec P. Consequently, a weaker spec can save time and energy while still being apt for its application.

Specifications can be strengthened by weakening their preconditions and strengthening their postconditions. For example, with **Contravariance**, supertypes can

replace argument types, weakening the precondition. Similarly, with **Covariance**, a subtype can replace a method's output type. Either way, types can be adjusted so specifications do not violate the client's expectations. Just do not throw any new exceptions.

A stronger specification is easier to use - the client has fewer preconditions to meet, and the client gets more guarantees in postconditions. However, they are also harder to implement. On the other hand, a weaker specification is easier to implement because it has a more extensive set of preconditions, relieves implementation from the burden of catching errors, and is easier to guarantee less to the client in the postcondition. Unfortunately, with all of this simplification, weaker specs are often more challenging to use due to their lack of rigor.

Sometimes, we use a method where another is expected with subclasses. We must obey the **Liskov Principle of Substitutability**: An object with a stronger specification can be substituted for an object with a weaker one without altering desirable properties like correctness or tasks performed.

2.7 Satisfaction of Specifications

P is an implementation, and Q is a specification. P satisfies Q if every P behavior is permitted by Q, and no P behavior violates Q. The statement "P is correct" is meaningless but often used; if P does not satisfy Q, either or both could be wrong. When P does not satisfy Q, it is usually better to change the program rather than the spec. If the spec is too complex, modify the spec.

2.8 Converting PSoft Specs into Logical Formulas

```
requires: R
modifies: M
effects: E

==

R => (E and (nothing but M is modified))
```

In PSoft Specs, `throws` and `returns` are absorbed into effects E.

2.9 Review

It is difficult to compare specifications; comparison by hand can be easier, but could be more precise. It may be difficult to see which conditions are stronger. Comparison by logical formulas is accurate, though sometimes expressing behaviors with precise, logical formulas is difficult.

Comparing by hand uses the `requires` clause:

```
Requires: stronger spec has fewer conditions; they require less.

Modifies and Effects: stronger spec modifies fewer objects.
        A stronger spec guarantees that more objects stay unmodified!

Returns and Throws: a stronger spec guarantees more in returns and
        throws clauses.  They are harder to implement, but easier to use.
```

3 Reasoning

Reasoning about code allows us to understand the exact behavior of our code before we run it, deeming it necessary for efficient development. Reasoning uses specifications to verify correctness, find errors, and understand those errors in code.

There are two types of reasoning:

1. **Forward Reasoning:** given a precondition, does the postcondition hold? This approach is suitable for discovering previously unknown edge cases in a program's output.
2. **Backward Reasoning:** given a postcondition, what is the proper precondition? This approach is useful for finding out which inputs are causing errors.

Backward reasoning is usually preferable to forward reasoning. Given a specific goal, backward reasoning shows what must be true before execution to achieve desired results. Given an error, it gives input that exposes the error.

3.1 Forward Reasoning

Forward reasoning is straightforward: given a precondition, go line by line through the code to evaluate what is true at the end of the program. For example:

```
// precondition: x is even
x = x + 3;  // x is odd
y = 2x;   // x is odd && y is even
x = 5;   // x = 5 && y is even
// postcondition: x = 5 && y is even
```

3.2 Backward Reasoning

Unlike forward reasoning, there is some specific syntax for backward reasoning. To denote our thought processes, we write `wp("expression," {Q}) = {Q}`, where wp stands for weakest precondition. We want the weakest precondition, so when we carry our work up to the top, the weakest precondition for the whole function remains. For example:

```
// precondition: x > ( -1 / 2 )
x = x + 3; wp( x = x + 3, { x > (5/2) } ) = { x > ( -1 / 2 ) }
y = 2x;  wp( y = 2x, { y > 5 } ) = { x > ( 5 / 2 ) }
x = 5;  wp( x = 5,  { y > x } ) = { y > 5 }
// postcondition: y > x
```

3.3 Reasoning about Loops

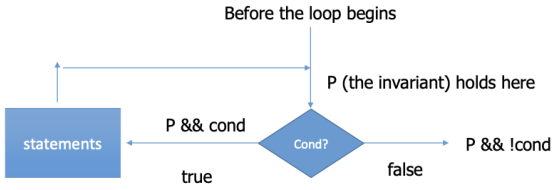
Computing the weakest possible precondition for loops is a complex problem to solve. However, a tool is available to developers for such a task: the loop invariant. We can find loop invariants using computational induction, a systematic formula for proving that recursive functions behave as expected.

3.3.1 Loop Invariant

Reasoning about loops can be tricky because a loop represents an unknown number of paths. Recursion presents the same problem - it might not be possible to enumerate all paths. The key to both is to choose a loop invariant, then prove by induction that the loop's specifications hold for each iteration.

A **Loop Invariant** is a condition that is true immediately before and immediately after each iteration of a loop. The execution of the loop body should preserve it. However, it does not say anything about the state of specifications part of the way through; this is because variables often change during loop execution in ways that temporarily violate the spec. We can reason about loop invariants using induction.

In **Forward Reasoning with a Loop**, a loop invariant must be true before entering the loop, after each iteration, and after the loop exits. It must also be relevant. A good loop invariant should involve the changing loop variable and the postcondition. *The decrementing function and the loop invariant must imply the postcondition.*



3.4 Induction

Reasoning about ADTs uses induction. When reasoning about representation invariants, the inductive step must consider all possible changes to the representation, including representation exposure. If the proof does not account for representation exposure, this is invalid.

The steps to successful induction are as follows:

1. **Initial Step**: Prove the proposition is true for $n = 0$.

2. **Inductive Step**: Prove that if the proposition is true for n = k, then it must also be true for n = k + 1. This step is complex, and breaking it up into several stages helps.

    - **Inductive Hypothesis**: Assume what the proposition asserts for the case n = k.
    - *Step 2*: Write down what the proposition asserts for the case n = k + 1. This is what the induction proves.
    - *Step 3*: Prove the statement in Stage 2 using the assumption in Stage 1. The technique varies from problem to problem, depending on the mathematical content.

Use ingenuity, common sense, and knowledge of mathematics here. Additionally, RPI's Foundations of Computer Science class and associated text have suitable methods for solving proofs with induction.

To illustrate this concept, see the example below:

```
int x = 10;
int z = 0;

\\ precondition: x >= 0 && i == x
\\ postcondition: z = x

for (int i = x; i > 0; i--) {
    z += 1;
}
```

1. **Initial Step**: In this loop, the decrementing variable is `i`, and the variables of interest are `x` and `z`. Thus, our loop invariant (LI) must incorporate these variables. The precondition values simplify to `i == x`; coupled with `z == 0`, we get `i + z = x` as our LI. We know it holds before the loop code executes by substituting the variables in for their values: `(10) + (0) = (10)`.

2. **Inductive Step**: Assuming `i + z == x` holds after iteration `k`, we show that `i + z == x` holds after iteration `k + 1`.

    - Assume that `(i - k) + (z + k) = x`. If that is indeed true, then `(i - k - 1) + (z + k + 1) = x` must also be true, because `(i - k - 1) + (z + k + 1) = (i - k) + (z + k) - 1 + 1 = (i - k) + (z + k) = x`
    - If the loop terminates, we know `i <= 0`. We have `z == x`, which is the postcondition.
    - We know if the loop terminates because the precondition `x >= 0` guarantees that `i >= 0` before the loop. At every iteration, `i` decreases by `1`. Thus, it eventually reaches `0`.

3.5 Hoare Logic

Hoare Logic is a formal framework for reasoning about code; its proper use can mechanize the process of reasoning about code. Sir Anthony Hoare created (or discovered, depending on your beliefs) Hoare Logic, the quicksort algorithm, and a bunch of programming languages. He won the Turing award in 1980.

A **Hoare Triple** is written as $\{P\}code\{Q\}$, where P and Q are logical statements about program values, and *code* is a given program's code. If a program satisfies condition P before execution terminates, it will terminate in a state satisfying condition Q. In other words, *If P && code, then Q.*

Why care about Hoare triples? Because they can help us guarantee postconditions using preconditions and code.

In a Hoare triple, we want the **Weakest Possible Precondition**: the most lenient condition possible, which still ensures correct output. Similarly, we want the **Strongest Possible Postcondition**: the most restrictive postcondition possible, which still allows the precondition to hold.

We can always substitute a stronger precondition, and the original Hoare triple can still be true. We can always substitute a weaker postcondition, and the Hoare triple can still be true. In an if-then-else block, we take the if and else statements and stick them in the Hoare triple like this, "or-ed" together:

```
// precondition: ??
if (b) S1
else S2
// postcondition: Q

wp(\if (b)S1 else S2", Q) = { ( b && wp(\S1",Q) ) ||
    ( not(b) && wp(\S2",Q) ) }
```

4 Abstraction

Abstraction is the hiding of unnecessary low-level details. Thus, abstractions that include unimportant information are useless. They are also unhelpful when they fail to include necessary information, which defeats their purpose. The point of having abstractions instead of reading code is to minimize the burden of remembering all that code.

Cognitive Load is the amount of knowledge a programmer must have to work on the code base. Abstractions are meant to make life easier - to reduce cognitive load. **Information Hiding** is a design principle that hides the parts of a program likely to change from the client. An **Abstract Data Type (ADT)**, then, uses information hiding toencapsulate of an object and its operations for faster, easier development.

4.1 Abstract Data Types (ADTs)

Luckily, despite how scary they sound, ADTs are all around us and easy to grasp: they are anything that represents something else in data. So, a `Set` in Java might represent a real-life mathematical set. An even more concrete example might be a book. Books come in all shapes, sizes, colors, and purposes, so `Book` is an ADT. However, `The Principles of Software Textbook` is a very concrete implementation of that abstract concept.

In addition to encapsulating an object and its operations, ADTs can be a specification mechanism, a way of thinking about programs, and a helpful design tool. Organizing and manipulating data is pervasive in code, though inventing and describing algorithms is not.

Thus, we should consider data representation when designing large software systems. Before implementation, consider what the data represents: will necessary operations be efficient without compromising understandability? ADTs solve this problem in well-written programs by shielding users from implementation – clients do not directly depend on the program's internals.

4.2 Control Abstraction (Procedural Abstraction)

Control Abstraction, or **Procedural Abstraction**, is the procedure for implementing code according to specifications, where method names, method signatures, and specifications are exposed to the client, but implementation is hidden. One part of this is a **Method Signature**, sometimes referred to simply as a **Signature**, which provides a method's name, parameter types, and return type. With these pieces, ADTs supply the abstraction barrier between implementation and representation.

4.3 Data Abstraction

Data Abstraction is where the data representation of a class is hidden from the client. For example: how are strings implemented in Java? Are they fixed arrays of characters? Linked lists? Data abstraction abstracts the use of information away from its representation. For example, a string is an abstraction for an array of characters.

4.4 ADT Methods

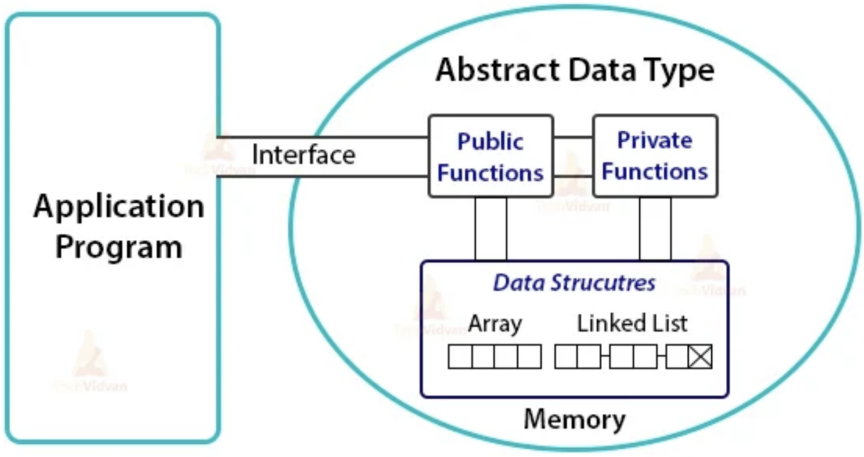
We can group the methods of an ADT into creators, observers, producers, and mutators, which all relate to an object O of type T. **Creators** create a new object of type T; constructors are creators. **Observers** return information about O; getter methods are observers. **Producers** return a new object of type T by performing operations on the given object O. **Mutators** change the object O; setter methods are mutators.

ADTs can be immutable or mutable. **Immutable Objects** cannot be changed, while **Mutable Objects** can be changed. No mutators exist in an immutable object's ADT, and producers are rare when specifying the ADT for a mutable object.

4.5 ADT Java Language Features

In Java, ADT operations are usually **public**, or accessible to users, and other operations are **private**, or inaccessible to users. We will talk more about this in the context of access control modifiers in the Java section of this text.

Clients can only access ADT *operations*; because ADTs rarely have producers and are most often immutable, new ADTs must be created to change older ones. This way, an ADT can act as a specification or set of operations.



4.6 Domain

The mathematical concepts of domain and range also apply to ADTs. In a data type, the **domain** consists of all implementations that satisfy the representation invariant. The representation invariant simplifies the abstract function by restricting its domain. The **range** of an ADT is the restriction of its possible stored values. The range can be tough to denote relative to the domain; it is easy for mathematical concepts like sets, but hard to define for real-world ADTs. For instance, in a set, the range might be restricted to the type of value it can contain. In other data types, the same range could be applied, but may also contain artifacts from the specifications of the ADT, such as a value range or a maximum length to a list.

4.7 Representation Invariants

The **Representation Invariant** states the constraints that data structures and algorithms impose on implementation. For example, a binary tree's representation invariant should verify that it has no cycles. An immutable object's representation invariant is that it can never be changed; representation invariants can be as simple as that.

Representation Invariants also describe whether or not ADTs are well formed. They must be true before and after every method is executed. In that way, they are like loop invariants - they do not need to hold during execution. The correctness of operation implementation depends on well-formed data. Luckily, the representation invariant ensures correctness by excluding values that do not correspond to abstract values.

Representation Exposure, when a necessary aspect of the ADT is left out of the representation invariant, is a significant problem. We can avoid making representation exposures by practicing **Defensive Programming**, where we write code assuming that the client will make mistakes: checking the representation invariant, preconditions, and postconditions against the implementation. The role of the representation invariant is to simplify the abstraction function by limiting valid concrete values, which then limits the domain of the abstraction function.

4.8 Reasoning About ADTs

The best way to start coding is to design and plan carefully by writing specifications, representation invariants, and abstraction functions. However, more is needed. After designing and implementing according to spec, the code must be verified using reasoning. Verification aims to prove that its implementation satisfies the specifications. Verification can be difficult, depending on the program. Luckily, it is also possible to use proofs to verify that the representation invariant holds.

In Java, we can add a **checkRep() Method** that checks implementation by verifying the representation invariant after each method. Of course, if the ADT is immutable, the representation invariant does not require checking beforehand, but it is always good practice to make sure. Unfortunately, exhaustively testing for representation exposures is often impossible, so we have to choose well what goes into the `checkRep()` method.

Through reasoning, we can prove that all objects satisfy the representation invariant. Know how to use reasoning appropriately; thorough testing can be easier than representation invariants to implement. When it is not, we can prove that all objects satisfy the representation invariant through induction and considering all ways to make a new object with creators and producers.

Review

The Representation Invariant defines the set of valid objects in implementation.

The Abstraction function defines, for each valid object, which abstract value it represents.

Together, they modularize the abstract data type's implementation, making it possible to reason about operations in isolation. Neither is part of the ADT abstraction.

5 Java

Java is a successor to several languages, including Lisp, Simula67, CLU, and SmallTalk. It is superficially similar to C and C++ because its syntax is borrowed from them. However, at a deeper level, it is quite different from C and C++.

Like C and C++, Java is an object-oriented language, where most of the data manipulated by programs is contained in objects. Objects contain both state and operations, where the state is comprised of fields and variables, and operations are called methods or functions.

5.1 Variables

What is a variable? A variable is a kind of state - a place to store objects. For example:

```
int P = Q;
```

P and Q are integers in this line of Java code. P is an **r-value**, a variable on the right of the assignment operator that stores data. Because Q is on the left, Q is an **l-value**, which refers to the location of Q's data. Thus, we have P and Q, but "buckets" containing data. Because we are setting P equal to Q, we are "pouring out" the data stored in P's location, and "filling P" with the data stored in Q's location. There are now two buckets filled with Q's data.

For another example:

```
int b = 2;
int c = b;
int d = b;
```

We know what value `b` is outright. Then, we copy the data stored at location `b` to location `c`. Finally, we combine data at `b` and `c` and copy it to `d`.

5.2 Models for Variables

A variable is a reference to an object with a value; every variable is an l-value that can be dereferenced to get an r-value. Different languages have different models for how they store variables. For example, C++ uses the value model, where references are aliases, and the "buckets" it uses as references are called pointers. Java and Python use a mixed model, where primitive types use the value model, and class types use the reference model. Objects are created with the keyword new.

To clarify the difference, here is how to create a string with a value model language:

```
int x = 7; // value model
char c = "Ahhh";
```

Furthermore, here is how to create a string with a reference model language:

```
String s = new String("cat");
// s contains a reference to an object containing "cat."
```

In Java, local variables live on the runtime stack. On the stack, memory is allocated for variables when their method is called, and deallocated when their method returns. Primitive types of variables (int, double, char, boolean) are stored on the stack. All other types of variables, including strings and arrays, contain references to objects that reside on the heap.

5.3 Differences between Java and C++

In Java, there is no reference arithmetic needed to iterate through pointers. A reference might be implemented by storing an address, but it is impossible to exchange integers and pointer values as in C++. References are strongly typed in Java, whereas C/C++ pointers can point to any object using casting. Java does not have explicit pointers, but C/C++ does.

In Java, there are two ways to denote equality: `==` and `.equals`. == is used when comparing primitive objects stored on the stack, and **.equals** when comparing objects stored on the heap. Java is an interpreted language, while C/C++ is a compiled language. Polymorphism is also handled differently here.

In Java, access control modifiers are different; protected is slightly different in C++. Java also has built-in garbage collection, whereas C++ does not. In C++, the programmer is responsible for memory management, whereas in Java, programmers do not need to explicitly free memory. Below is a table with access modifiers and what they do to program visibility:

	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

A protected object is visible to its parent class, the other classes in its package, and subclasses, but not to the outside world.

Private objects are only visible to the class containing the object.

There are many differences between C++ and Java. For example, C++ allows multiple inheritance and can give rise to the dreaded triangle inheritance

dilemma. Java does not allow this; it only allows for single-class inheritance, though it does allow for multiple interface inheritance. For example:

```
class B extends A implements J, K, L, ...
    public interface ShapeInterface {
        public void draw();
        public double getArea();
    }
    public class Square implements ShapeInterface {
    ...
}
```

In Java, an Interface cannot contain any implementation, only method signatures. This restriction creates a contract between the interface class and subclasses; interfaces cannot be instantiated independently. Abstract classes can contain implementations for some methods, though they cannot be directly instantiated. Abstract classes can be subclasses, though.

```
public abstract class AbstractMapEntry<K,V> implements MapEntry<K,V> {
public abstract K getKey(); // must be implemented by subclass
public abstract V getValue();

public boolean equals( Object o ) {
    if ( o == this ) // etc.
    }
...
}
```

In C++, we use the terms base class and derived class to refer to the equivalent Java superclass and subclass. In C++, we have member variables and member functions, whereas in Java, we have instance or static fields and methods. Java also has interfaces, which are just collections of method signatures.

5.4 Types and Type Checking

Types create abstraction and safety within a program. By disallowing operations on unsupported objects, types and type checking prevent the program from going wrong.

Java is a strongly typed language. It checks the code to ensure that each assignment and method call is typed correctly. Every variable declaration gives the variable a type; the header of every method defines its type signature. Legal programs are guaranteed to be type-safe - they will not run otherwise. Java

provides automatic storage management for all objects and checks all array accesses to ensure they are within the proper bounds. Objects out of bounds throw ArrayIndexOutOfBounds Exception.

Type safety is where no operation is ever applied on an object of the wrong type; C/C++ is type unsafe. Java and C/C++ are statically typed, which means they require type annotations. Most type-checking occurs before runtime. In Java and C++, expressions have static (compile-time) types, and objects have dynamic (runtime) types. The alternative to strong typing is dynamic typing, where substantial type-checking occurs during runtime. Java ensures type safety with both compile-time and runtime type checking. As a result, the compiler rejects plenty of programs.

Some checks are left for runtime. For example, where `B b = (B) x`, Java will not check b's type during compile-time because it looks correct. Instead, Java will save that check for runtime, when it checks that `x` refers to a `B` object. If not, it throws an exception.

5.5 Equality

There are two kinds of equality - reference equality and value equality. We went over this earlier.

5.6 Inheritance from Object Class

In Java, `Java.lang.Object` is the top-level class. Every other class is a subclass of object; every class implements its methods. `Object.equals()` uses == when checking for equality. `Object.equals()` tests for identity; does not test for equality of contents.

Java deals in subtyping, and PSoft deals in subtype polymorphism. Subtype polymorphism ensures that a subclass superclass can be used where a superclass is expected. Subtype polymorphism relies on the open-closed principle to work.

In C++, static binding is the default; dynamic binding is specified with the keyword virtual. In Java, dynamic binding is the default; private, final, or static methods in a class use static binding. Static binding happens at compile time; at binding, the compiler knows the class of all objects. On the other hand, dynamic binding happens at runtime, where methods can be overridden, and the compiler does not know what type the object will be at runtime.

Overloading is the ability to use the same method name with different arguments. In contrast, **Overriding** is where a subclass uses the same method signature as a superclass in inheritance. Overloaded methods are bound using static binding during compile time, and overridden methods are bound using dynamic binding at runtime.

Subtype Polymorphism is useful because it enables extensibility and reuse. It also enables (and depends on) the open/closed principle.

Dispatching - what is dispatching? It is where the compiler knows the apparent type of an object, while at runtime, the object has an actual type. **Dispatching** calls code in its actual type at runtime.

5.7 Exceptions

Java throws lots of exceptions. Here are some examples:

`ArrayIndexOutOfBounds at x[i]=0;` is thrown if i is out of bounds for array x

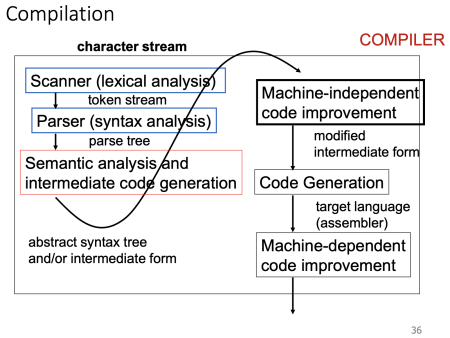
`ClassCastException at B q = (B) x;` is thrown if the runtime type of x is not a B

`NullPointerException at x.f = 0;` if x is null

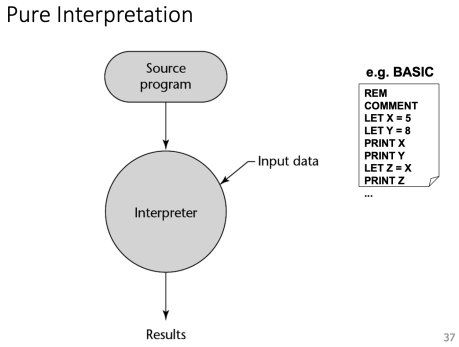
Exceptions are helpful - they tell us what went wrong and prevent applications from messing themselves up too much.

5.8 Compilation vs Interpretation

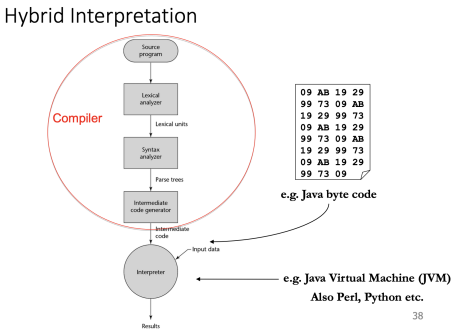
In compilation, a high-level program is translated directly into executable machine code. C++ uses compilation.



In pure interpretation, a program is translated and executed one statement at a time using an interpreter.



In hybrid interpretation, which Java uses, a program is compiled into intermediate code, and then intermediate code is interpreted. Java uses both a compiler and an interpreter to run.



Compilation begets faster execution, though it depends on many different factors. Interpretation provides greater flexibility through dynamic (type) checks; other dynamic features are much more manageable. Interpreters are also easier to write than compilers.

5.9 Specification Fields in Java

Describe abstract values. Abstract values are used in the overview (before `@requires`) in your JavaDoc spec of the abstract data type. The abstract value is an object with fields.

Often, abstract values are not clean mathematical objects. The concepts of customer, meeting, and item can all be defined loosely and in spec fields. In general, spec fields are different from representation fields.

5.10 ADTs in Java

Java Classes:

- Make operations of the abstract data type public methods
- Make other operations private
- Clients can only access the ADT operations

Java Interfaces:

- Clients only see the abstract data type operations, nothing else
- There can be multiple implementations with no code in common
- Cannot include creators or fields

Both classes and interfaces rely upon careful specifications. However, we prefer interface types over specific classes because they force programmers to decouple implementation from abstract data types.

When defensive programming, programmers must check what they are doing. Specifically, they must check:

- Precondition
- Postcondition
- Representation Invariant
- Loop Invariants

Defensive programming should be done statically - before execution. It works in more straightforward cases but can be difficult in general. Thus, adequate defensive programming motivates us to simplify and decompose our code!

It is also essential to check representation invariants dynamically via assertions. These are tests checked at runtime. Write these similarly to code - and they are not to be confused with JUnit methods such as `assertEquals`.

By default, Java runs with assertions disabled. Java-ea runs with assertions enabled. Always enable assertions during development, and disable them in production. Do not use assertions when the behavior of a code block is apparent, or when there could be unintended consequences for doing so.

Assertions can fail from:

- Precondition violations
- Errors in code from representation exposures, exceptions, bugs, etc.

- Unpredictable external problems such as running out of memory, missing files, memory corruption, connection failure, etc.

Good Java developers want to fail friendly and early to prevent harm in their programs. Their first goal is to add clarity to and prevent harm in the code base.

It is better to use preconditions instead of exceptions when verifying code. Java maintains a call stack of methods that are currently executing. When an exception is thrown, the control transfers to the nearest method with a matching catch block. If no catch block is found, use the top-level handler. Exceptions allow non-local error handling.

Return codes can cause problems when ignored. Exceptions are also typed, as are special values, but a method can throw multiple exceptions. Methods can only return one type. `Java.lang.Math` returns NaN for many standard math functions. NaNs are sticky, like `SomeType o = add(a, div(b, c));` it may be challenging to know where a NaN arose.

Generally, throw an exception when something should not happen. Return a special value when something unusual but expected can happen, and the client code can react to it. There are two distinct uses of exceptions:

1. External Failures (device failures) are unexpected and usually unrecoverable. If a condition is unchecked, the exception propagates up the call stack.

2. Special results are expected and can always be checked and handled locally. Therefore, take special action and continue computing.

Checked exceptions and unchecked exceptions should both be caught. Calls throwing checked exceptions are enclosed in a `try{}` block in a level above in the caller of the method. In that case, the current method must declare that it throws the exceptions so that the callers can make appropriate arrangements to handle the exception.

Checked exceptions are checked at compile time; the compiler checks that the exception is being handled there. Unchecked exceptions are not checked at compile time. Checked exceptions are tested at compile time; the method must either handle the exception, or specify that the exception will be thrown using the throws keyword. In Java Exceptions, checked and unchecked exceptions are helpful. Exceptions under error and RuntimeException classes are unchecked exceptions, and every other throwable is a checked exception. In C++, all exceptions are throwable.

Checked exceptions are for edge cases, and unchecked exceptions are for failures. In the library, there is no need to declare exceptions; in the client, there is no need to catch exceptions. Do not ignore exceptions! Empty exception brackets indicate lazy programming and bad design.

5.11 Review

In practice, always write a representation invariant. Write an abstraction when needed. A description is essential; make it precise, concise, and informal.

Use an exception when checking if the condition is feasible. It is used in a broad or unpredictable context. Using a precondition when checking would be prohibitive; for example, requiring that a list is sorted before operating on it would unnecessarily bloat the method. Use in a narrow context when calls can be checked.

Avoid preconditions when the caller may violate a precondition. The program can fail in an uninformative or dangerous way. We want the program to fail as early as possible. Use checked exceptions most of the time. Handle exceptions sooner rather than later.

Unchecked exceptions are better if the client writes code that avoids the exception. The exception reflects completely unanticipated failures. Otherwise, use a checked exception. It must be caught and handled, which prevents program defects. They used checked exceptions, which should be locally caught and handled. Checked exceptions that propagate over long distances indicate lousy design. If they are not caught, those exceptions generate program terminations.

6 Dafny

Dafny is a language designed to support the static verification of programs. It uses annotations to reason about code and generates a proof to show that the given code matches the annotations.

```
forall k: int :: 0 <= k < a.Length ==> 0 < a[k]
```

The example above shows that all elements of array k are greater than 0. Dafny can also prove that there are no runtime errors, null references, or anything else that Java would ordinarily time out or throw an exception over. Dafny is an interesting language.

In Dafny, the **method** is the smallest available unit for verification. Dafny methods are similar to the functions of other languages, except that they are imperative rather than functional.

Dafny's syntax is not based on any other language; it is truly unique! `:=` is the assignment operator, preconditions are denoted by the `requires` keyword, and postconditions employ the `ensures` keyword. Similarly to Java, Dafny uses assertions to verify program results.

Let us start with a simple HelloWorld program:

```
method Main() {
  print "Hello, World!\n";
  assert 10 < 2;   // this assertion fails
}
```

So far, Dafny looks like Java: methods enclosed in curly braces, clear function names spelled out in lower case, and semicolons at the end of lines. Notice, however, the lack of parenthesis and the immediate use of an assertion. Dafny relies heavily on assertions to prove the program's correctness.

Here is a more complex problem, a calculation of the nth number in the Fibonacci series:

```
function Fibonacci(n: int): int
  decreases n
{
  if n < 2 then n else Fibonacci(n-2) + Fibonacci(n-1)
}
```

The `if, then, else` statement, as well as the recursion in `Fibonacci(n-2)` and `Fibonacci(n-1)`, are familiar concepts. The `decreases` statement here is new; it is the representation invariant. In Dafny, representation invariants are compiled and run with the program so they can be verified together. A proper understanding of reasoning is therefore knowledge of Dafny and its syntax.

Here is another example of specifications at work within a Dafny program:

```
method  Abs(x: int) returns (x': int)
  ensures x' >= 0
  ensures (x < 0 && x' == -1 * x) || (x' == x)
{
  x' := x;
  if (x' < 0) { x' := x' *  -1; }
}

method Main()
{
   var v := Abs(3);
   assert v == 3;
   assert 0 < v;
   assert 0 <= v;
   v := Abs(0);
   assert v == 0;
}
```

The `Abs` method calculates the absolute value of a given integer `x` and returns it in the variable `x'`. The `Main()` method, featuring several assertions and variable assignments, tests the absolute value function. The only thing we have not seen in Dafny here is the `ensures` keyword, which defines the method's postcondition. If the method runs and those conditions are not met, Dafny will throw an error.

Finally, it might be useful to verify a binary search method implemented in Java:

```
predicate sorted(a: array?<int>)
   requires a != null
   reads a
{
   forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
method BinarySearch(a: array?<int>, value: int) returns (index: int)
   requires a != null && 0 <= a.Length && sorted(a)
   ensures 0 <= index ==> index < a.Length && a[index] == value
   ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
   var low, high := 0, a.Length;
   while low < high
      invariant 0 <= low <= high <= a.Length
      invariant forall i ::
         0 <= i < a.Length && !(low <= i < high) ==> a[i] != value
   {
      var mid := (low + high) / 2;
      if a[mid] < value {
         low := mid + 1;
      }
      else if value < a[mid] {
         high := mid;
      }
      else {
         return mid;
      }
   }
   return -1;
}
```

What syntax is reminiscent of Java? What is new? What logical assumptions and assertions does Dafny make here?

7 Testing

Testing is exactly what it sounds like - executing software to find bugs. Good testing creates a high probability of finding undiscovered bugs; successful testing discovers those bugs.

> Program testing can be used to show the presence of bugs, but never to show their absence. - Edsger Dijkstra 1970

We have seven rules for testing:

1. Exhaustive testing is usually not possible
2. A small number of modules usually contain most of the defects (aka defect clustering)
3. The Pesticide Paradox: repetitive use of the same pesticide builds more substantial bugs. Old tests will not find new bugs.
4. Testing shows the presence of defects, not the absence
5. The absence of Error fallacy: absence of evidence is not evidence of the absence of bugs
6. Test early and often
7. Testing is context-dependent; different data will reveal different bugs

Testing is complicated because rule number one is true; exhaustive testing is prohibitively costly. The critical problem is choosing a set of inputs for the test suite so it finishes quickly, but is large enough to validate the program.

7.1 Quality Assurance (QA)

Testing for *quality assurance* is self-explanatory: test code to ensure that it is up to spec. We can also use the following testing methods:

- static analysis (finding bugs without executing code (by reading the code))
- proofs of correctness (theorems)
- code reviews (peer review)
- software process (good development)

Nothing can guarantee software quality, so we have several different testing methods.

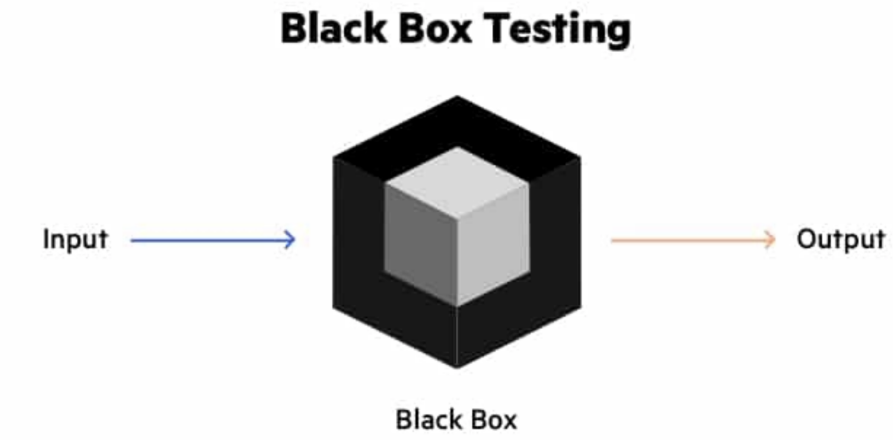
7.2 Testing Scope

There are a few different testing scopes: **Unit Testing** checks that each module does what it should. In Object-Oriented Programming (OOP), unit testing means *class testing*, since classes are easy to break into distinct modules. **Integration Testing** verifies that all parts work together as expected. Finally, **System Testing**checks whether the program satisfies functional requirements and works within a large software project.

7.3 Testing Strategies

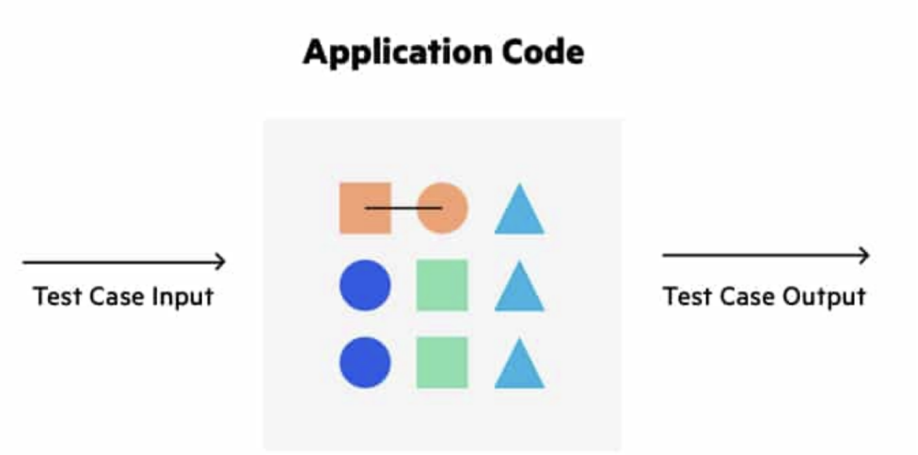
In addition to different scopes, there are a few different testing strategies, or points of view, we can use to cover our bases.

Black Box Testing requires developers to ignore the code and test according to specifications. Ask, "given some input, is the produced output correct according to the spec?" Black Box Tests are specification tests; equivalence partitioning, discussed below, is a black box testing strategy.



Black Box Testing is advantageous because it is robust concerning changes in implementation. Furthermore, it is independent of implementation, so test data does not need to be changed when the code is changed. Black Box Testing also allows for independent testing, where developers unfamiliar with the code base can contribute. Most importantly, black box tests can be written before their corresponding code; it is good practice to do so.

White Box Testing is the practice of covering all possible internal logical paths of the program. Choose paths with knowledge of the implementation to cover as many use cases (or input conditions) as possible. Unit testing, integration testing, and control-flow-based testing are white box testing strategies.



White Box Testing is advantageous because the test suite can cover the entire program. Thus, successful tests with high coverage imply few errors in the program. The focus is not on features described in the program, but on control-flow details, performance optimizations, and alternate algorithms for different cases.

The bottom line for white and black box testing is to *test early and often*. Write tests before code to catch bugs early, before they propagate through the system. Automate the process - thorough testing saves time in the long run. Be systematic about it, too; writing tests is an excellent way to understand the spec and, by extension, the code. Specifications can also be buggy - when finding bugs, write tests that verify their absence before eradicating them.

7.4 Equivalence Partitioning

Equivalence Partitioning divides input and output domains into equivalence classes. Equivalence partitioning is a good variety of black-box testing. **Equivalence Classes** are partitions of comparable data from which test cases can be derived. They usually apply to input data, and it is best to test each partition once.

Intuitively, values from different equivalence classes will drive the program through different logical paths. We do not formally define equivalence classes, though we know:

- input values have valid and invalid ranges
- we want to choose tests from the valid and invalid regions, and values near or at the boundaries of regions to cover edge cases

Note that we can only run tests on invalid arguments if the spec tells us what will happen for invalid data.

When partitioning values, carve them into sets of valid data. For example, suppose our specification says that valid input is an array of 4 to 24 numbers, and each number is a 3-digit positive integer. One equivalence class could be the size of an array:

- Classes are `n < 4, 4 <= n <= 24, n > 24`
- Chosen values: `3, 4, 5, 14, 23, 24, 25`

Another equivalence class could be integer values:

- Classes are `x < 100, 100 <= x <= 999, x > 999`
- Chosen values: `99, 100, 101, 500, 998, 999, 1000`

Because these data dimensions are unrelated, we need to test a range of array sizes and values within the array. We want to get arithmetic boundaries: the smallest, most significant, or zero values.

- We can also partition the output domain: the location of the value
  - Four classes: "first element", "last element", "middle element", "not found"

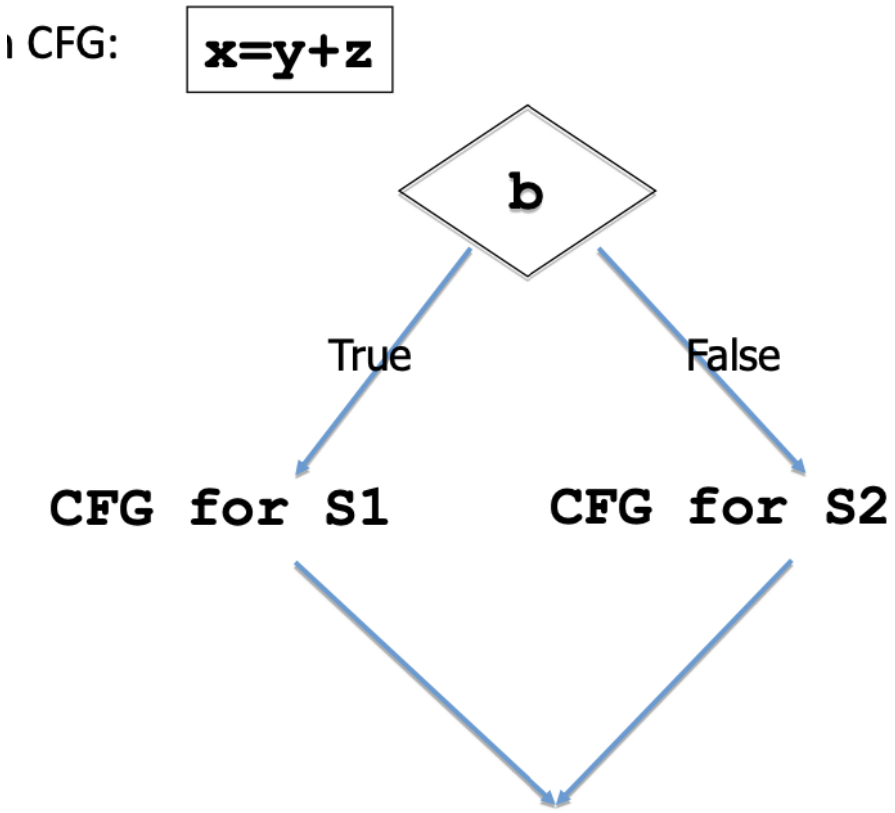
Array	Value	Output
Empty	5	-1
[7]	7	0
[7]	2	-1
[1,6,4,7,2]	1	0 (boundary, start)
[1,6,4,7,2]	4	2 (mid array)
[1,6,4,7,2]	2	4 (boundary, end)
[1,6,4,7,2]	3	-1

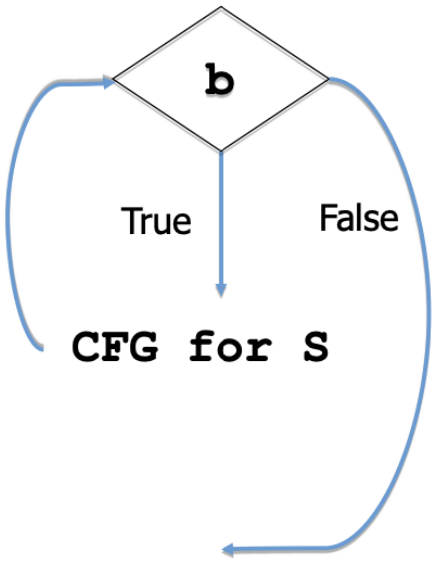
7.5 Control-Flow-Based Testing

Control Flow Based White Box Testing requires developers to extract a control flow graph (CFG) from the code; the test suite must cover essential elements of this control flow graph. The idea is to define a coverage target and ensure the test suite covers the target. The coverage target approximates the whole program.

In a CFG, each node represents a block of code. In that, there is an entry block and an exit block. Directed edges within the graph represent the control flow.

ı CFG:





7.6 Coverage

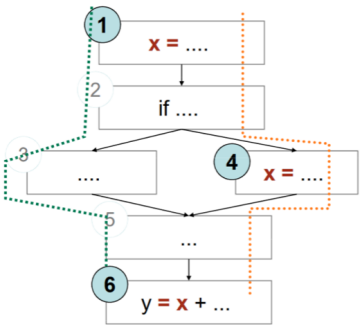
A traditional coverage target is **Statement Coverage**, which covers all statements (nodes) in the CFG. This way, we can ensure that even code that never gets executed will be tested.

Branch Coverage tests all the edges in a CFG, where true/false statements and if/else blocks create branches. In a loop, two branch edges correspond to its boolean conditions. In modern languages, branch coverage implies statement coverage.

The motivation for branch coverage is to test decision-making in the code. However, statement coverage does not imply branch coverage because it can hit all nodes without a top edge covering.

7.7 Definition-Use (DU) Pairs

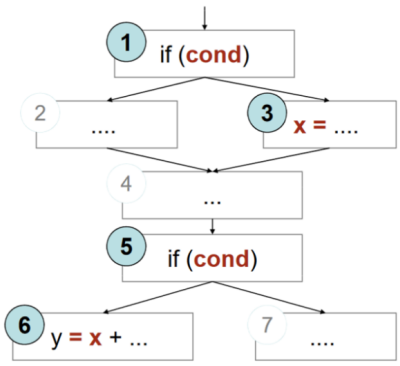
A **Def-Use Pair** consists of a definition and a variable where at least one path exists from the definition to the use. For example, `x = 1` could be a definition, and `y = x  3+` could be the use. A DU Path could be from the definition of a variable to the use of the same variable, with no other definition of the variable on the path. Loops can create infinite DU paths.



- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is "killed" (reassigned) at node 4
- (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

We want to test all DU pairs, all DU paths, and all definitions. Sometimes, there are impossible DU paths, so developers make up for the gaps in representation. Detecting infeasibility can be difficult; in practice, the goal is to get reasonable testing coverage. Aliasing, infeasible paths, and worst cases can hinder testing, but pragmatism is imperative here.

- Suppose cond doesn't change between 1 and 5
  - Or conditions could be different, but 1 implies 5
- (3, 6) is not a feasible DU path
- It is very difficult to find infeasible paths
- Infeasible paths are a problem
  - Difficult to find
  - Impossible to test



8 Identity Equality

The question of equality in Java is a loaded one: what does it *mean* to be equal? Java uses a reference model for class types. This means that the **Assignment Operator (=)** copies addresses to objects, not their values. Primitives are the exception; because they are often so small, the assignment operator copies their data. In Java, == tests for **Reference Equality** - whether or not two objects have the same address. `.equals()` tests for **Value Equality**, which tests if values are the same. Value equality is used more often; is it clear why?

If two objects are equal now, will they always be equal? For immutable objects, yes; for mutable objects, no.

Two objects are **Behaviorally Equivalent** if no sequence of operations can distinguish them. Behavioral equivalence is also called **Eternal Equality**. Two objects are **Observationally Equivalent** if no sequence of observer operations can distinguish them. Excluding mutators and ==, with observational equivalence, objects look the same because they have the same attributes.

8.1 Properties of Equality

Equality is:

- **Reflexive**: `a.equals(a);`
- **Symmetric**: `a.equals(b) -> b.equals(a)`
- **Transitive**: `a == b && b == c -> a == c`

In implementation, these all must hold. Equality is an equivalence relationship that can get complicated in the context of inheritance. For example, equal objects must have the same `hashCode()`.

Be very careful with elements of sets. Ideally, elements should be immutable, because immutable objects guarantee behavioral equivalence. That guarantee is a valuable feature; the Java specification for sets warns about using mutable objects as set elements.

8.2 Hash Function

A **Hash Function** maps an object to a number. The number is used as an index in a fixed-size table called a hash table for data storage. Hashing allows data storage and retrieval applications to access data quickly and nearly constantly per retrieval. However, developers beware: the worst case for hashing time can be extensive.

The implementation for `.hashCode` is consistent with `.equals()`; equal objects must have the same `hashCode`. `hashCode()` is used for bucketing in Hash

implementations - it places objects in the same place in memory. `.contains()` takes the element's hash code, then looks for the address where the hash code points. The value received from **hashCode()** is used to determine the address for, as an example, storing elements of a set or map.

By definition, if two objects are equal, their hash code must also be equal; if `equals()` is overridden, the way two objects are equated must also change.

8.3 Overloading vs. Overriding

Method **Overloading** is when two or more methods in the same class have the same name but different parameters. Overloading happens at compile time, meaning all objects will be seen as the Object type until the overload.

Method **Overriding** is when a derived class requires a different definition for an inherited method. Arguments stay the same in a method override. Overriding happens at runtime. When altering object equality, `.equals()` and `.hashCode()` need to be overridden.

An exciting thing about method overriding is that Java supports covariant return types for overridden methods. This means an overridden method may have a *more* specific return type than its original form. As long as the new return type is assignable to the overridden method's return type, covariance is allowed.

The overriding method cannot have a less restrictive access modifier than the overridden method. In addition, the return type must be the same as, or a subtype of, the return type declared in the superclass' overridden method. If the superclass method does not declare any exceptions, then the subclass' overridden method cannot declare a checked exception, though it can compare unchecked exceptions.

A final or static method cannot be overridden.

8.4 Implementation

Overriding `Object.equals()`:

```
// requires Object arg
// modifies none
// effects none
// throws none
// returns true if this == arg else false
// i.e., returns true if arg is the same ref as this
@Override
public boolean equals(Object obj) {
    if (obj == null) return false;
    if (obj == this) return true;
    if (!obj.getClass.equals(this.getClass)) return false;
    Class c = (Class) obj;
    // then compare attributes and return true if they work
}
```

Overriding `Object.hashCode()`:

```
// requires none
// modifies none
// effects none
// throws none
// returns true if this == arg else false
// i.e., returns true if arg is the same ref as this
@Override
public int hashCode() {
    int h = this.hashCode();
    // use attributes to figure it out
}
```

8.5 Abstract Classes

We subclass abstract classes because they cannot be instantiated. Therefore, there is no equality problem if the superclass cannot be instantiated.

With ADTs, we compare abstract values, not representations. Usually, many valid representations map to the same abstract value. If two concrete objects map to the same abstract value, they must be equal. In that way, a stronger representation invariant shrinks the domain of the Abstract Function and simplifies `.equals()`.

9 Parametric Polymorphism: Generics

Explicit Parametric Polymorphism is the technical term for generic usage in object-oriented languages.

Generics clarify code. Without generics, code has pseudo-generic containers. **Type Erasure** happens when all type parameters in generic types are within their bounds. Thus, the produced bytecode contains only ordinary classes, interfaces, and methods. Type casts can be inserted if necessary to improve type safety. Type erasure ensures that no new classes are created for parameterized types. Consequently, generics incur no runtime overhead.

The point of generic methods is that they can be called with arguments of different types. Based on the types of arguments passed to the generic method, the compiler handles each method call appropriately.

All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precede the method's return type. For example:

```
class MySet<T> {
     // rep invariant: non-null,
     // contains no duplicates List<T>
     List<T> theRep;
     T lastLookedUp;
}
MySet<String> s;
MySet<Integer> intSet;
MySet<int> i; // compiler error, does not autobox
```

A **Type Parameter**or **Type Variable** is an identifier that specifies a generic type name. Each type parameter section contains one or more type parameters separated by commas. Type parameters can be used to declare a method's return type and act as placeholders for argument types passed to the generic method, also known as **Actual Type** arguments.

A generic method's body is declared like that of any other method. Type parameters can represent only reference types, not primitive types (like int, double, and char). For example:

```
public class GenericMethodDemo {
// generic method printArray
     public static <E> void printArray( E[] inputArray ) {
         // Display array elements
         for(E element : inputArray) {
             System.out.print(element + " ");
         }
         System.out.println();
```

```
}

public static void main(String args[]) {
// Create arrays of Integer, Double, and Character
     Integer[] intArray = { 1, 2, 3, 4, 5 };
     Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
     Character[] charArray = {'H', 'E', 'L', 'L', 'O'};

     System.out.println("Array integerArray contains:");
     printArray(intArray); // pass an Integer array

     System.out.println("\nArray doubleArray contains:");
     printArray(doubleArray); // pass a Double array

     System.out.println("\nArray characterArray contains:");
     printArray(charArray); // pass a Character array
```

The compiler ensures type compatibility when using generics, so all code components work well together. Additionally, Java generics are explicitly polymorphic, so developers cannot assign illegal types to objects, like adding a cat object to an array list full of dog objects.

9.1 How does a method call execute?

For example, `x.foo(5);`

The compiler determines the **Compile Time Class**, what class to look in at compile time. Then, it determines the **Method Family**, or **Method Signature**: all methods in the class with the correct name, including inherited methods.

Then, the compiler looks for method overrides and return types that may be subtypes of the desired outcome. The types of the actual arguments must be the same, or be subtypes of the corresponding formal parameter type. For example, `5` has type `int` above.

The compiler keeps only methods that are accessible. A private method is not accessible to calls from outside the class. The compiler keeps track of the method's signature for runtime.

At run time, the compiler determines the runtime type of the receiver - `x` in this case. Then, it looks at the object in the heap to determine its runtime type. Next, it locates the method to invoke; starting with the runtime type, look for a method with the correct name and argument types found statically in the method family. The types of the actual arguments may be the same, or subtypes of the corresponding formal parameter type. If the method is found in

the runtime type, then invoke it. Otherwise, continue the search in the superclass of the runtime type. The compiler looks only at method-family members. Therefore, this procedure will always find a method to invoke, due to the checks done during static type checking.

9.2 Parametric Polymorphism

Overloading is typically referred to as "ad-hoc" polymorphism. In contrast, in **Parametric Polymorphism**, methods take type as a parameter. In python, there are no explicit type parameters; the code is polymorphic because it works with many different types by default.

Subtype Polymorphism is the ability to use a subclass where a superclass is expected. In **Implicit Parametric Polymorphism**, there are no explicit type parameters, yet the code works on many different types. How does implicit parametric polymorphism differ from subtype polymorphism? Subtype polymorphism is static, and implicit parametric polymorphism is dynamic. Subtype polymorphism requires declared types for iterable types, whereas implicit polymorphism does not. Subtype polymorphism guarantees that every subclass of the declared type will be iterable, while implicit polymorphism does not. For example, Java subtyping guarantees that when `intersect` is called, the runtime iterable types will implement `iteration`. With subtype polymorphism, the compiler prevents calling `intersect(2,2)`, as an example.

Python, Perl, Scheme, and other dynamic languages use implicit parametric polymorphism with no explicit type parameters. As a result, the code works with many different types. Usually, there is a single copy of the code, and all type checking is delayed until runtime. The code works if the arguments are of a type expected by the code. If not, the code issues a type error at runtime.

C++, Java, and C# use **Explicit Parametric Polymorphism**, which has explicit type parameters for variables and methods. Explicit Parametric Polymorphism is also known as **Genericity**. As an example, C++ has templates. Usually, templates are implemented by creating multiple copies of the generic code, one for each concrete type argument, and then compiling. Problems arise if the "wrong" type argument is instantiated. In that case, the C++ compiler returns long, cryptic error messages referring to the generic (templated) code in the Standard Template Library (STL).

Luckily, Java generics work differently from C++ templates – there is more type-checking on generic code. Object Oriented languages usually have both subtype and explicit parametric polymorphism, referred to as either generics or templates.

9.3 Using Generics

Defining a Generic Class looks something like below:

```
class MySet<T> {
     // rep invariant: non-null,
     // contains no duplicates
     List<T> theRep;
     T lastLookedUp;
}

MySet<String> s;
MySet<Integer> intSet;
MySet<int> i; // compiler error, doesn't autobox
```

The convention is to use a one-letter name, such as `T`, for type. The client supplies type arguments to instantiate generic classes.

9.4 Bounded Types

Bounded Types are generic types that extend preexisting classes. An **Upper Bound Type Argument** can be any of its bounded subtypes. The upper bound on a type parameter has three effects:

1. **Restricted Instantiation**: The upper bound restricts the set of types that can be used for instantiation of the generic type. `<T extends Number>` means `T` can be instantiated as a `Number` or an `Integer`.
2. **Access to Non-Static Members**: The upper bound gives access to all upper-bound public non-static methods and fields. In the class `Box<T extends Number?>`, we can invoke all public non-static methods defined in class `Number`, such as `intValue()`.
3. **Type Erasure**: The leftmost upper bound is used for type erasure and replaces the type parameter in bytecode. For example, in class `Box<T extends Number>`, all occurrences of `T` would be replaced by the upper bound `Number`.

Why doesn't Java allow Lower Bounded Type Parameters? Two reasons:

- **Access to Non-Static Members**: A lower-type parameter bound does not give access to any particular methods beyond those inherited from class Objects. For example, in `Box<T super Number>`, the supertypes of `Number` have nothing in common, except that they are reference types and, therefore, subtypes of `Object`.
- **Type Erasure**: Replaces all occurrences of the type variable T by type Object and not by its lower bound. The lower bound would have the same effect as a "no bound."

The bottom line is that `<T super Type>` does not get anyone anywhere and leads to confusion.

9.5 Java Wildcards

A **Wildcard** is an anonymous type variable. Use `?` when using a type variable exactly once. `?` appears at the instantiation site of the generic - the use site, as opposed to the declaration site. Thus, `<? extends E>`, but not `<E extends ?>`, is valid. The purpose of the wildcard is to make a library more flexible and easier to use by allowing limited subtyping. Wildcards limit the kind of operations that instances of a class can perform.

Wildcards appear at the instantiations of generic objects. There is also `<? super E>`. Intuitively, `<? extends E>` makes sense here: the syntax `<? extends E>` means "type E or a subtype of E."

`? extends` introduces covariant subtyping. For example, `Apple` is a subtype of `Fruit`, `Strawberry` is a subtype of `Fruit`, and `List<Apple>` is a subtype of `List<? extends Fruit>`:

```
List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples;
fruits.add(new Strawberry());
```

The above code, as is, will not compile. `fruits.add(new Fruit());` won't compile either.

```
Object o;
Number n;
Integer i;
PositiveInteger p; // extends Integer


List<? extends Integer> lei;
```

First, which of these is legal?

~~`lei = new ArrayList<Object>();`~~

~~`lei = new ArrayList<Number>();`~~

```
lei = new ArrayList<Integer>();
lei = new ArrayList<PositiveInteger>();
lei = new ArrayList<NegativeInteger>();
```

Which of these is legal?

~~`lei.add(o);`~~

~~`lei.add(n);`~~

~~`lei.add(i);`~~

~~`lei.add(p);`~~

```
lei.add(null);
o = lei.get(0);
n = lei.get(0);
i = lei.get(0);
```

~~`p = lei.get(0);`~~

The type declaration `List<? extends Integer>` means that every `List< type>` such that type extends `Integer`, is a subtype of `List<? extends Integer>`, and so can be used where `List<? extends Integer>` is expected. Covariant subtyping must be restricted to immutable lists.

Use `<? extends T>` when getting values from a producer, and use `<? super T>` when adding values into a consumer.
For example, `<T> void copy(List<? super T> DST, List<? extends T> src)`.
Use the `? extends` wildcard to retrieve objects from data structures. Use the `? super` wildcard to put objects in a data structure. Wildcards are optional when both reading and writing.

All type arguments become java objects, or type bound, when compiled. This is due to backward compatibility with outdated bytecode. At runtime, all generic instantiations have the same type, so developers cannot use `instanceOf` to find type arguments at that point. So what happens to `equals()` on elements of generic type?

Early versions of Java did not include generics. If arrays were not covariant, something like boolean `equalArrays(Object[] a1, Object [] a2);` would only work with Objects. There would have to be a different method for every type combination. When generics were introduced, they were purposefully not made covariant to prevent strange subtype phenomena from happening. When writing a generic class, start by writing a concrete class. Make sure it is correct using reasoning and testing. Then, think about writing a second, more concrete class, with different types. How would the original class need to be modified to accommodate generics? With practice, it gets easier to start with generics.

9.6 Contravariance

A programming language can have features that may support the following subtyping rules:

- **Covariant**: A feature that allows a subtype to replace a supertype, like the Java covariant return type.
- **Contravariant**: A feature that allows a supertype to replace a subtype, such as contravariant argument types; these apply to Overloads in Java rather than overrides
- **Bivariant**: A feature that is both covariant and contravariant.
- **Invariant**: A feature that does not allow any of the above substitutions.

9.7 Review

Do not use raw types because Eclipse complains. Do not use private final Collection, and do not use raw iterators. Raw types lose type safety. Eliminate warnings as much as possible, and add comments. Use lists instead of arrays because lists are more type-safe than arrays.

Use generic types rather than fixed types. This makes code more flexible, and making classes generic without affecting client code is straightforward. Generic methods are helpful in the same way.

Use bounded wildcards to enhance flexibility.

10 Design Patterns

A design pattern is a generic solution to a recurring design problem. Design patterns seek to solve common programming problems with a few common solutions:

- **Encapsulation**: hiding implementation components in a layer below what the client can access.
- **Subclassing**: Making a "wrapper" that implements something else to reduce repetition.
- **Iteration**: The object does traversals to access all members of a collection.
- **Exceptions**: Errors in one part of the code should be handled elsewhere, so the language is structured for throwing and catching exceptions.
- **Generics**: Generic types allow users to use objects of any type in data structures.

SOLID Design Principles:

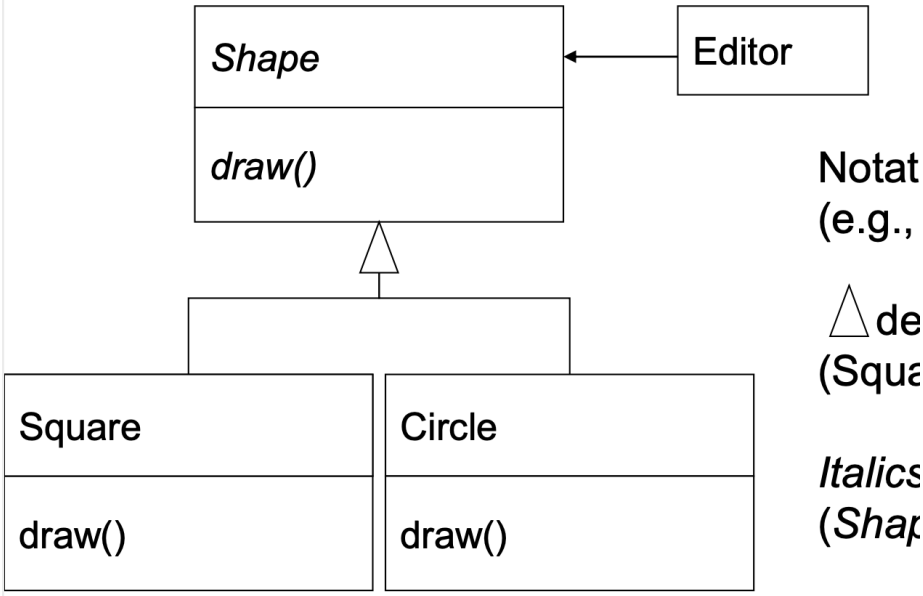
- **Single Responsibility Principle**: A class should have only one responsibility.
- **Open-Closed Principle**: A class's behavior can be extended without modifying it.
- **Liscov Substitution Principle**: The derived classes must be substitutable for their base classes.
- **Interface Segregation Principle**: Many client-specific interfaces are better than one general-purpose interface.
- **Dependency Inversion Principle**: Implementation should depend on abstractions, not concrete implementations.

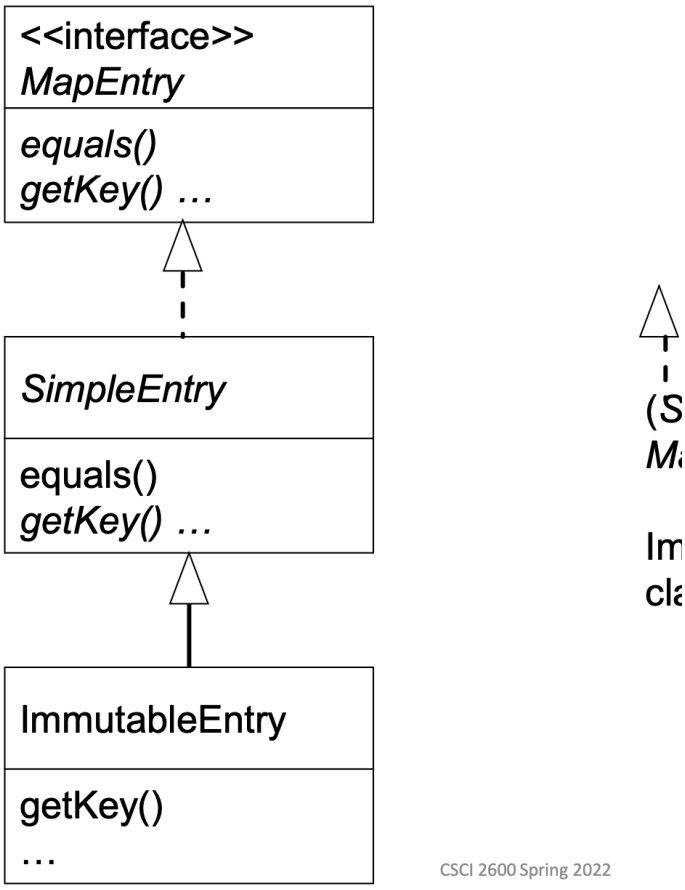
Design patterns do not solve all design problems, but they can increase code clarity and improve modularity. There are three different kinds of design patterns: creational patterns, structural patterns, and behavioral patterns. Before we can go into detail about definitions, uses, and examples of each design pattern, UML diagrams need discussion. These give context to code structures and will be used in code examples.

10.1 Unified Modeling Language (UML)

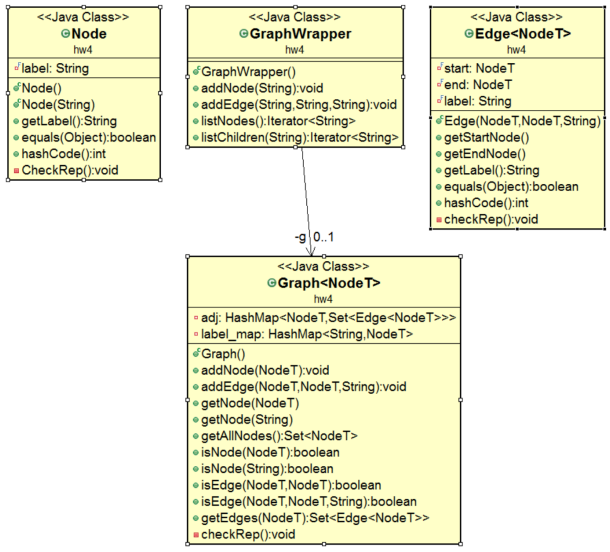
UML class diagrams show classes and their relationships, including inheritance and composition, attributes, and operations. Additionally, UML sequence diagrams can show the dynamics of a system. UML is the standard "language" of object-oriented modeling and design. Here is some UML notation:

- Boxes are classes
- △ on arrows denote inheritance with subclasses pointing to superclasses
- *italics* denote abstract types
- dotted lines between classes denote interface inheritance





UML associations often represent a composition relationship; objects of one class enclose objects of another. Thus, UML is often used to model abstract concepts and their relationships. Attributes and associations correspond to ADT operations. UML can express designs - there is a close correspondence to implementation. Attributes and associates correspond to representation fields.



10.2 Creational Patterns

10.2.1 Abstract Factory

The **Abstract Factory** pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

For example, the `AbstractSoupFactory` defines the method names and return types to make various kinds of soup.

The `BostonConcreteSoupFactory` and the `HonoluluConcreteSoupFactory` both extend the `AbstractSoupFactory`.

An object can be defined as an `AbstractSoupFactory` and instantiated as either a `BostonConcreteSoupFactory` (BCSF) or a `HonoluluConcreteSoupFactory` (HCSF). Both BCSF or HCSF have the `makeFishChowder` method, and both return a `FishChowder` type class. However, the BCSF returns a `FishChowder` subclass of `BostonFishChowder`, while the HCSF returns a `FishChowder` subclass of HonoluluFishChowder.

Code for all three classes is featured below, starting with the `AbstractSoupFactory`:

```
abstract class AbstractSoupFactory {
    String factoryLocation;
    public String getFactoryLocation() {
        return factoryLocation;
    }
```

```
    public ChickenSoup makeChickenSoup() {
        return new ChickenSoup();
    }
    public ClamChowder makeClamChowder() {
        return new ClamChowder();
    }
    public FishChowder makeFishChowder() {
        return new FishChowder();
    }
    public Minnestrone makeMinnestrone() {
        return new Minnestrone();
    }
    public PastaFazul makePastaFazul() {
        return new PastaFazul();
    }
    public TofuSoup makeTofuSoup() {
        return new TofuSoup();
    }
    public VegetableSoup makeVegetableSoup() {
        return new VegetableSoup();
    }
}
```

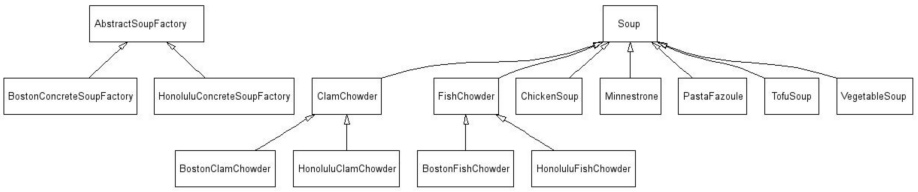
BCSF and HCSF are next. Notice how, in the extended classes, there are different clam and fish chowders:

```
class BostonConcreteSoupFactory extends AbstractSoupFactory {
    public BostonConcreteSoupFactory() {
        factoryLocation = "Boston";
    }
    public ClamChowder makeClamChowder() {
        return new BostonClamChowder();
    }
    public FishChowder makeFishChowder() {
        return new BostonFishChowder();
    }
}

class BostonClamChowder extends ClamChowder {
    public BostonClamChowder() {
        soupName = "QuahogChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Quahogs");
        soupIngredients.add("1 cup corn");
        soupIngredients.add("1/2 cup heavy cream");
```

```
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup potato chips");
    }
}

class BostonFishChowder extends FishChowder {
    public BostonFishChowder() {
        soupName = "ScrodFishChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Scrod");
        soupIngredients.add("1 cup corn");
        soupIngredients.add("1/2 cup heavy cream");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup potato chips");
    }
}



class HonoluluConcreteSoupFactory extends AbstractSoupFactory {
    public HonoluluConcreteSoupFactory() {
        factoryLocation = "Honolulu";
    }
    public ClamChowder makeClamChowder() {
       return new HonoluluClamChowder();
    }
    public FishChowder makeFishChowder() {
       return new HonoluluFishChowder();
    }
}

class HonoluluClamChowder extends ClamChowder {
    public HonoluluClamChowder() {
        soupName = "PacificClamChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Pacific Clams");
        soupIngredients.add("1 cup pineapple chunks");
        soupIngredients.add("1/2 cup coconut milk");
        soupIngredients.add("1/4 cup SPAM");
        soupIngredients.add("1/4 cup taro chips");
    }
}

class HonoluluFishChowder extends FishChowder {
    public HonoluluFishChowder() {
        soupName = "OpakapakaFishChowder";
```

```
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Opakapaka Fish");
        soupIngredients.add("1 cup pineapple chunks");
        soupIngredients.add("1/2 cup coconut milk");
        soupIngredients.add("1/4 cup SPAM");
        soupIngredients.add("1/4 cup taro chips");
    }
}
```

The UML diagram for this `AbstractSoupFactory` system is pictured below:



10.2.2 Builder

The **Builder** pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.

In this example, the abstract `SoupBuffetBuilder` defines the methods necessary to create a `SoupBuffet`.

`BostonSoupBuffetBuilder` and the `HonoluluSoupBuffetBuilder` both extend the `SoupBuffetBuilder`.

An object can be defined as an `SoupBuffetBuilder` and instantiated as either a `BostonSoupBuffetBuilder` (BSBB) or a `HonoluluSoupBuffetBuilder` (HSBB). BSBB and HSBB both have a `buildFishChowder` method, and both return a `FishChowder` type class. However, the BSBB returns a `FishChowder` subclass of `BostonFishChowder`, while the HSBB returns a `FishChowder` subclass of `HonoluluFishChowder`.

Code for all three classes is featured below, starting with `SoupBuffetBuilder`:

```
abstract class SoupBuffetBuilder {
    SoupBuffet soupBuffet;

    public SoupBuffet getSoupBuffet() {
        return soupBuffet;
    }
```

```
    public void buildSoupBuffet() {
        soupBuffet = new SoupBuffet();
    }

    public abstract void setSoupBuffetName();

    public void buildChickenSoup() {
        soupBuffet.chickenSoup = new ChickenSoup();
    }
    public void buildClamChowder() {
        soupBuffet.clamChowder = new ClamChowder();
    }
    public void buildFishChowder() {
        soupBuffet.fishChowder = new FishChowder();
    }
    public void buildMinnestrone() {
        soupBuffet.minnestrone = new Minnestrone();
    }
    public void buildPastaFazul() {
        soupBuffet.pastaFazul = new PastaFazul();
    }
    public void buildTofuSoup() {
        soupBuffet.tofuSoup = new TofuSoup();
    }
    public void buildVegetableSoup() {
        soupBuffet.vegetableSoup = new VegetableSoup();
    }
}

class BostonSoupBuffetBuilder extends SoupBuffetBuilder {
    public void buildClamChowder() {
       soupBuffet.clamChowder = new BostonClamChowder();
    }
    public void buildFishChowder() {
       soupBuffet.fishChowder = new BostonFishChowder();
    }

    public void setSoupBuffetName() {
       soupBuffet.soupBuffetName = "Boston Soup Buffet";
    }
}

class BostonClamChowder extends ClamChowder {
    public BostonClamChowder() {
        soupName = "QuahogChowder";
```

```
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Quahogs");
        soupIngredients.add("1 cup corn");
        soupIngredients.add("1/2 cup heavy cream");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup potato chips");
    }
}

class BostonFishChowder extends FishChowder {
    public BostonFishChowder() {
        soupName = "ScrodFishChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Scrod");
        soupIngredients.add("1 cup corn");
        soupIngredients.add("1/2 cup heavy cream");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup potato chips");
    }
}

class HonoluluSoupBuffetBuilder extends SoupBuffetBuilder {
    public void buildClamChowder() {
        soupBuffet.clamChowder = new HonoluluClamChowder();
    }
    public void buildFishChowder() {
        soupBuffet.fishChowder = new HonoluluFishChowder();
    }

    public void setSoupBuffetName() {
        soupBuffet.soupBuffetName = "Honolulu Soup Buffet";
    }
}

class HonoluluClamChowder extends ClamChowder {
    public HonoluluClamChowder() {
        soupName = "PacificClamChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Pacific Clams");
        soupIngredients.add("1 cup pineapple chunks");
        soupIngredients.add("1/2 cup coconut milk");
        soupIngredients.add("1/4 cup SPAM");
        soupIngredients.add("1/4 cup taro chips");
    }
}
```

```
class HonoluluFishChowder extends FishChowder {
    public HonoluluFishChowder() {
        soupName = "OpakapakaFishChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Opakapaka Fish");
        soupIngredients.add("1 cup pineapple chunks");
        soupIngredients.add("1/2 cup coconut milk");
        soupIngredients.add("1/4 cup SPAM");
        soupIngredients.add("1/4 cup taro chips");
    }
}
```

We've also included the `Soup.java` and `SoupBuffer.java` helper class to supplement the classes we already have:

```
import java.util.ArrayList;
import java.util.ListIterator;

abstract class Soup
{
   ArrayList soupIngredients = new ArrayList();
   String soupName;

   public String getSoupName()
   {
       return soupName;
   }

   public String toString()
   {
        StringBuffer stringOfIngredients = new StringBuffer(soupName);
        stringOfIngredients.append(" Ingredients: ");
        ListIterator soupIterator = soupIngredients.listIterator();
        while (soupIterator.hasNext())
        {
            stringOfIngredients.append((String)soupIterator.next());
        }
        return stringOfIngredients.toString();
   }
}

class ChickenSoup extends Soup
{
    public ChickenSoup()
    {
```

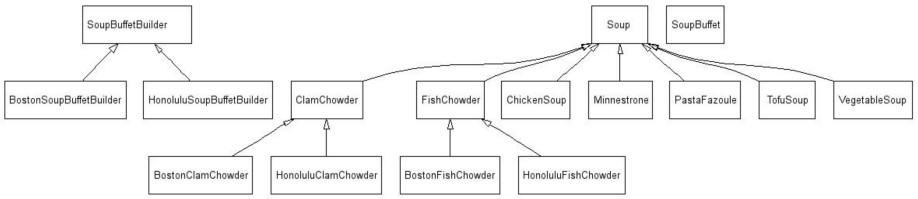
```
        soupName = "ChickenSoup";
        soupIngredients.add("1 Pound diced chicken");
        soupIngredients.add("1/2 cup rice");
        soupIngredients.add("1 cup bullion");
        soupIngredients.add("1/16 cup butter");
        soupIngredients.add("1/4 cup diced carrots");
    }
}

class ClamChowder extends Soup
{
    public ClamChowder()
    {
        soupName = "ClamChowder";
        soupIngredients.add("1 Pound Fresh Clams");
        soupIngredients.add("1 cup fruit or vegetables");
        soupIngredients.add("1/2 cup milk");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup chips");
    }
}

class FishChowder extends Soup
{
    public FishChowder()
    {
        soupName = "FishChowder";
        soupIngredients.add("1 Pound Fresh fish");
        soupIngredients.add("1 cup fruit or vegetables");
        soupIngredients.add("1/2 cup milk");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup chips");
    }
}

class Minnestrone extends Soup
{
    public Minnestrone()
    {
        soupName = "Minestrone";
        soupIngredients.add("1 Pound tomatos");
        soupIngredients.add("1/2 cup pasta");
        soupIngredients.add("1 cup tomato juice");
    }
}
```

```
class PastaFazul extends Soup
{
    public PastaFazul()
    {
        soupName = "Pasta Fazul";
        soupIngredients.add("1 Pound tomatos");
        soupIngredients.add("1/2 cup pasta");
        soupIngredients.add("1/2 cup diced carrots");
        soupIngredients.add("1 cup tomato juice");
    }
}

class TofuSoup extends Soup
{
    public TofuSoup()
    {
        soupName = "Tofu Soup";
        soupIngredients.add("1 Pound tofu");
        soupIngredients.add("1 cup carrot juice");
        soupIngredients.add("1/4 cup spirolena");
    }
}

class VegetableSoup extends Soup
{
    public VegetableSoup()
    {
        soupName = "Vegetable Soup";
        soupIngredients.add("1 cup bullion");
        soupIngredients.add("1/4 cup carrots");
        soupIngredients.add("1/4 cup potatoes");
    }
}

class SoupBuffet {
   String soupBuffetName;

   ChickenSoup chickenSoup;
   ClamChowder clamChowder;
   FishChowder fishChowder;
   Minnestrone minnestrone;
   PastaFazul pastaFazul;
   TofuSoup tofuSoup;
   VegetableSoup vegetableSoup;

   public String getSoupBuffetName() {
```

```
    return soupBuffetName;
}
public void setSoupBuffetName(String soupBuffetNameIn) {
   soupBuffetName = soupBuffetNameIn;
}

public void setChickenSoup(ChickenSoup chickenSoupIn) {
    chickenSoup = chickenSoupIn;
}
public void setClamChowder(ClamChowder clamChowderIn) {
    clamChowder = clamChowderIn;
}
public void setFishChowder(FishChowder fishChowderIn) {
   fishChowder = fishChowderIn;
}
public void setMinnestrone(Minnestrone minnestroneIn) {
   minnestrone = minnestroneIn;
}
public void setPastaFazul(PastaFazul pastaFazulIn) {
    pastaFazul = pastaFazulIn;
}
public void setTofuSoup(TofuSoup tofuSoupIn) {
    tofuSoup = tofuSoupIn;
}
public void setVegetableSoup(VegetableSoup vegetableSoupIn) {
    vegetableSoup = vegetableSoupIn;
}

public String getTodaysSoups() {
     StringBuffer stringOfSoups = new StringBuffer();
     stringOfSoups.append(" Today's Soups!  ");
     stringOfSoups.append(" Chicken Soup: ");
     stringOfSoups.append(this.chickenSoup.getSoupName());
     stringOfSoups.append(" Clam Chowder: ");
     stringOfSoups.append(this.clamChowder.getSoupName());
     stringOfSoups.append(" Fish Chowder: ");
     stringOfSoups.append(this.fishChowder.getSoupName());
     stringOfSoups.append(" Minnestrone: ");
     stringOfSoups.append(this.minestrone.getSoupName());
     stringOfSoups.append(" Pasta Fazul: ");
     stringOfSoups.append(this.pastaFazul.getSoupName());
     stringOfSoups.append(" Tofu Soup: ");
     stringOfSoups.append(this.tofuSoup.getSoupName());
     stringOfSoups.append(" Vegetable Soup: ");
     stringOfSoups.append(this.vegetableSoup.getSoupName());
     return stringOfSoups.toString();
```

```
    }
}
```

The UML diagram for this `SoupBuffetBuilder` system is pictured below:



10.2.3 Factory Method

The **Factory** pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. The factory method lets a class defer instantiation to subclasses.

For example, the `SoupFactoryMethod` defines the `makeSoupBuffet` method, which returns a `SoupBuffet` object. The `SoupFactoryMethod` also defines the methods needed in creating the `SoupBuffet`.

The `BostonSoupFactoryMethodSubclass` and the `HonoluluSoupFactoryMethodSubclass` both extend the `SoupFactoryMethod`. An object can be defined as an `SoupFactoryMethod` and instantiated as either a `BostonSoupFactoryMethodSubclass` (BSFMS) or a `HonoluluSoupFactoryMethodSubclass` (HSFMS).

Both BSFMS and HSFMS override `SoupFactoryMethod`'s `makeFishChowder` method. The BSFMS returns a `SoupBuffet` with a `FishChowder` subclass of `BostonFishChowder`, while the HSFMS returns a `SoupBuffet` with a `FishChowder` subclass of `HonoluluFishChowder`.

Example code for all the classes in the soup factory system is featured below:

```
lass SoupFactoryMethod {
    public SoupFactoryMethod() {}

    public SoupBuffet makeSoupBuffet() {
        return new SoupBuffet();
    }

    public ChickenSoup makeChickenSoup() {
        return new ChickenSoup();
    }
    public ClamChowder makeClamChowder() {
```

```
        return new ClamChowder();
    }
    public FishChowder makeFishChowder() {
        return new FishChowder();
    }
    public Minnestrone makeMinnestrone() {
        return new Minnestrone();
    }
    public PastaFazul makePastaFazul() {
        return new PastaFazul();
    }
    public TofuSoup makeTofuSoup() {
        return new TofuSoup();
    }
    public VegetableSoup makeVegetableSoup() {
        return new VegetableSoup();
    }

    public String makeBuffetName() {
        return "Soup Buffet";
    }
}

class BostonSoupFactoryMethodSubclass extends SoupFactoryMethod {
    public String makeBuffetName() {
        return "Boston Soup Buffet";
    }
    public ClamChowder makeClamChowder() {
        return new BostonClamChowder();
    }
    public FishChowder makeFishChowder() {
        return new BostonFishChowder();
    }
}

class BostonClamChowder extends ClamChowder {
    public BostonClamChowder() {
        soupName = "QuahogChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Quahogs");
        soupIngredients.add("1 cup corn");
        soupIngredients.add("1/2 cup heavy cream");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup potato chips");
    }
}
```

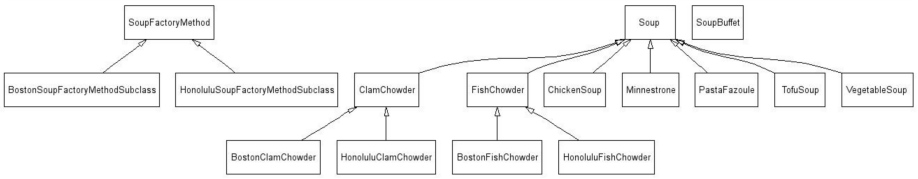
```
class BostonFishChowder extends FishChowder {
    public BostonFishChowder() {
        soupName = "ScrodFishChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Scrod");
        soupIngredients.add("1 cup corn");
        soupIngredients.add("1/2 cup heavy cream");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup potato chips");
    }
}
```

Featured below is the code for some additional subclasses:

```
class HonoluluSoupFactoryMethodSubclass extends SoupFactoryMethod {
    public String makeBuffetName() {
        return "Honolulu Soup Buffet";
    }
    public ClamChowder makeClamChowder() {
        return new HonoluluClamChowder();
    }
    public FishChowder makeFishChowder() {
        return new HonoluluFishChowder();
    }
}

class HonoluluClamChowder extends ClamChowder {
    public HonoluluClamChowder() {
        soupName = "PacificClamChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Pacific Clams");
        soupIngredients.add("1 cup pineapple chunks");
        soupIngredients.add("1/2 cup coconut milk");
        soupIngredients.add("1/4 cup SPAM");
        soupIngredients.add("1/4 cup taro chips");
    }
}

class HonoluluFishChowder extends FishChowder {
    public HonoluluFishChowder() {
        soupName = "OpakapakaFishChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Opakapaka Fish");
        soupIngredients.add("1 cup pineapple chunks");
```

```
        soupIngredients.add("1/2 cup coconut milk");
        soupIngredients.add("1/4 cup SPAM");
        soupIngredients.add("1/4 cup taro chips");
    }
}
```

This factory method example also uses the `Soup.java` subclass. Pictured below is the UML diagram for the factory system:



10.2.4 Prototype

With the **Prototype** pattern, developers can specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype. In other words, new objects can be created by cloning prototypical objects.

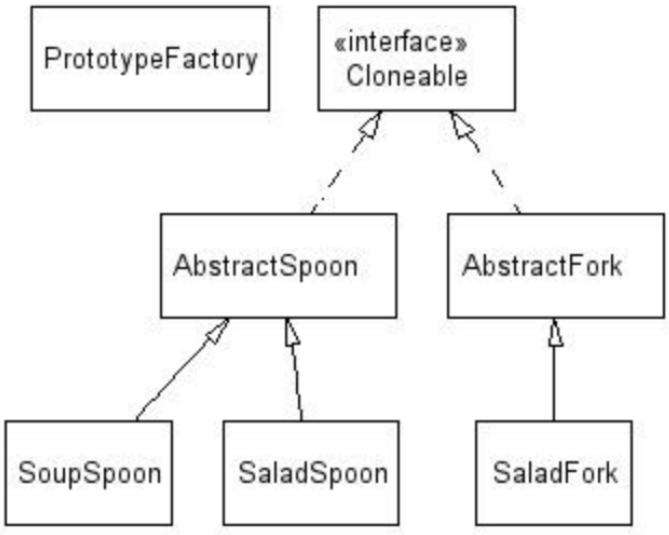
For example, there are several prototype objects (`AbstractSpoon.java`, `AbstractFork.java`) and a factory object that make up this system:

```
public class PrototypeFactory {
    AbstractSpoon prototypeSpoon;
    AbstractFork prototypeFork;

    public PrototypeFactory(AbstractSpoon spoon, AbstractFork fork) {
       prototypeSpoon = spoon;
       prototypeFork = fork;
   }

   public AbstractSpoon makeSpoon() {
       return (AbstractSpoon)prototypeSpoon.clone();
   }
   public AbstractFork makeFork() {
       return (AbstractFork)prototypeFork.clone();
   }
}
```

Concrete prototype objects (`SoupSpoon.java`, `SaladSpoon.java`, `SaladFork.java`) that extend the abstract prototypes are featured below:

```
public class SoupSpoon extends AbstractSpoon {
   public SoupSpoon() {
       setSpoonName("Soup Spoon");
   }
}

public class SaladSpoon extends AbstractSpoon {
   public SaladSpoon() {
       setSpoonName("Salad Spoon");
   }
}

public class SaladFork extends AbstractFork {
   public SaladFork() {
       setForkName("Salad Fork");
   }
}
```

Notice how these classes are all minimal compared to the abstract classes they extend. The difference in scale is a symptom of modularity and inheritance, and keeps code readable and organized. The UML for this prototype example follows:



10.2.5 Singleton

The **Singleton** design pattern ensures that a class only has one instance and provides a global point of access to it.

For example, the `SingleSpoon` class holds one instance of `SingleSpoon in private static SingleSpoon the`
The `SingleSpoon` class determines the spoons availability using `private static boolean theSpoonIsAvaila`
The first time `SingleSpoon.getTheSpoon()` is called, it creates an instance of a `SingleSpoon`. The `SingleSpoon` cannot be distributed again until it is returned with `SingleSpoon.returnTheSpoon()`.

If a developer were to create a spoon "pool," they would have the same basic logic as shown. However, multiple spoons would be distributed. The variable `theSpoon` would hold an array or collection of spoons. The variable `theSpoonIsAvaialable` would become a counter of the number of available spoons.

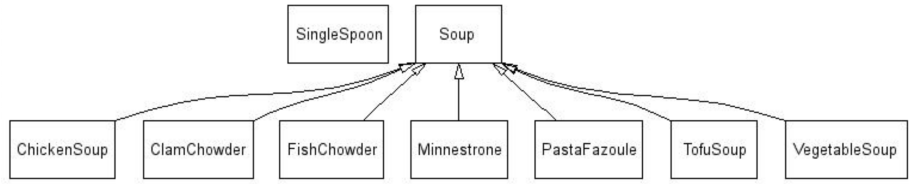
Please also note that this example *is not thread-safe*. The `getTheSpoon()` method should be synchronized to make it thread-safe.

Example code is shown below:

```
public class SingleSpoon
{
   private Soup soupLastUsedWith;

   private static SingleSpoon theSpoon;
   private static boolean theSpoonIsAvailable = true;

   private SingleSpoon() {}

   public static SingleSpoon getTheSpoon() {
       if (theSpoonIsAvailable) {
           if (theSpoon == null) {
               theSpoon = new SingleSpoon();
           }
           theSpoonIsAvailable = false;
           return theSpoon;
       } else {
           return null;
           //spoon not available,
           //  could throw an error or return null (as shown)
       }
   }

   public String toString() {
       return "Behold the glorious single spoon!";
```

```
    }

    public static void returnTheSpoon() {
        theSpoon.cleanSpoon();
        theSpoonIsAvailable = true;
    }

    public Soup getSoupLastUsedWith() {
        return this.soupLastUsedWith;
    }
    public void setSoupLastUsedWith(Soup soup) {
        this.soupLastUsedWith = soup;
    }

    public void cleanSpoon() {
        this.setSoupLastUsedWith(null);
    }
}
```

The UML diagram for this example is featured below:



10.3 Structural Patterns

10.3.1 Adapter

With the **Adapter** design pattern, developers convert the interface of a class into another interface that clients expect. An adapter lets classes work together that otherwise could not because of incompatible interfaces.

For example, the `TeaBall` class takes in an instance of `LooseLeafTea`. The `TeaBall` class uses the `steepTea` method from `LooseLeafTea` and adapts it to provide the `steepTeaInCup` method, which the `TeaCup` class requires. See `TeaBag.java` below:

```
public class TeaBag {
    boolean teaBagIsSteeped;

    public TeaBag() {
```

```
        teaBagIsSteeped = false;
    }

    public void steepTeaInCup() {
        teaBagIsSteeped = true;
        System.out.println("tea bag is steeping in cup");
    }
}
```

Next, we have an adapter object that creates an interface for the adaptee. See `TeaBall.java` below:

```
public class TeaBall extends TeaBag {
    LooseLeafTea looseLeafTea;

    public TeaBall(LooseLeafTea looseLeafTeaIn) {
        looseLeafTea = looseLeafTeaIn;
        teaBagIsSteeped = looseLeafTea.teaIsSteeped;
    }

    public void steepTeaInCup() {
        looseLeafTea.steepTea();
        teaBagIsSteeped = true;
    }
}
```

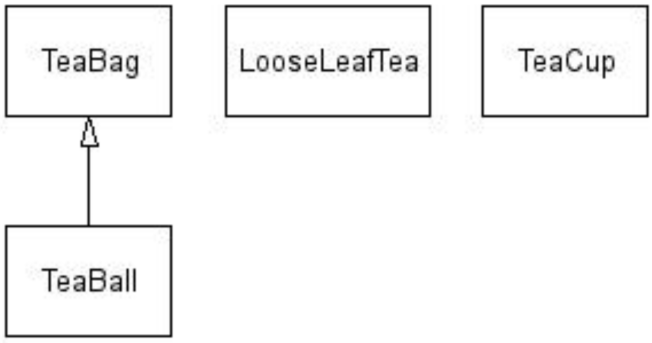
Code for the adaptee object, `LooseLeafTea.java`, is below:

```
public class LooseLeafTea {
    boolean teaIsSteeped;

    public LooseLeafTea() {
        teaIsSteeped = false;
    }

    public void steepTea() {
        teaIsSteeped = true;
        System.out.println("tea is steeping");
    }
}
```

Finally, the `TeaCup.java` class accepts the adapter and adaptee to allow users to steep tea bags, as well as tea balls, is featured below:

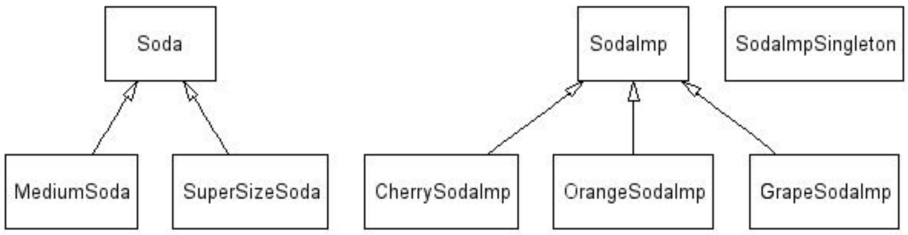
```
public class TeaCup {
   public void steepTeaBag(TeaBag teaBag) {
       teaBag.steepTeaInCup();
   }
}
```

Notice how the root class is uncomplicated and has exactly as many extensions as the functionality requires. Thus, the Singleton pattern shows why we use design patterns: they make life easier. The UML diagram for this example is shown below:



10.3.2 Bridge

The **Bridge** pattern decouples an abstraction from its implementation so that the two can vary independently. The UML diagram below illustrates the Bridge structure well:



Notice how the abstract soda class and the actual soda implementation are distinctly separated. The code abstract `Soda.java` class, along with its abstract extending classes `MediumSoda.java` and `SuperSizeSoda.java`, is featured below:

```
public abstract class Soda {
   SodaImp sodaImp;
```

```
   public void setSodaImp() {
       this.sodaImp = SodaImpSingleton.getTheSodaImp();
   }
   public SodaImp getSodaImp() {
       return this.sodaImp;
   }

   public abstract void pourSoda();
}

public class MediumSoda extends Soda {
   public MediumSoda() {
       setSodaImp();
   }

   public void pourSoda() {
       SodaImp sodaImp = this.getSodaImp();
       for (int i = 0; i < 2; i++) {
           System.out.print("...glug...");
           sodaImp.pourSodaImp();
       }
       System.out.println(" ");
   }
}

public class SuperSizeSoda extends Soda {
   public SuperSizeSoda() {
       setSodaImp();
   }

   public void pourSoda() {
       SodaImp sodaImp = this.getSodaImp();
       for (int i = 0; i < 5; i++) {
           System.out.print("...glug...");
           sodaImp.pourSodaImp();
       }
       System.out.println(" ");
   }
}
```

The very small `SodaImp.java` class connects these abstract classes to a concrete object, and `CherrySodaImp.java`, `GrapeSodaImp.java`, `OrangeSodaImp.java` all extend the concrete "bridge" so there are several different, fleshed out, compatible objects to work with. There is also an additional Singleton class `SodaImpSingleton.java` to keep all instances clear:

```
public abstract class SodaImp {
   public abstract void pourSodaImp();
}

public class CherrySodaImp extends SodaImp {
   CherrySodaImp() {}

   public void pourSodaImp() {
       System.out.println("Yummy Cherry Soda!");
   }
}

public class GrapeSodaImp extends SodaImp {
   GrapeSodaImp() {}

   public void pourSodaImp() {
       System.out.println("Delicious Grape Soda!");
   }
}

public class OrangeSodaImp extends SodaImp {
   OrangeSodaImp() {}

   public void pourSodaImp() {
       System.out.println("Citrusy Orange Soda!");
   }
}

public class SodaImpSingleton {
    private static SodaImp sodaImp;

    public SodaImpSingleton(SodaImp sodaImpIn) {
        this.sodaImp = sodaImpIn;
    }

    public static SodaImp getTheSodaImp() {
        return sodaImp;
    }
}
```

10.3.3 Composite

The **Composite** design pattern allows developers to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly by assembling groups of objects with the same signature. Consider the `TeaBags.java` example once

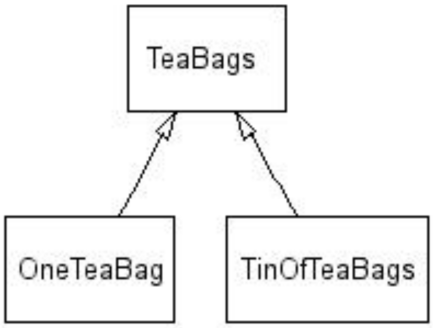
again, with the composite of all tea bags being `TinOfTeaBags.java`:

```
import java.util.LinkedList;
import java.util.ListIterator;

public abstract class TeaBags {
   LinkedList teaBagList;
   TeaBags parent;
   String name;

   public abstract int countTeaBags();

   public abstract boolean add(TeaBags teaBagsToAdd);
   public abstract boolean remove(TeaBags teaBagsToRemove);
   public abstract ListIterator createListIterator();

   public void setParent(TeaBags parentIn) {
       parent = parentIn;
   }
   public TeaBags getParent() {
      return parent;
   }

   public void setName(String nameIn) {
       name = nameIn;
   }
   public String getName() {
       return name;
   }
}

public class OneTeaBag extends TeaBags {
    public OneTeaBag(String nameIn) {
        this.setName(nameIn);
    }

    public int countTeaBags() {
        return 1;
    }

    public boolean add(TeaBags teaBagsToAdd) {
        return false;
    }
    public boolean remove(TeaBags teaBagsToRemove) {
        return false;
```

```
    }
    public ListIterator createListIterator() {
        return null;
    }
}

public class TinOfTeaBags extends TeaBags {
   public TinOfTeaBags(String nameIn) {
       teaBagList = new LinkedList();
       this.setName(nameIn);
   }

   public int countTeaBags() {
       int totalTeaBags = 0;
       ListIterator listIterator = this.createListIterator();
       TeaBags tempTeaBags;
       while (listIterator.hasNext()) {
           tempTeaBags = (TeaBags)listIterator.next();
           totalTeaBags += tempTeaBags.countTeaBags();
       }
       return totalTeaBags;
   }

   public boolean add(TeaBags teaBagsToAdd) {
       teaBagsToAdd.setParent(this);
       return teaBagList.add(teaBagsToAdd);
   }

   public boolean remove(TeaBags teaBagsToRemove) {
       ListIterator listIterator =
           this.createListIterator();
       TeaBags tempTeaBags;
       while (listIterator.hasNext()) {
           tempTeaBags = (TeaBags)listIterator.next();
           if (tempTeaBags == teaBagsToRemove) {
               listIterator.remove();
               return true;
           }
       }
       return false;
   }

   public ListIterator createListIterator() {
       ListIterator listIterator = teaBagList.listIterator();
       return listIterator;
   }
```

```
}
```

The UML diagram for this example is pictured below:



10.3.4 Decorator

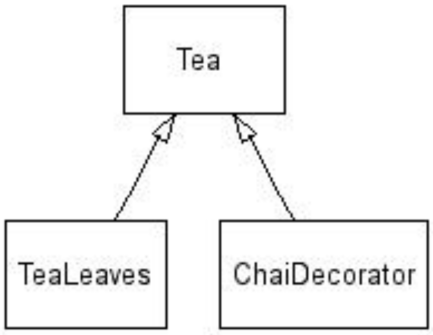
The **Decorator** pattern allows developers to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

For example, a decorator class (`ChaiDecorator.java` here) takes in a decoratee class (`TeaLeaves.java` in this case), both of which extend the same abstract class (`Tea.java`) and adds functionality to the decoratee. See the example code below:

```
import java.util.ArrayList;
import java.util.ListIterator;

public abstract class Tea {
   boolean teaIsSteeped;

   public abstract void steepTea();
}

public class ChaiDecorator extends Tea {
    private Tea teaToMakeChai;
    private ArrayList chaiIngredients = new ArrayList();

    public ChaiDecorator(Tea teaToMakeChai) {
        this.addTea(teaToMakeChai);
        chaiIngredients.add("bay leaf");
        chaiIngredients.add("cinnamon stick");
```

```
        chaiIngredients.add("ginger");
        chaiIngredients.add("honey");
        chaiIngredients.add("soy milk");
        chaiIngredients.add("vanilla bean");
    }

    private void addTea(Tea teaToMakeChaiIn) {
        this.teaToMakeChai = teaToMakeChaiIn;
    }

    public void steepTea() {
        this.steepChai();
    }

    public void steepChai() {
        teaToMakeChai.steepTea();
        this.steepChaiIngredients();
        System.out.println("tea is steeping with chai");
    }

    public void steepChaiIngredients() {
        ListIterator listIterator = chaiIngredients.listIterator();
        while (listIterator.hasNext()) {
            System.out.println(((String)(listIterator.next())) +
                                        " is steeping");
        }
        System.out.println("chai ingredients are steeping");
    }
}

public class TeaLeaves extends Tea {
   public TeaLeaves() {
       teaIsSteeped = false;
   }

   public void steepTea() {
       teaIsSteeped = true;
       System.out.println("tea leaves are steeping");
   }
}
```

The UML diagram for this relationship is featured below:



10.3.5 Facade

In the **Facade** pattern, one class has a method that calls several other classes to achieve complex goals. The pattern is useful for providing a unified interface to a set of interfaces in a subsystem; a facade defines a higher-level interface that makes the subsystem easier to use.

This design pattern uses no polymorphism, so use this class if extending classes is difficult, impossible, or clunky.

Below are four classes: `FacadeCuppaMaker.java`, the facade in the system, and the three classes it interfaces: `FacadeTeaCup.java`, `FacadeWater.java`, and `FacadeTeaBag.java`. Take note of the method `FacadeCuppaMaker.java` uses to instantiate all these objects:

```
public class FacadeCuppaMaker {
   boolean teaBagIsSteeped;

   public FacadeCuppaMaker() {
       System.out.println(
         "FacadeCuppaMaker ready to make you a cuppa!");
   }

   public FacadeTeaCup makeACuppa() {
       FacadeTeaCup cup = new FacadeTeaCup();
       FacadeTeaBag teaBag = new FacadeTeaBag();
       FacadeWater water = new FacadeWater();
       cup.addFacadeTeaBag(teaBag);
       water.boilFacadeWater();
       cup.addFacadeWater(water);
       cup.steepTeaBag();
       return cup;
   }
```

```
}

public class FacadeTeaCup {
   boolean teaBagIsSteeped;
   FacadeWater facadeWater;
   FacadeTeaBag facadeTeaBag;

   public FacadeTeaCup() {
       setTeaBagIsSteeped(false);
       System.out.println("behold the beautiful new tea cup");
   }

   public void setTeaBagIsSteeped(boolean isTeaBagSteeped) {
       teaBagIsSteeped = isTeaBagSteeped;
   }
   public boolean getTeaBagIsSteeped() {
       return teaBagIsSteeped;
   }

   public void addFacadeTeaBag(FacadeTeaBag facadeTeaBagIn) {
       facadeTeaBag = facadeTeaBagIn;
       System.out.println("the tea bag is in the tea cup");
   }

   public void addFacadeWater(FacadeWater facadeWaterIn) {
       facadeWater = facadeWaterIn;
       System.out.println("the water is in the tea cup");
   }

   public void steepTeaBag() {
       if ( (facadeTeaBag != null) &&
             ( (facadeWater != null) &&
               (facadeWater.getWaterIsBoiling()) )
          ) {
          System.out.println("the tea is steeping in the cup");
          setTeaBagIsSteeped(true);
       } else {
          System.out.println("the tea is not steeping in the cup");
          setTeaBagIsSteeped(false);
       }
   }

   public String toString() {
       if (this.getTeaBagIsSteeped()) {
           return ("A nice cuppa tea!");
       } else {
```

```
            String tempString = new String("A cup with ");
            if (facadeWater != null) {
                 if (facadeWater.getWaterIsBoiling()) {
                     tempString = (tempString + "boiling water ");
                 } else {
                     tempString = (tempString + "cold water ");
                 }
            } else {
                 tempString = (tempString + "no water ");
            }

            if (facadeTeaBag != null) {
                 tempString = (tempString + "and a tea bag");
            } else {
                 tempString = (tempString + "and no tea bag");
            }
            return tempString;
        }

    }
}

public class FacadeWater {
    boolean waterIsBoiling;

    public FacadeWater() {
        setWaterIsBoiling(false);
        System.out.println("behold the wonderous water");
    }

    public void boilFacadeWater() {
        setWaterIsBoiling(true);
        System.out.println("water is boiling");
    }

    public void setWaterIsBoiling(boolean isWaterBoiling) {
        waterIsBoiling = isWaterBoiling;
    }
    public boolean getWaterIsBoiling() {
        return waterIsBoiling;
    }
}

public class FacadeTeaBag {
    public FacadeTeaBag() {
        System.out.println("behold the lovely tea bag");
```

```
   }
}
```

A UML diagram detailing the interfacing of these objects is pictured below. Here, they are all separate classes that do not extend one another, so there are no arrows:

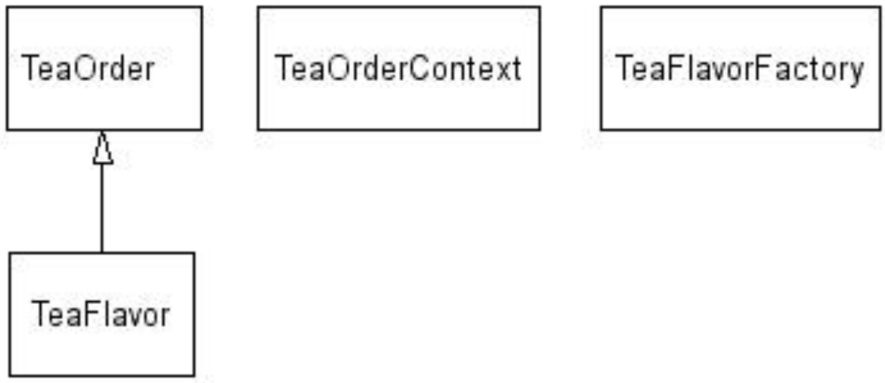
10.3.6 Flyweight

The **Flyweight** pattern uses sharing to efficiently support large numbers of fine-grained objects. A class's reusable and variable parts are broken into two classes: the flyweight (`TeaOrder.java`) and the context (`TeaOrderContext java`), to save resources. This example uses tea and several helper classes to show the relationship between objects:

```
public abstract class TeaOrder {
    public abstract void serveTea(TeaOrderContext teaOrderContext);
}

public class TeaOrderContext {
   int tableNumber;

   TeaOrderContext(int tableNumber) {
       this.tableNumber = tableNumber;
   }

   public int getTable() {
       return this.tableNumber;
   }
}

public class TeaFlavor extends TeaOrder {
    String teaFlavor;

    TeaFlavor(String teaFlavor) {
        this.teaFlavor = teaFlavor;
    }

    public String getFlavor() {
        return this.teaFlavor;
```

```
    }

    public void serveTea(TeaOrderContext teaOrderContext) {
        System.out.println("Serving tea flavor " +
                           teaFlavor +
                          " to table number " +
                           teaOrderContext.getTable());
    }
}

public class TeaFlavorFactory {
   TeaFlavor[] flavors = new TeaFlavor[10];
     //no more than ten flavors can be made
   int teasMade = 0;

   public TeaFlavor getTeaFlavor(String flavorToGet) {
       if (teasMade > 0) {
           for (int i = 0; i < teasMade; i++) {
               if (flavorToGet.equals((flavors[i]).getFlavor())) {
                   return flavors[i];
               }
           }
       }
       flavors[teasMade] = new TeaFlavor(flavorToGet);
       return flavors[teasMade++];
   }

   public int getTotalTeaFlavorsMade() {return teasMade;}
}
```

This example also contains a factory for simplicity. The UML for this example is exhibited below:



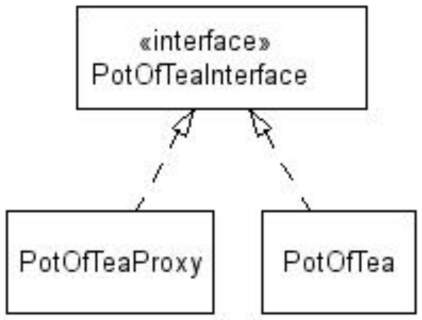
10.3.7 Proxy

The **Proxy** design pattern provides a surrogate or placeholder for another object to control access to the main class. The Proxy, here `PotOfTeaProxy.java`, controls the creation of and access to objects in another the subject, here `PotOfTea.java`:

```
//PotOfTeaInterface will ensure that the proxy
//  has the same methods as its real subject
public interface PotOfTeaInterface {
   public void pourTea();
}

public class PotOfTeaProxy implements PotOfTeaInterface {
   PotOfTea potOfTea;

   public PotOfTeaProxy() {}

   public void pourTea() {
       potOfTea = new PotOfTea();
       potOfTea.pourTea();
   }
}

public class PotOfTea implements PotOfTeaInterface {
   public PotOfTea() {
       System.out.println("Making a pot of tea");
   }

   public void pourTea() {
       System.out.println("Pouring tea");
```

```
   }
}
```

A UML diagram detailing the proxy relationship is below:



10.4 Behavioral Patterns

10.4.1 Chain of Responsibility

The **Chain of Responsibility** pattern helps developers avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Instead, use the pattern to chain the receiving objects together and pass the request along until an object handles it. A method called in one class will move up a class hierarchy until a method is found that can properly handle the call.

In this example, we use a DvD class hierarchy to illustrate the strata necessary for deploying the Chain of Responsibility:

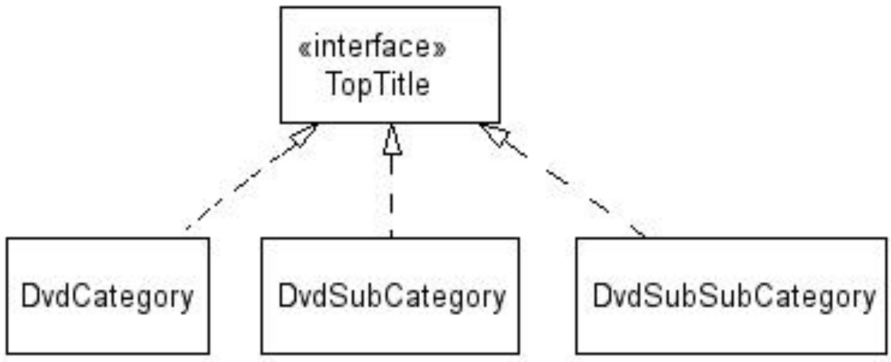
```
public interface TopTitle {
   public String getTopTitle();

   public String getAllCategories();
}

public class DvdCategory implements TopTitle {
   private String category;
   private String topCategoryTitle;

   public DvdCategory(String category) {
        this.setCategory(category);
   }
```

```
    public void setCategory(String categoryIn) {
        this.category = categoryIn;
    }
    public String getCategory() {
        return this.category;
    }
    public String getAllCategories() {
        return getCategory();
    }

    public void setTopCategoryTitle(String topCategoryTitleIn) {
        this.topCategoryTitle = topCategoryTitleIn;
    }
    public String getTopCategoryTitle() {
        return this.topCategoryTitle;
    }

    public String getTopTitle() {
        return this.topCategoryTitle;
    }
}

public class DvdSubCategory implements TopTitle {
    private String subCategory;
    private String topSubCategoryTitle;
    private DvdCategory parent;

    public DvdSubCategory(DvdCategory dvdCategory, String subCategory) {
         this.setSubCategory(subCategory);
         this.parent = dvdCategory;
    }

    public void setSubCategory(String subCategoryIn) {
        this.subCategory = subCategoryIn;
    }
    public String getSubCategory() {
        return this.subCategory;
    }
    public void setCategory(String categoryIn) {
        parent.setCategory(categoryIn);
    }
    public String getCategory() {
        return parent.getCategory();
    }
    public String getAllCategories() {
        return (getCategory() + "/" + getSubCategory());
```

```
    }

    public void setTopSubCategoryTitle(String topSubCategoryTitleIn) {
        this.topSubCategoryTitle = topSubCategoryTitleIn;
    }
    public String getTopSubCategoryTitle() {
        return this.topSubCategoryTitle;
    }
    public void setTopCategoryTitle(String topCategoryTitleIn) {
        parent.setTopCategoryTitle(topCategoryTitleIn);
    }
    public String getTopCategoryTitle() {
        return parent.getTopCategoryTitle();
    }

    public String getTopTitle() {
        if (null != getTopSubCategoryTitle()) {
            return this.getTopSubCategoryTitle();
        } else {
            System.out.println("no top title in Category/SubCategory " +
                               getAllCategories());
            return parent.getTopTitle();
        }
    }
}

public class DvdSubSubCategory implements TopTitle {
    private String subSubCategory;
    private String topSubSubCategoryTitle;
    private DvdSubCategory parent;

    public DvdSubSubCategory(DvdSubCategory dvdSubCategory,
                             String subCategory) {
         this.setSubSubCategory(subCategory);
         this.parent = dvdSubCategory;
    }

    public void setSubSubCategory(String subSubCategoryIn) {
        this.subSubCategory = subSubCategoryIn;
    }
    public String getSubSubCategory() {
        return this.subSubCategory;
    }
    public void setSubCategory(String subCategoryIn) {
        parent.setSubCategory(subCategoryIn);
    }
```

```
    public String getSubCategory() {
        return parent.getSubCategory();
    }
    public void setCategory(String categoryIn) {
        parent.setCategory(categoryIn);
    }
    public String getCategory() {
        return parent.getCategory();
    }
    public String getAllCategories() {
        return (getCategory() + "/" +
                getSubCategory() + "/" +
                getSubSubCategory());}

    public void setTopSubSubCategoryTitle(
      String topSubSubCategoryTitleIn) {
       this.topSubSubCategoryTitle = topSubSubCategoryTitleIn;
    }
    public String getTopSubSubCategoryTitle() {
        return this.topSubSubCategoryTitle;
    }
    public void setTopSubCategoryTitle(
      String topSubCategoryTitleIn) {
        parent.setTopSubCategoryTitle(topSubCategoryTitleIn);
    }
    public String getTopSubCategoryTitle() {
        return parent.getTopSubCategoryTitle();
    }
    public void setTopCategoryTitle(String topCategoryTitleIn) {
        parent.setTopCategoryTitle(topCategoryTitleIn);
    }
    public String getTopCategoryTitle() {
        return parent.getTopCategoryTitle();
    }

    public String getTopTitle() {
        if (null != getTopSubSubCategoryTitle()) {
            return this.getTopSubSubCategoryTitle();
        } else {
            System.out.println(
              "no top title in Category/SubCategory/SubSubCategory " +
               getAllCategories());
            return parent.getTopTitle();
        }
    }
}
```

The hierarchical structure of these classes and their relationship is well illustrated in the following UML diagram:



10.4.2 Command Pattern

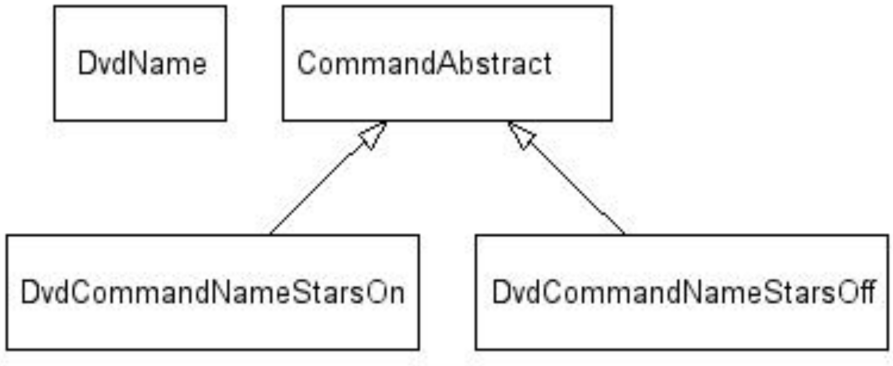
The **Command Pattern** design pattern encapsulates a request as an object, letting programmers parameterize clients with different requests, queue or log requests, and support undoable operations. In this pattern, a commanding object encapsulates everything needed to execute a method in a receiver object.

In this example, `CommandAbstract.java` is the commander, and `DvdName.java` is the receiver.

```
public abstract class CommandAbstract {
   public abstract void execute();
}

public class DvdName {
   private String titleName;

   public DvdName(String titleName) {
       this.setTitleName(titleName);
   }

   public final void setTitleName(String titleNameIn) {
       this.titleName = titleNameIn;
   }
   public final String getTitleName() {
       return this.titleName;
   }
```

```
   public void setNameStarsOn() {
       this.setTitleName(this.getTitleName().replace(' ','*'));
   }
   public void setNameStarsOff() {
       this.setTitleName(this.getTitleName().replace('*',' '));
   }

   public String toString() {
       return ("DVD: " + this.getTitleName());
   }
}

public class DvdCommandNameStarsOn extends CommandAbstract {
   private DvdName dvdName;

   public DvdCommandNameStarsOn(DvdName dvdNameIn) {
       this.dvdName = dvdNameIn;
   }
   public void execute() {
       this.dvdName.setNameStarsOn();
   }
}

public class DvdCommandNameStarsOff extends CommandAbstract {
   private DvdName dvdName;

   public DvdCommandNameStarsOff(DvdName dvdNameIn) {
       this.dvdName = dvdNameIn;
   }
   public void execute() {
       this.dvdName.setNameStarsOff();
   }
}
```

A UML diagram detailing this relationship is featured below:



10.4.3 Interpreter

Given a language, the **Interpreter** pattern defines a representation for its grammar in addition to an interpreter that uses the representation to interpret sentences in the language. A context defines a macro language and syntax, parsing input into objects which perform the correct operations on behalf of the client.

```
import java.util.StringTokenizer;
import java.util.ArrayList;
import java.util.ListIterator;

public class DvdInterpreterClient {
   DvdInterpreterContext dvdInterpreterContext;

   public DvdInterpreterClient(
     DvdInterpreterContext dvdInterpreterContext) {
       this.dvdInterpreterContext = dvdInterpreterContext;
   }

   // expression syntax:
   // show title | actor [for actor | title ]
   public String interpret(String expression) {
       StringBuffer result = new StringBuffer("Query Result: ");

       String currentToken;
       StringTokenizer expressionTokens =
         new StringTokenizer(expression);

       char mainQuery = ' ';
       char subQuery = ' ';
       boolean forUsed = false;
       String searchString = null;
```

```
boolean searchStarted = false;
boolean searchEnded = false;

while (expressionTokens.hasMoreTokens())
{
    currentToken = expressionTokens.nextToken();
    if (currentToken.equals("show")) {
        continue;
        //show in all queries, not used
    } else if (currentToken.equals("title")) {
        if (mainQuery == ' ') {
            mainQuery = 'T';
        } else {
            if ((subQuery == ' ') && (forUsed)) {
                subQuery = 'T';
            }
        }
    } else if (currentToken.equals("actor")) {
        if (mainQuery == ' ') {
            mainQuery = 'A';
        } else {
            if ((subQuery == ' ') && (forUsed)) {
                subQuery = 'A';
            }
        }
    } else if (currentToken.equals("for")) {
        forUsed = true;
    } else if ((searchString == null) &&
               (subQuery != ' ') &&
               (currentToken.startsWith("<"))) {
          searchString = currentToken;
          searchStarted = true;
          if (currentToken.endsWith(">")) {
              searchEnded = true;
          }
    } else if ((searchStarted) && (!searchEnded)) {
        searchString = searchString + " " + currentToken;
        if (currentToken.endsWith(">")) {
            searchEnded = true;
        }
    }
}

if (searchString != null) {
    searchString =
      searchString.substring(1,(searchString.length() - 1));
```

```
            //remove <>
        }

        DvdAbstractExpression abstractExpression;

        switch (mainQuery) {
            case 'A' :  {
                switch (subQuery) {
                    case 'T' : {
                        abstractExpression =
                          new DvdActorTitleExpression(searchString);
                        break;
                    }
                    default : {
                        abstractExpression =
                          new DvdActorExpression();
                        break;
                    }
                }
                break;
            }
            case 'T' : {
                switch (subQuery) {
                    case 'A' : {
                        abstractExpression =
                          new DvdTitleActorExpression(searchString);
                        break;
                    }
                    default : {
                        abstractExpression = new DvdTitleExpression();
                        break;
                    }
                }
                break;
            }

            default : return result.toString();
        }


        result.append(
          abstractExpression.interpret(dvdInterpreterContext));

        return result.toString();
    }
}
```

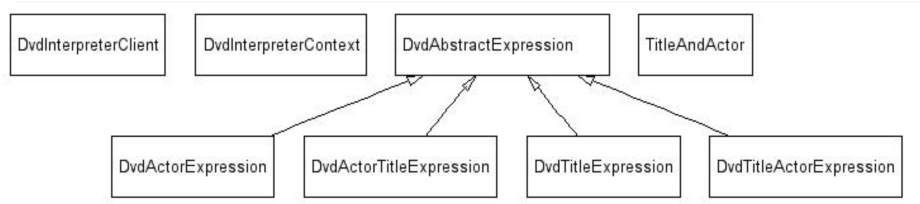
```
public class DvdInterpreterContext {
   private ArrayList titles = new ArrayList();
   private ArrayList actors = new ArrayList();
   private ArrayList titlesAndActors = new ArrayList();

   public void addTitle(String title) {
       titles.add(title);
   }
   public void addActor(String actor) {
       actors.add(actor);
   }
   public void addTitleAndActor(TitleAndActor titleAndActor) {
       titlesAndActors.add(titleAndActor);
   }

   public ArrayList getAllTitles() {
       return titles;
   }
   public ArrayList getAllActors() {
       return actors;
   }
   public ArrayList getActorsForTitle(String titleIn) {
       ArrayList actorsForTitle = new ArrayList();
       TitleAndActor tempTitleAndActor;
       ListIterator titlesAndActorsIterator =
         titlesAndActors.listIterator();
       while (titlesAndActorsIterator.hasNext()) {
           tempTitleAndActor =
             (TitleAndActor)titlesAndActorsIterator.next();
           if (titleIn.equals(tempTitleAndActor.getTitle())) {
               actorsForTitle.add(tempTitleAndActor.getActor());
           }
       }
       return actorsForTitle;
   }
   public ArrayList getTitlesForActor(String actorIn) {
       ArrayList titlesForActor = new ArrayList();
       TitleAndActor tempTitleAndActor;
       ListIterator actorsAndTitlesIterator =
         titlesAndActors.listIterator();
       while (actorsAndTitlesIterator.hasNext()) {
           tempTitleAndActor =
             (TitleAndActor)actorsAndTitlesIterator.next();
           if (actorIn.equals(tempTitleAndActor.getActor())) {
               titlesForActor.add(tempTitleAndActor.getTitle());
```

```
            }
        }
        return titlesForActor;
    }
}

public abstract class DvdAbstractExpression {
    public abstract String interpret(
      DvdInterpreterContext dvdInterpreterContext);
}

public class DvdActorExpression extends DvdAbstractExpression {
    public String interpret(DvdInterpreterContext dvdInterpreterContext) {
        ArrayList actors = dvdInterpreterContext.getAllActors();
        ListIterator actorsIterator = actors.listIterator();
        StringBuffer titleBuffer = new StringBuffer("");
        boolean first = true;
        while (actorsIterator.hasNext()) {
            if (!first) {
                titleBuffer.append(", ");
            } else {
                first = false;
            }
            titleBuffer.append((String)actorsIterator.next());
        }
        return titleBuffer.toString();
    }
}

public class DvdActorTitleExpression extends DvdAbstractExpression {
    String title;

    public DvdActorTitleExpression(String title) {
        this.title = title;
    }

    public String interpret(DvdInterpreterContext dvdInterpreterContext) {
        ArrayList actorsAndTitles =
            dvdInterpreterContext.getActorsForTitle(title);
        ListIterator actorsAndTitlesIterator =
            actorsAndTitles.listIterator();
        StringBuffer actorBuffer = new StringBuffer("");
        boolean first = true;
        while (actorsAndTitlesIterator.hasNext()) {
            if (!first) {
                actorBuffer.append(", ");
```

```
            } else {
                first = false;
            }
            actorBuffer.append((String)actorsAndTitlesIterator.next());
        }
        return actorBuffer.toString();
    }
}

public class DvdTitleExpression extends DvdAbstractExpression {
    public String interpret(DvdInterpreterContext
                            dvdInterpreterContext) {
        ArrayList titles = dvdInterpreterContext.getAllTitles();
        ListIterator titlesIterator = titles.listIterator();
        StringBuffer titleBuffer = new StringBuffer("");
        boolean first = true;
        while (titlesIterator.hasNext()) {
            if (!first) {
                titleBuffer.append(", ");
            } else {
                first = false;
            }
            titleBuffer.append((String)titlesIterator.next());
        }
        return titleBuffer.toString();
    }
}

public class DvdTitleActorExpression extends DvdAbstractExpression {
    String title;

    public DvdTitleActorExpression(String title) {
        this.title = title;
    }

    public String interpret(DvdInterpreterContext dvdInterpreterContext) {
        ArrayList titlesAndActors =
            dvdInterpreterContext.getTitlesForActor(title);
        ListIterator titlesAndActorsIterator =
            titlesAndActors.listIterator();
        StringBuffer titleBuffer = new StringBuffer("");
        boolean first = true;
        while (titlesAndActorsIterator.hasNext()) {
            if (!first) {
                titleBuffer.append(", ");
            } else {
```

```
                first = false;
            }
            titleBuffer.append((String)titlesAndActorsIterator.next());
        }
        return titleBuffer.toString();
    }
}

public class TitleAndActor {
    private String title;
    private String actor;
    public TitleAndActor(String title, String actor) {
        this.title = title;
        this.actor = actor;
    }
    public String getTitle() {return this.title;}
    public String getActor() {return this.actor;}
}
```

A UML diagram explaining the above diagram in further detail is pictured below:



10.4.4 Iterator

An **Iterator** provides a way to access the elements of an aggregate object sequentially, without exposing its underlying representation. Lists and ArrayLists have built-in iterators in Java, whereas more complex data structures might be custom iterators. Below is a simple example of an iterator in action:

```
public interface DvdListIterator {
   public void first();
   public void next();
   public boolean isDone();
   public String currentItem();
}

public class DvdList {
```

```
private String[] titles;
//Yes, it would be easier to do this whole example with ArrayList
// and ListIterator, but that would not be as much fun
private int titleCount;
//title count is always a real count of titles, but one ahead of
//itself as a subscript
private int arraySize;

public DvdList() {
     titles = new String[3];
     //using 3 to demonstrate array expansion more easily,
     //  not for efficiency
     titleCount = 0;
     arraySize = 3;
}

public int count() {
    return titleCount;
}

public void append(String titleIn) {
    if (titleCount >= arraySize) {
        String[] tempArray = new String[arraySize];
        for (int i = 0; i < arraySize; i++)
        {tempArray[i] = titles[i];}
        titles = null;
        arraySize = arraySize + 3;
        titles = new String[arraySize];
        for (int i = 0; i < (arraySize - 3); i++) {
            titles[i] = tempArray[i];
        }
    }
    titles[titleCount++] = titleIn;
}

public void delete(String titleIn) {
    boolean found = false;
    for (int i = 0; i < (titleCount -1); i++) {
        if (found == false) {
            if (titles[i].equals(titleIn)) {
                found = true;
                titles[i] = titles[i + 1];
            }
        } else {
            if (i < (titleCount -1)) {
                titles[i] = titles[i + 1];
```

```
            } else {
                titles[i] = null;
            }
        }
    }

    if (found == true) {
        --titleCount;
    }
}


public DvdListIterator createIterator() {
    return new InnerIterator();
}
```

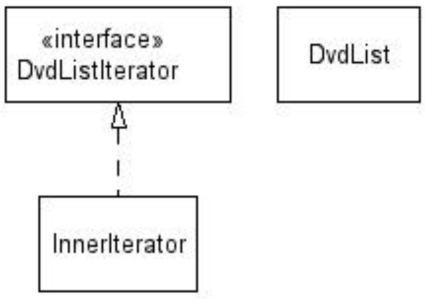
This example shows the Concrete Iterator as an inner class. The Iterator Pattern in the Gang of Four book allows multiple types of iterators for a given list or aggregate class. This can be accomplished with multiple Iterators in multiple inner classes. The `createIterator` class would then have multiple variations.

However, this implementation assumes that the developer has a limited number of iterator variants - which is usually the case. If more flexibility is required in iterator creation, the iterators should not be in inner classes, and perhaps some factory should be employed to create them.

```
private class InnerIterator implements DvdListIterator {
    private int currentPosition = 0;

    private InnerIterator() {}

    public void first() {
        currentPosition = 0;
    }

    public void next() {
        if (currentPosition < (titleCount)) {
            ++currentPosition;
        }
    }

    public boolean isDone() {
        if (currentPosition >= (titleCount)) {
            return true;
        } else {
```

```
            return false;
        }
    }

    public String currentItem() {
        return titles[currentPosition];
    }
  }
}
```

A UML diagram showing the inner workings of the Iterator pattern is shown below:



10.4.5 Mediator

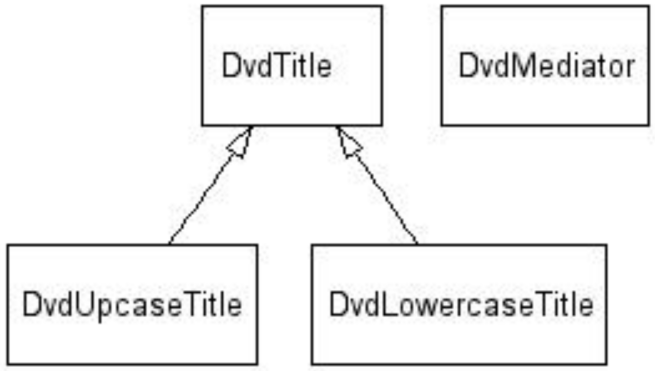
A **Mediator** is an object that encapsulates how a set of objects interact. A mediator promotes loose coupling by keeping objects from explicitly referring to each other, allowing programmers to vary their interaction independently. Code for a simple example of a mediator in action is shown below:

```
public abstract class DvdTitle {
   private String title;

   public void setTitle(String titleIn) {
       this.title = titleIn;
   }
   public String getTitle() {
       return this.title;
   }
}

public class DvdLowercaseTitle extends DvdTitle {
   private String LowercaseTitle;
   private DvdMediator dvdMediator;
```

```
    public DvdLowercaseTitle(String title, DvdMediator dvdMediator) {
        this.setTitle(title);
        resetTitle();
        this.dvdMediator = dvdMediator;
        this.dvdMediator.setLowercase(this);
    }

    public DvdLowercaseTitle(DvdTitle dvdTitle,
                             DvdMediator dvdMediator) {
        this(dvdTitle.getTitle(), dvdMediator);
    }

    public void resetTitle() {
        this.setLowercaseTitle(this.getTitle().toLowerCase());
    }
    public void resetTitle(String title) {
        this.setTitle(title);
        this.resetTitle();
    }

    public void setSuperTitleLowercase() {
        this.setTitle(this.getLowercaseTitle());
        dvdMediator.changeTitle(this);
    }

    public String getLowercaseTitle() {
        return LowercaseTitle;
    }
    private void setLowercaseTitle(String LowercaseTitle) {
        this.LowercaseTitle = LowercaseTitle;
    }
}

public class DvdUpcaseTitle extends DvdTitle {
    private String upcaseTitle;
    private DvdMediator dvdMediator;

    public DvdUpcaseTitle(String title,
                          DvdMediator dvdMediator) {
        this.setTitle(title);
        resetTitle();
        this.dvdMediator = dvdMediator;
        this.dvdMediator.setUpcase(this);
    }
```

```
    public DvdUpcaseTitle(DvdTitle dvdTitle,
                          DvdMediator dvdMediator) {
        this(dvdTitle.getTitle(), dvdMediator);
    }

    public void resetTitle() {
        this.setUpcaseTitle(this.getTitle().toUpperCase());
    }
    public void resetTitle(String title) {
        this.setTitle(title);
        this.resetTitle();
    }

    public void setSuperTitleUpcase() {
        this.setTitle(this.getUpcaseTitle());
        dvdMediator.changeTitle(this);
    }

    public String getUpcaseTitle() {
       return upcaseTitle;
    }
    private void setUpcaseTitle(String upcaseTitle) {
        this.upcaseTitle = upcaseTitle;
    }
}

public class DvdMediator {
    private DvdUpcaseTitle dvdUpcaseTitle;
    private DvdLowercaseTitle dvdLowercaseTitle;

    public void setUpcase(DvdUpcaseTitle dvdUpcaseTitle) {
        this.dvdUpcaseTitle = dvdUpcaseTitle;
    }

    public void setLowercase(DvdLowercaseTitle dvdLowercaseTitle) {
        this.dvdLowercaseTitle = dvdLowercaseTitle;
    }

    public void changeTitle(DvdUpcaseTitle dvdUpcaseTitle) {
        this.dvdLowercaseTitle.resetTitle(dvdUpcaseTitle.getTitle());
    }

    public void changeTitle(DvdLowercaseTitle dvdLowercaseTitle) {
        this.dvdUpcaseTitle.resetTitle(dvdLowercaseTitle.getTitle());
    }
}
```

Here is a UML diagram showing how the mediator was used above:



10.4.6 Memento

A program using the **Memento** pattern captures and externalizes an object's internal state so the object can be restored to this state later, without violating encapsulation; one object stores another object's state. See the example below for details on a memento in action:

```
import java.util.ArrayList;
import java.util.ListIterator;

//the originator
public class DvdDetails {
   private String titleName;
   private ArrayList stars;
   private char encodingRegion;

   public DvdDetails(String titleName,
                     ArrayList stars,
                     char encodingRegion) {
       this.setTitleName(titleName);
       this.setStars(stars);
       this.setEncodingRegion(encodingRegion);
   }

   private void setTitleName(String titleNameIn) {
       this.titleName = titleNameIn;
   }
   private String getTitleName() {
```

```
    return this.titleName;
}

private void setStars(ArrayList starsIn) {
    this.stars = starsIn;
}
public void addStar(String starIn) {
    stars.add(starIn);
}
private ArrayList getStars() {
    return this.stars;
}
private static String getStarsString(ArrayList starsIn) {
    int count = 0;
    StringBuffer sb = new StringBuffer();
    ListIterator starsIterator = starsIn.listIterator();
    while (starsIterator.hasNext()) {
        if (count++ > 0) {sb.append(", ");}
        sb.append((String) starsIterator.next());
    }
    return sb.toString();
}

private void setEncodingRegion(char encodingRegionIn) {
   this.encodingRegion = encodingRegionIn;
}
private char getEncodingRegion() {
    return this.encodingRegion;
}

public String formatDvdDetails() {
    return ("DVD: " + this.getTitleName() +
            ", starring: " + getStarsString(getStars()) +
            ", encoding region: " + this.getEncodingRegion());
}

//sets the current state to what DvdMemento has
public void setDvdMemento(DvdMemento dvdMementoIn) {
    dvdMementoIn.getState();
}
//save the current state of DvdDetails in a DvdMemento
public DvdMemento createDvdMemento() {
    DvdMemento mementoToReturn = new DvdMemento();
    mementoToReturn.setState();
    return mementoToReturn;
}
```

```
    //an inner class for the memento
    class DvdMemento {
        private String mementoTitleName;
        private ArrayList mementoStars;
        private char mementoEncodingRegion;

        //sets DvdMementoData to DvdDetails
        public void setState() {
             //Because String is immutable, we can just set
             //  the DvdMemento Strings to = the DvdDetail Strings.
             mementoTitleName = getTitleName();
             mementoEncodingRegion = getEncodingRegion();
             //However, ArrayLists are not immutable,
             //  so we need to instantiate a new ArrayList.
             mementoStars = new ArrayList(getStars());
        }
        //resets DvdDetails to DvdMementoData
        public void getState() {
             setTitleName(mementoTitleName);
             setStars(mementoStars);
             setEncodingRegion(mementoEncodingRegion);
        }

        //only useful for testing
        public String showMemento() {
             return ("DVD: " + mementoTitleName +
                     ", starring: " + getStarsString(mementoStars) +
                     ", encoding region: " + mementoEncodingRegion);
        }
    }
}
```

There is no polymorphism in this example, so the UML diagram is irrelevant and will not be included in this example.

10.4.7 Observer

An **Observer** defines a one-to-many dependency between objects so that when one object changes state, all of its dependencies are notified and updated automatically; an object notifies other objects if it changes.

```
import java.util.ArrayList;
import java.util.ListIterator;
```

```
public class DvdReleaseByCategory {
   String categoryName;
   ArrayList subscriberList = new ArrayList();
   ArrayList dvdReleaseList = new ArrayList();

   public DvdReleaseByCategory(String categoryNameIn) {
       categoryName = categoryNameIn;
   }

   public String getCategoryName() {
       return this.categoryName;
   }

   public boolean addSubscriber(DvdSubscriber dvdSubscriber) {
       return subscriberList.add(dvdSubscriber);
   }

   public boolean removeSubscriber(DvdSubscriber dvdSubscriber) {
       ListIterator listIterator = subscriberList.listIterator();
       while (listIterator.hasNext()) {
           if (dvdSubscriber == (DvdSubscriber)(listIterator.next())) {
               listIterator.remove();
               return true;
           }
       }
       return false;
   }

   public void newDvdRelease(DvdRelease dvdRelease) {
       dvdReleaseList.add(dvdRelease);
       notifySubscribersOfNewDvd(dvdRelease);
   }

   public void updateDvd(DvdRelease dvdRelease) {
       boolean dvdUpdated = false;
       DvdRelease tempDvdRelease;
       ListIterator listIterator = dvdReleaseList.listIterator();
       while (listIterator.hasNext()) {
           tempDvdRelease = (DvdRelease)listIterator.next();
           if (dvdRelease.getSerialNumber().
                 equals(tempDvdRelease.getSerialNumber())) {
               listIterator.remove();
               listIterator.add(dvdRelease);
               dvdUpdated = true;
               break;
           }
```

```
        }
        if (dvdUpdated == true) {
            notifySubscribersOfUpdate(dvdRelease);
        } else {
            this.newDvdRelease(dvdRelease);
        }
    }

    private void notifySubscribersOfNewDvd(DvdRelease dvdRelease) {
        ListIterator listIterator = subscriberList.listIterator();
        while (listIterator.hasNext()) {
            ((DvdSubscriber)(listIterator.next())).
                newDvdRelease(dvdRelease, this.getCategoryName());
        }
    }

    private void notifySubscribersOfUpdate(DvdRelease dvdRelease) {
        ListIterator listIterator = subscriberList.listIterator();
        while (listIterator.hasNext()) {
            ((DvdSubscriber)(listIterator.next())).
                updateDvdRelease(dvdRelease, this.getCategoryName() );
        }
    }
}

public class DvdSubscriber {
    private String subscriberName;

    public DvdSubscriber(String subscriberNameIn) {
        this.subscriberName = subscriberNameIn;
    }

    public String getSubscriberName() {
        return this.subscriberName;
    }

    public void newDvdRelease(DvdRelease newDvdRelease,
                              String subscriptionListName) {
        System.out.println("");
        System.out.println("Hello " + this.getSubscriberName() +
                           ", subscriber to the " +
                              subscriptionListName +
                           " DVD release list.");
        System.out.println("The new Dvd " +
                             newDvdRelease.getDvdName() +
                           " will be released on " +
```

```
                            newDvdRelease.getDvdReleaseMonth() + "/" +
                            newDvdRelease.getDvdReleaseDay() + "/" +
                            newDvdRelease.getDvdReleaseYear() + ".");
    }

    public void updateDvdRelease(DvdRelease newDvdRelease,
                                 String subscriptionListName) {
        System.out.println("");
        System.out.println("Hello " + this.getSubscriberName() +
                           ", subscriber to the " +
                             subscriptionListName +
                           " DVD release list.");
        System.out.println(
          "The following DVDs release has been revised: " +
          newDvdRelease.getDvdName() + " will be released on " +
          newDvdRelease.getDvdReleaseMonth() + "/" +
          newDvdRelease.getDvdReleaseDay() + "/" +
          newDvdRelease.getDvdReleaseYear() + ".");
    }
}

public class DvdRelease {
    private String serialNumber;
    private String dvdName;
    private int dvdReleaseYear;
    private int dvdReleaseMonth;
    private int dvdReleaseDay;


    public DvdRelease(String serialNumber,
                      String dvdName,
                      int dvdReleaseYear,
                      int dvdReleaseMonth,
                      int dvdReleaseDay) {
        setSerialNumber(serialNumber);
        setDvdName(dvdName);
        setDvdReleaseYear(dvdReleaseYear);
        setDvdReleaseMonth(dvdReleaseMonth);
        setDvdReleaseDay(dvdReleaseDay);
    }

    public void updateDvdRelease(String serialNumber,
                                 String dvdName,
                                 int dvdReleaseYear,
                                 int dvdReleaseMonth,
                                 int dvdReleaseDay) {
```

```
    setSerialNumber(serialNumber);
    setDvdName(dvdName);
    setDvdReleaseYear(dvdReleaseYear);
    setDvdReleaseMonth(dvdReleaseMonth);
    setDvdReleaseDay(dvdReleaseDay);
}

public void updateDvdReleaseDate(int dvdReleaseYear,
                                 int dvdReleaseMonth,
                                 int dvdReleaseDay) {
    setDvdReleaseYear(dvdReleaseYear);
    setDvdReleaseMonth(dvdReleaseMonth);
    setDvdReleaseDay(dvdReleaseDay);
}

public void setSerialNumber(String serialNumberIn) {
    this.serialNumber = serialNumberIn;
}
public String getSerialNumber() {
    return this.serialNumber;
}

public void setDvdName(String dvdNameIn) {
    this.dvdName = dvdNameIn;
}
public String getDvdName() {
    return this.dvdName;
}

public void setDvdReleaseYear(int dvdReleaseYearIn) {
    this.dvdReleaseYear = dvdReleaseYearIn;
}
public int getDvdReleaseYear() {
    return this.dvdReleaseYear;
}

public void setDvdReleaseMonth(int dvdReleaseMonthIn) {
    this.dvdReleaseMonth = dvdReleaseMonthIn;
}
public int getDvdReleaseMonth() {
    return this.dvdReleaseMonth;
}

public void setDvdReleaseDay(int dvdReleaseDayIn) {
    this.dvdReleaseDay = dvdReleaseDayIn;
}
```

```
   public int getDvdReleaseDay() {
       return this.dvdReleaseDay;
   }
}
```

Once again, a UML diagram detailing this design pattern consists of three labeled boxes with no arrows between them since there is no polymorphism here. Thus, it will not be supplied.

10.4.8 State

Programs using the **State** design pattern empower objects to alter their behavior when their internal states change. Objects may even appear to change their classes. An object appears to change its' class when the class it passes calls through to switches itself for a related class.

The following example exhibits this well:

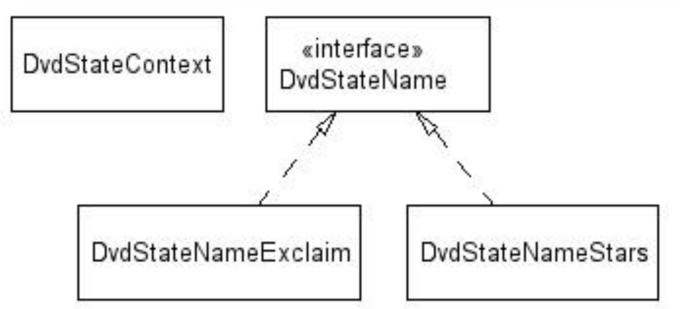
```
public class DvdStateContext {
   private DvdStateName dvdStateName;

   public DvdStateContext() {
       setDvdStateName(new DvdStateNameStars());
       //start with stars
   }

   public void setDvdStateName(DvdStateName dvdStateNameIn) {
       this.dvdStateName = dvdStateNameIn;
   }

   public void showName(String nameIn) {
       this.dvdStateName.showName(this, nameIn);
   }
}
```

The interface for the actual program states, along with both possible states to change to, follow below:

```
public interface DvdStateName {
   public void showName(DvdStateContext dvdStateContext,
                        String nameIn);
}

public class DvdStateNameExclaim implements DvdStateName {
```

```
    public DvdStateNameExclaim() {}

    public void showName(DvdStateContext dvdStateContext,
                         String nameIn) {
         System.out.println(nameIn.replace(' ','!'));
         //show exclaim only once, switch back to stars
         dvdStateContext.setDvdStateName(new DvdStateNameStars());
    }
}

public class DvdStateNameStars implements DvdStateName {
    int starCount;

    public DvdStateNameStars() {
        starCount = 0;
    }

    public void showName(DvdStateContext dvdStateContext,
                         String nameIn) {
         System.out.println(nameIn.replace(' ','*'));
         // show stars twice, switch to an exclamation point
         if (++starCount > 1) {
             dvdStateContext.setDvdStateName(
               new DvdStateNameExclaim());
         }
    }
}
```

The UML diagram for this example follows:



10.4.9 Strategy or Policy

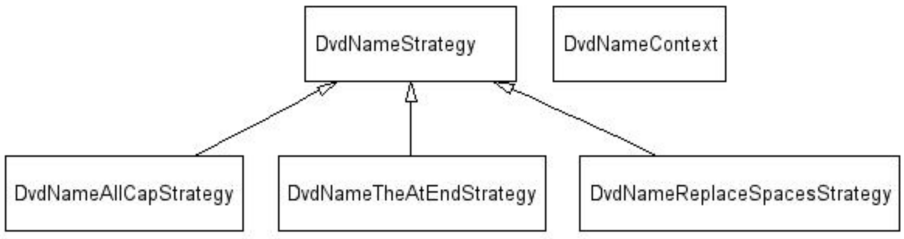
Define a family of algorithms, encapsulate each, and make them interchangeable. The **Strategy** design pattern lets the algorithm vary independently from clients

that use it. An object controls which of a family of methods, each member called a strategy, is called. Each class method extends a common base class. A simple example of different strategies connected by an interface is shown below:

```
public abstract class DvdNameStrategy {
   public abstract String
     formatDvdName(String dvdName, char charIn);
}

public class DvdNameAllCapStrategy extends DvdNameStrategy {
   public String formatDvdName(String dvdName, char charIn) {
       return dvdName.toUpperCase();
   }
}

public class DvdNameTheAtEndStrategy extends DvdNameStrategy {
   public String formatDvdName(String dvdName, char charIn) {
       if (dvdName.startsWith("The ")) {
           return new String(dvdName.substring(4,
             (dvdName.length())) + ", The");
       }
       if (dvdName.startsWith("THE ")) {
           return new String(dvdName.substring(4,
             (dvdName.length())) + ", THE");
       }
       if (dvdName.startsWith("the ")) {
           return new String(dvdName.substring(4,
             (dvdName.length())) + ", the");
       }
       return dvdName;
   }
}

public class DvdNameReplaceSpacesStrategy extends DvdNameStrategy {
   public String formatDvdName(String dvdName, char charIn) {
       return dvdName.replace(' ', charIn);
   }
}

public class DvdNameContext {
   private DvdNameStrategy dvdNameStrategy;

   public DvdNameContext(char strategyTypeIn) {
       switch (strategyTypeIn) {
           case 'C' :
```

```
            this.dvdNameStrategy = new DvdNameAllCapStrategy();
            break;
          case 'E' :
            this.dvdNameStrategy = new DvdNameTheAtEndStrategy();
            break;
          case 'S' :
            this.dvdNameStrategy =
              new DvdNameReplaceSpacesStrategy();
            break;
          default  :
            this.dvdNameStrategy = new DvdNameTheAtEndStrategy();
        }
    }

    public String[] formatDvdNames(String[] namesIn) {
        return this.formatDvdNames(namesIn, ' ');
    }

    public String[] formatDvdNames(String[] namesIn, char replacementIn) {
        String[] namesOut = new String[namesIn.length];
        for (int i = 0; i < namesIn.length; i++) {
            namesOut[i] =
              dvdNameStrategy.formatDvdName(namesIn[i], replacementIn);
            System.out.println(
              "Dvd name before formatting: " + namesIn[i]);
            System.out.println(
              "Dvd name after formatting:  " + namesOut[i]);
            System.out.println("==========================");
        }
        return namesOut;
    }
}
```

It is apparent that this is an example of polymorphism and modularity at work; developers often employ the strategy design pattern without even knowing it. Check out the UML diagram below to see this in action:



10.4.10 Template

With the **Template** design pattern, programmers define the skeleton of an algorithm in an operation, deferring some steps to subclasses. The template method lets subclasses redefine specific steps of an algorithm without changing the algorithm's structure. An abstract class defines various methods and has one non-overridden method, which calls the various methods.

Templating is a common practice even in small systems - type definitions, structs, and generics all utilize the template method in their internal representation. There is no example for the template pattern in this text, as examples exist everywhere and are easily implementable. As such, there is no accompanying UML diagram.

10.4.11 Visitor

A **Visitor** represents an operation to be performed on the elements of an object structure. A visitor lets programmers define new operations without changing the classes and elements of their dependencies. One or more related classes have the same method, which calls a method specific for themselves in another class. The example below illustrates well the complex nesting style needed here:

```
public abstract class TitleBlurbVisitor {
   String titleBlurb;
   public void setTitleBlurb(String blurbIn) {
       this.titleBlurb = blurbIn;
   }
   public String getTitleBlurb() {
       return this.titleBlurb;
   }

   public abstract void visit(BookInfo bookInfo);
   public abstract void visit(DvdInfo dvdInfo);
   public abstract void visit(GameInfo gameInfo);
}

public class TitleLongBlurbVisitor extends TitleBlurbVisitor {
   public void visit(BookInfo bookInfo) {
       this.setTitleBlurb("LB-Book: " +
                          bookInfo.getTitleName() +
                          ", Author: " +
                          bookInfo.getAuthor());
   }
```

```
    public void visit(DvdInfo dvdInfo) {
        this.setTitleBlurb("LB-DVD: " +
                           dvdInfo.getTitleName() +
                           ", starring " +
                           dvdInfo.getStar() +
                           ", encoding region: " +
                           dvdInfo.getEncodingRegion());
    }

    public void visit(GameInfo gameInfo) {
        this.setTitleBlurb("LB-Game: " +
                           gameInfo.getTitleName());
    }
}

public class TitleShortBlurbVisitor extends TitleBlurbVisitor {
    public void visit(BookInfo bookInfo) {
        this.setTitleBlurb("SB-Book: " + bookInfo.getTitleName());
    }

    public void visit(DvdInfo dvdInfo) {
        this.setTitleBlurb("SB-DVD: " + dvdInfo.getTitleName());
    }

    public void visit(GameInfo gameInfo) {
        this.setTitleBlurb("SB-Game: " + gameInfo.getTitleName());
    }
}

public abstract class AbstractTitleInfo {
    private String titleName;
    public final void setTitleName(String titleNameIn) {
        this.titleName = titleNameIn;
    }
    public final String getTitleName() {
        return this.titleName;
    }

    public abstract void accept(TitleBlurbVisitor titleBlurbVisitor);
}

public class BookInfo extends AbstractTitleInfo {
    private String author;

    public BookInfo(String titleName, String author) {
        this.setTitleName(titleName);
```

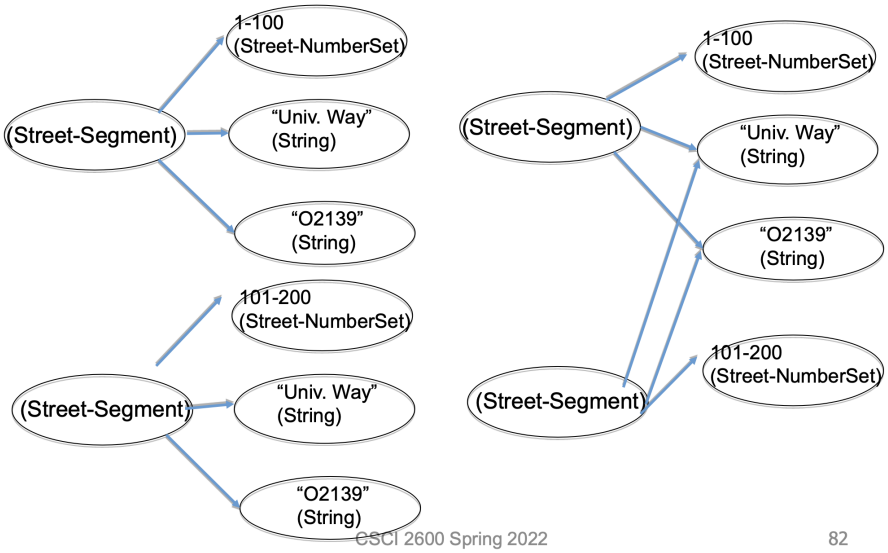
```
        this.setAuthor(author);
    }

    public void setAuthor(String authorIn) {
        this.author = authorIn;
    }
    public String getAuthor() {
        return this.author;
    }

    public void accept(TitleBlurbVisitor titleBlurbVisitor) {
        titleBlurbVisitor.visit(this);
    }
}

public class DvdInfo extends AbstractTitleInfo {
    private String star;
    private char encodingRegion;

    public DvdInfo(String titleName,
                   String star,
                   char encodingRegion) {
        this.setTitleName(titleName);
        this.setStar(star);
        this.setEncodingRegion(encodingRegion);
    }

    public void setStar(String starIn) {
        this.star = starIn;
    }
    public String getStar() {
        return this.star;
    }
    public void setEncodingRegion(char encodingRegionIn) {
        this.encodingRegion = encodingRegionIn;
    }
    public char getEncodingRegion() {
        return this.encodingRegion;
    }

    public void accept(TitleBlurbVisitor titleBlurbVisitor) {
        titleBlurbVisitor.visit(this);
    }
}

public class GameInfo extends AbstractTitleInfo {
```

```
    public GameInfo(String titleName) {
        this.setTitleName(titleName);
    }

    public void accept(TitleBlurbVisitor titleBlurbVisitor) {
        titleBlurbVisitor.visit(this);
    }
}
```

The below UML diagram illustrates the relationship between modules and their abstract visitor or visitee interfaces:

10.5 Interning Pattern

The interning pattern is not a Gang of Four design pattern. With the interning pattern, existing objects with the same value are copied, instead of creating new, identical objects. Unlike the singleton pattern, multiple instances of the same object exist, but objects with the same value are reused where needed. Additionally, interned objects can be compared with == instead of equals because they are the same object. Interning may improve program speed!



Interning applies to immutable objects only. Thus, Java strings can be interned. Here is an example of the Interning Pattern:

```
HashMap<String,String> names;

String canonicalName(String n) {
    if (names.containsKey(n))
        return names.get(n);
    else {
        names.put(n,n);
        return n;
    }
}
```

Here is the JavaDoc for String.intern():

```
public String intern()
\\
\\
Returns a canonical representation for the string object.
\\
\\
A pool of strings, initially empty, is maintained privately by the class String.
\\
\\
When the intern method is invoked, if the pool already contains a string equal to this Strin
\\
\\
It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if
```

```
\\
\\
All literal strings and string-valued constant expressions are interned.  String literals ar
\\
\\
Returns:
\\
a string that has the same contents as this string, but is guaranteed to be
from a pool of unique strings.
```

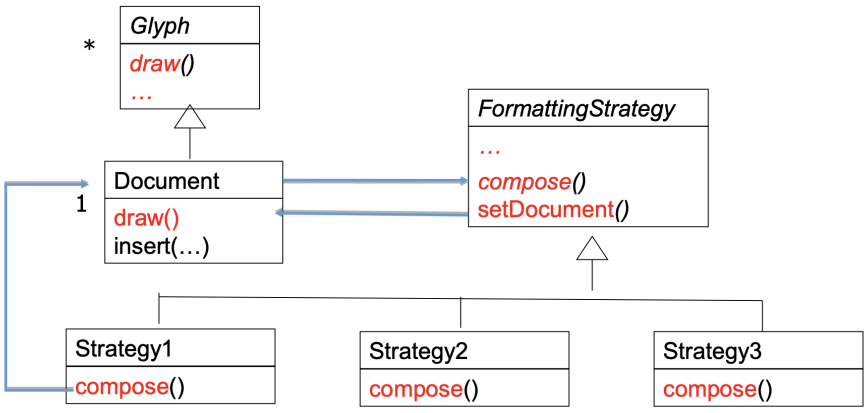
10.6 Wrappers

A wrapper is a thin layer over an encapsulated class that uses composition/delegation to modify its interface - as we did with the GraphWrapper. Then, the wrapper extends the behavior of the encapsulated class and restricts access to the encapsulated object. The encapsulated object (delegate) does most work. The wrapper is not a GoF pattern, but is similar to Adapter and Façade GoF patterns.

10.7 Strategy / Policy Pattern

This pattern defines multiple algorithms and lets client applications pass the algorithm to be used as a parameter. The strategy pattern reduces coupling by allowing users to program to an interface, rather than an implementation. Collections.sort is an example. It takes the Comparator parameter. Based on the different implementations of Comparator interfaces, the compared objects are sorted in different ways. Strategy pattern is very similar to State and Command Patterns. The strategy pattern is useful when we have multiple algorithms for a specific task. We want our application to be flexible in choosing any algorithms at runtime for a specific task.

- Encapsulates an algorithm in an object



11 Refactoring

Refactoring is necessary when we have complex, ugly code that works. There is a tendency to leave that awful code alone as long as it works: "if it isn't broken, don't fix it." The trouble there is that it often will break; it is only a matter of when. **Refactoring** is a preemptive, disciplined rewrite of code likely to break in the future, consisting of small-step behavior-preserving transformations, followed by the execution of test cases. Of course, this depends on having a good suite of tests. Continuous refactoring combined with testing is an essential software development practice.

Programming is just refactoring a blank screen to make it act according to spec. There are several possible refactoring methods:

- *Extract*
- *Move*
- *Replace Temp with Query*
- *Replace Type Code with State or Strategy*
- *Replace Conditional with Polymorphism*
- *Form Template*
- *Replace Magic Number with Symbolic Constant*

These have relatively well-defined mechanics and can be automated; Eclipse can automate some of these.

Technical Debt reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution. The accumulation of technical debt often stems from the following causes. Notice how each of these comes from ignorance of the code base or programming itself:

- Business pressures
- Lack of understanding
- Tightly coupled modules
- Not enough or improper testing
- Lack of documentation
- Lack of collaboration

To avoid these, make small, disciplined changes, and test after each change.

11.1 Code Smells

Refactorings attack code smells. Code Smells can be:

- Any characteristic in the source code of a program that indicates a deeper problem
- Big methods
- A **God Class**, an oversized class that controls everything
- Similar methods, classes, or subclasses
- Little or no use of subtype polymorphism
- High coupling between objects
- **Feature Envy**, where a class that uses methods of another class excessively
- **Inappropriate Intimacy**, when a class that depends on implementation details of another class
- **Refused Bequest** a class that overrides a base class method such that the contract of the base class is broken
- **Lazy Class/Freeloader**: a class that does too little
- Excessive use of literals
- **Cyclomatic Complexity** has too many branches or loops
- **Downcasting** happens when a type cast breaks the abstraction model
- **Orphan Variable** or **Constant Class** is a class that typically has a collection of constants that belong elsewhere
- **Data Clump** occurs when a group of variables is passed around in various parts of the program; it would be more appropriate to formally group the different variables into a single object

Smells are certain structures in the code that indicate a violation of fundamental design principles and negatively impact design quality. They are not necessarily bugs and are not technically incorrect, though they indicate possible weaknesses in code. They are also a reliable indicator of technical debt similar to antipatterns.

To fix code smells, make long methods shorter, remove duplicate code, introduce design patterns, and remove the use of hard-coded constants. The goal is to achieve short, tight, clear code without duplication.

11.2 Extract Method

The method `Customer.statement()` is too big, so it is difficult to understand and maintain. The solution to this problem is to find a logical, frequently used chunk of code to be extracted out of `statement()` and perform an extract method refactoring.

With the **Extract Method**, create a new method, copy duplicate code in the new method, and scan this (the extracted code) for references to local variables. If local variables are used only in extracted code, declare them in the new method. See if the extracted code modifies any other variables; this can be tricky. Pass unmodified local variables only used by extracted code into the method as variables. Replace the extracted code with a method call, compile, and test.

11.3 Move Method

There is unnecessary coupling from `Customer` to `Rental` through `getDaysRented()` and `Movie`. Additionally, the customer does not have the information to compute their rental amount. To fix this, move `amountFor(Rental)` to the logical information expert class.

With the **Move Method**, examine all features used by the source method defined on the source class. Consider if they should also be moved. Next, check the sub- and superclasses for other declarations of the method and declare it in the target class. Then, appropriately copy the code from source to target. Finally, reference the correct target object from the source and turn it into a delegating method. After all this, compile and test to ensure the move worked.

In this particular case, initially, replace the old method with delegation:

```
class Customer {
    private double amountFor(Rental aRental) {
        return aRental.getCharge();
    }
}
```

Compile and test to see if it works. Next, find each reference to `amountFor` and replace it with a call to the new method; `thisAmount = amountFor(each);` becomes `thisAmount = each.getCharge();`. Again, compile, test, and hope the move method worked!

11.4 Replacing Conditional Logic

A switch statement can be challenging to work with, hard to understand, and hard to change. The first step towards a solution is to move `getCharge` and `getFrequentRenterPoints` from `Rental` to `Movie`.

For example:

```
    class Rental { // replace with delegation
        double getCharge() {
        ...
        return _movie.getCharge(_daysRented);
    }
}
```

Replace abstract class `Price` with concrete subclasses `Regular`, `Children's`, and `NewRelease`. This way, each Price subclass defines its unique `getPriceCode()` and `getCharge()` method.

11.5 Replace Temp with Query Refactoring

The problem is that the temporary variable `thisAmount` is meaningless and hinders readability. To fix this, replace `thisAmount` with the query method `each.getCharge()`. A **Query Method** is more informative than a temporary variable, even though it does not modify anything.

With the **Replace Temp with Query Method**, look for a temporary variable that is assigned only once. Declare the temp as final, compile, and extract the right-hand side of the assignment into a query method. Then, replace all occurrences of the temporary variable with the query. The query method computing the temporary value should be free of side effects, which improves implementation.

11.6 Replacing Type Code with State or Strategy

Can an algorithm vary independently from the object that uses it? Let us use movie pricing as an example. The class `Movie` represents a movie; there are several pricing algorithms and strategies for movies, and we need to add new ones quickly. However, placing the pricing algorithms and strategies in `Movie` will make the class too big and complex, so developers add a switch statement code smell that temporarily solves the problem.

Here, adding a State pattern solves the problem of making behavior depend on state. For this, define a context class to present a single interface to the outside world. In the example, `Chain` is the context. Then, define a `State` abstract base class. Represent the different states of the state machine as derived classes of the `State` base class. Define state-specific behavior in the appropriate classes. Maintain the current state in the context class by composition. To change the state of the state machine, change the current state reference.

11.7 Replace Conditional with Polymorphism Refactoring

With this design `Regular` defines its `getCharge`:

```
double getCharge(int daysRented) {
    double result = 2;
    if (daysRented > 2)
        result += (daysRented - 2)*1.5;
    return result;
}
```

`Children's` and `NewRelease` define their `getCharge(int)`, so `getCharge(int)` in `Price` becomes abstract. The last two refactorings go together and break the transformation into small steps, the goal being to replace the switch statement with polymorphism. To implement the **Replace Conditional with Polymorphism** refactoring, refactor the type code according to the State design pattern. Second, place each case branch into a subclass, and add a virtual call, in this case, `price.getCharge(daysRented)`.

11.8 Template Method Design Pattern and Form Template Method Refactoring

Several methods implement the same algorithm but differ at some steps, such as `TextStatement.value` and `HtmlStatement.value`. To solve this problem, define the skeleton of the algorithm in a superclass and defer differing steps to subclasses. Take the example of `TextStatement` and `HtmlStatement`. First, record a header substring, `customer info` in this case, and iterate over all the rentals, recording each rental substring. Finally, record the footer substring: `total charge`. Recorded substrings differ from text to Html.

12 Usability

Usability is the ease of use and learnability of a human-made object, such as a tool or device. In software engineering, usability is the degree to which specified consumers can use software to achieve quantified objectives with effectiveness, efficiency, and satisfaction in the context of use. Ideally, products would be more time-efficient, intuitive, and satisfying to use. We often refer to usability as UX – User Experience.

Usability is how well users can use the system's functionality. The dimensions of usability are:

- Learnability: is it easy to learn?
- Efficiency: once learned, is it fast to use?
- Safety: are errors few and recoverable?
- Memorability: is it easy for users to remember what they learned?
- Satisfaction: is it enjoyable to use?
- Simplicity

Different dimensions vary in importance, depending on the user and the task. Usability is only one aspect of the system, so software designers must worry about other aspects: cost, security, functionality, maintainability, size, and more.

12.1 Learnability and Memorability

In humans, working memory is small: we have $7 + -2$ "chunks" of thought we can keep track of at once. These are usually gone in 10 seconds. **Maintenance Rehearsal** is required to keep short-term memory from decaying, but it costs attention. Maintenance rehearsal is the process of repeatedly verbalizing or thinking about a piece of information. Human short-term memory can be increased to about 30 seconds by using Maintenance Rehearsal.

For example, Alice wants to order a pizza, and Bob tells Alice the phone number. Alice repeats the phone number to herself until she can type it into the phone. Alice just used maintenance rehearsal to order the pizza.

In contrast, long-term memory is practically infinite in size and duration. Elaborative rehearsal is required to transfer short-term thoughts into long-term memory. **Elaborative Rehearsal** is a memory technique that utilizes thinking about the meaning of the term to be remembered.

An excellent example of elaborative rehearsal is a medical student's attempt to remember the term "neuron." In order to permanently commit the term to memory, they look up what it means, learn its purpose, study a diagram, and think about how a neuron might relate to things they already know. If they do (*rehearse*) this several times, they will be more likely to remember the term "neuron."

Consistency is required when designing software for learnability. Similar things should look similar, and different items in an application should behave differently. Terminology, location, internal, external, and metaphorical design should be consistent with use. When designing, use common, simple words instead of tech jargon, and rely on recognition instead of recall. Labeled buttons are better than commands, and combo boxes are better than text boxes.

12.2 Perception

Good design makes heavy use of human perception. For instance, humans have something called **Perceptual Fusion**, where stimuli 100ms apart, or ten frames per second, seem to create a moving picture. Thus, a computer graphics response of $<$ 100ms looks instantaneous. In many cases, computer response time dictates what design choices we make:

- ¡ 0.1 seconds: seems instantaneous
- 0.1 - 1 seconds: user notices
- 1 - 5 seconds: display a loading animation or busy cursor
- ¿ 5 seconds: display a progress bar

Software designers must also consider visibility when creating the UI and UX for a system. For example, color blindness is pervasive, with roughly 8% of all males unable to distinguish red from green. With that in mind, it is good practice not to differentiate different items by color alone. To design for visibility, make the system state visible: keep the user informed about what is happening in the application. For instance, keep the cursor, selection highlight, and status bar visible. Prompt feedback about user actions is also helpful and keeps users engaged.

Even motor control and function affect software; humans use **Closed-Loop Control**. We get sensory feedback, correct errors accordingly, and consciously, continuously, adjust muscle movements, with a cycle time of about 140ms. An excellent example is threading a needle - a slow process requiring total attention. We also use **Open-Loop Control**, where the motor processor runs with no feedback in a cycle time of around 70ms. Open-loop control in practice exhibits itself in a gymnast's double-back somersault - it is a swift process!

Finally, we also have something called **Fitt's Law**, which measures the time it takes a human to point at a target of S size from a distance D. That function can be measured as

$$T = RT + MT = a + b \log \frac{D}{S}$$

, where RT is Reaction Time, MT is Movement Time, and $\log \frac{D}{S}$ is the index difficulty of the pointing task, such as moving a hand. Fitt's Law applies only if a path to the target is unconstrained. The **Steering Law**,

$$T = RT + MT = a + b \frac{D}{S}$$

, is applied when the path is constrained to something like a tunnel.

Keeping Fitt's Law and the Steering Law in mind, make important targets big or nearby and avoid steering tasks in applications. Provide shortcuts, such as keyboard accelerators, styles, bookmarks, and history, to increase usability.

12.3 Safety

Modes are states in which actions have different meanings, as in vim's insert mode vs. vim's command mode. Avoid mode errors by eliminating them, making them visible, or putting disjoint action sets in different modes. Programmers and users alike need feedback about which mode their application is in.

To further increase safety, consider what the user might already know about an application's tools:

- *Buttons* are used for single independent actions relevant to the current screen.
- *Toolbars* are good for common actions.
- *Menus* are used for infrequent actions that may apply to multiple screens
- *Checkboxes* are for on/off switches when one switch can be toggled independently of others
- *Radio Buttons* are good for related choices when only one choice can be activated at a time
- *Lists* are utilized when there are several fixed choices - too many for radio buttons to be practical.
- *Text Fields* show up when the user needs to enter text information
- *Combo Boxes* are good for when there are multiple fixed choices, but the screen would be cluttered if they were all displayed at once

- A *Tabbed Pane* is useful when there are several screens that the user may want to switch between
- *Dialog Boxes* present temporary screens or options.

When programmers come across error messages, they should be precise, speak the user's language, offer constructive help, and be polite.

12.4 Simplicity and Satisfaction

In design, less is more! Always omit extraneous information, graphics, and features. Good graphic design constitutes few, well-chosen colors and fonts, and groupings with white space tooltips. In the same vein, concise language helps users understand what is going on while keeping the site simple.

In an extensive system, the user manual is also essential. When writing it, use program and UI metaphors, and include key functionality, but do not include an exhaustive list of all menus. Ask:

- What is hard to do?
- Who is the target audience?

Power users need a manual; casual users might not. As such, the software should be obvious to use - piecemeal online help is no substitute for good documentation.

12.5 Prototyping and Testing

The first step of design is to create a low-fidelity prototype before working on the GUI. Paper is a speedy and effective prototyping tool; it is quick work to sketch windows, menus, dialogs, and widgets. Designers who use paper prototyping can crank many designs and evaluate them quickly. Thus, hand sketching is preferable because it allows designers to focus on behavior and interactions, not fonts and colors. With some extra thought, paper prototypes can even be executed. Use pieces of paper to represent windows, dialogs, and menus, and simulate the computer's responses by moving those pieces around and writing on them.

When testing, start with a prototype, write a few (short but non-trivial) representative tasks, find three representative users, and watch them do tasks with the prototype. Brief the user first. Tell them:

1. "I am testing the system, not testing you."
2. "If you have trouble, it is the system's fault."
3. "Feel free to quit at any time."

A significant ethical issue is informed consent - ensure the user receives this. Ask the user to think aloud without helping, explaining, or pointing out mistakes. Only speak up to prod the user to think aloud, and to move on to the next task when stuck. Take lots of notes.

Programmers test with users to watch for **Critical Incidents** that strongly affect task performance or satisfaction. User responses to these are usually negative - they can run into errors, repeat attempts, and may even curse at the tester. However, responses can also be positive, such as "Cool!" or "Oh, now I see."

12.6 Review

Programmers are not users.

Keep human capabilities and design principles in mind.

Iterate over the design.

Write documentation.

Make cheap, throw-away prototypes.

Evaluate prototypes and designs with users.

13 Graphical User Interface (GUI) Programming

We use design patterns and concepts to program GUIs. The Observer or Composite design pattern is often used when programming graphical user interfaces. GUI components are often part of the View and Controller. GUI components are often composites. GUIs use Event-driven programming and Large APIs.

13.1 Terminology

We must lay out some important definitions before moving forward with GUI programming. For instance, a **Window** is a first-class citizen of the graphical desktop, such as a frame.

A **Component** is a GUI widget that resides in a window, such as a button, text box, or label. Every property in a component has accessor methods, such as `getFont` or `isVisible`, and modifier methods, like `setFont`.

Some modifiable properties include **Background**, the color behind component, **Border**, the line around a component; **Enabled**, an indicator of interactability; **Focusable**, whether text can be edited; and **Font**, used for text in components.

A **Container** is a component that holds other components. Windows, such as `JFrame` or `Jdialog`, are **Top-Level Containers**: they live at the top of UI hierarchy, and are not nested; they can be used by themselves, but usually host other components. `JPanel`, `JToolBar` are **Mid-Level Containers** – they can stand alone, but more often contain other components. **JPanel** is a general-purpose component for drawing or hosting other UI elements. Some examples of **Specialized Containers** are menus or list boxes; all components are containers, but not all are general purpose.

There are some methods used very often in GUI programming. **setSize(int width, int height)** gives the frame a fixed size in pixels. **pack()** resizes the frame to fit components, and **setVisible(true)** shows the window.

13.2 JFrame

A **JFrame** is a top-level graphical window that typically holds other components. In implementation, some common methods are:

- `JFrame(String title)`: constructor, title optional
- `setSize(int width, int height)`
- `add(Component c)`: add component to window
- `setVisible(boolean v)`: necessary to initialize on scree. Do not forget this step!

Even when containers are closed, they may still be serving their purpose. `setDefaultCloseOperation(int o)` sets a given action for the frame to perform when it closes. A common value for this is `JFrame.EXIT\_ON\_CLOSE`. If the closing operation is not set, the program will never exit, even if the frame is closed. Do not forget this to set closing behavior! Options for behavior on close are:

- **DO_NOTHING_ON_CLOSE**: Do not do anything - handle the operation in the windowClosing method of a registered WindowListener object.
- **HIDE_ON_CLOSE**: Automatically hide the frame after invoking any registered WindowListener objects.
- **DISPOSE_ON_CLOSE**: Automatically hide and dispose of the frame after invoking any registered WindowListener objects.
- **EXIT_ON_CLOSE**: Exit the application using the System exit method. Use this only in applications.

A good example of `JFrame` in action, based on examples 7-1 in Core Java 8th edition, is as follows:

```
package gui1;
import javax.swing.*;
/* Simple graphics window (based on ex. 7-1 in Core Java 8th ed.) */

public class SimpleFrameMain {
/** Create a frame and display it. */
public static void main(String[] args) {
 SimpleFrame frame = new SimpleFrame("A Window");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.setVisible(true);
}
}
class SimpleFrame extends JFrame {
public SimpleFrame(String title) {
 super(title); // set the title
 pack(); // resize the frame to fit components if there are any
 this.setSize(1200,800);
}
}
```

13.3 JPanel

Panels are used to group other containers. They contain graphics, buttons, labels, and anything else a panel should have. Panels must be added to a frame or other container via `frame.add(new JPanel(...));`. `JPanel` has many methods in common with `JFrame` because they serve a similar function, though

panels can be nested at any depth. However, some new methods exist, like `setPreferredSize(Dimension d)`.

13.4 Layout Managers

Layout Managers are in charge of sizing and positioning. **Absolute Positioning** is a feature of C++, C#, and many other languages, where the programmer specifies the exact coordinates of every component. For example, with absolute positioning, one can say, "Put this button at ($x = 15$, $y = 75$) and make it$70x31$ pixels in size". In contrast, layout managers in Java create objects that decide where to position each component based on general rules or criteria. For example, "Put these four buttons into a $2x2$ grid" or "place these text boxes in a horizontal flow in the southern part of the frame."

Each container has a layout manager. **FlowLayout (left to right, top to bottom)**, which is the default layout manager for JPanel, and **BorderLayout (center, north, south, east, west)**, which is the default layout manager for JFrame. FlowLayout treats containers as left-to-right, top-to-bottom "paragraphs." Components are then given a preferred size and positioned in the added order. If they are too long, components wrap around to the next line.

`BorderLayout` divides containers into five regions: North and South regions expand to fill components horizontally and use the preferred size convention vertically. West and East regions expand to fill regions vertically and the preferred size horizontally. Finally, center regions use all space not occupied by other elements.

13.5 Graphing, Painting, and other things

What if we want to draw something like an image or a path? Extending JPanel and overriding `paintComponent` enable this. The method in `JComponent` that draws components is `SimplePaintMain.java`.

There are many methods to draw various lines, shapes, and images. For images, the source file into an Image object and use `g.drawImage(...)`. Then, the implementation will look like this:

```
package gui3;
import java.awt.*;
import javax.swing.*;
/* Example overriding paintComponent to create a static drawing. */
public class SimplePaintMain {
/** Create a labeled panel with a couple of shapes and display it. */
public static void main(String[] args) {
JFrame frame = new JFrame("Simple Painting Example");
JPanel panel = new SimplePainting();
```

```
panel.setPreferredSize(new Dimension(300,200));
JLabel label = new JLabel("A Work of Art");
label.setHorizontalAlignment(SwingConstants.CENTER);
frame.add(panel,BorderLayout.CENTER);
frame.add(label,BorderLayout.SOUTH);
frame.pack();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
}
class SimplePainting extends JPanel {
private static final long serialVersionUID = -4725346878809211984L;
/** Paint some simple shapes whenever the window manager calls this. */
@Override
public void paintComponent(Graphics g) {
// ensure any background belonging to the container is painted
super.paintComponent(g);
// Cast g to its actual class to make graphics2d methods available.
// We do not use them here, but this cast is often necessary.
Graphics2D g2 = (Graphics2D) g;
// draw a couple of shapes
g2.setColor(Color.green);
g2.fillOval(40,30,120,100);
g2.setColor(Color.red);
g2.drawRect(60,50,60,60);
System.out.println("Finished drawing");
}
}
```

The window manager calls `paintComponent` when the window is first made visible and whenever it is needed afterward. Never call `paintComponent` manually - that may take its resources away from a more automatic repaint. Instead, if manually redrawing a window is necessary, call `repaint()`. That tells the window manager to schedule repainting. The window manager will then call `paintComponent` when it can, which may not be right away.

Use `paintComponent`'s `Graphics` parameter to do all the drawing, and only the drawing. Do not meddle with the `Graphics` parameter otherwise; it is quick to anger. In the same vein, do not create new Graphics or Graphics2D objects.

13.6 Event-Driven Programming

The main body of the program functions like an event loop, though it does not exist inside a loop. **Event-Driven Programming** is a style of programming where events dictate the flow of execution. The program loads, then waits for the user to input **Events**, objects that represent user interaction with the GUI

component. As events occur, the program runs code to respond. The flow of what is executed is determined by event order.

In contrast, application or algorithm-driven control expects input in well-defined places. Algorithm-driven control is typical for large non-GUI applications.

A **Listener** is an object that waits for events and handles them. To handle an event, attach a listener to a component. The listener will be notified when the event (e.g., button click) occurs.

There are a few different kinds of GUI events:

- **Mouse**: move, drag, click, button press, and button release
- **Keyboard**: key press and release, sometimes with modifiers like shift, control, or alt
- **Touchscreen**: finger tap and drag
- Joystick, drawing tablet, other device inputs
- **Window** resize, minimize, restore, or close
- Network activity or file I/O (start, done, error)
- Timer interrupt (including animations)

Every event object contains information about events it can handle. That information includes the triggering GUI component and other information depending on the event, like ActionEvent (text string from a button) or MouseEvent (mouse coordinates). An **Action Event** is an action that has occurred on a GUI component. Action events are the most common event type in Swing. Some examples are button clicks and check box-checking or unchecking. These are all represented by the class `ActionEvent`. They are handled by objects that implement `interface ActionListener`.

The appropriate method specified in the interface is called when an event occurs. For example, when an action event occurs, `actionPerformed` gets called on the attached or registered Listeners. An event object is then passed as a parameter to the event listener method - for instance, `actionPerformed(ActionEvent e)`.

`JButton(String text)` creates a new button with the given string as text. `String getText()` and `void setText(String text)` get and set a button's text, respectively. See this in action below:

```
package gui4;
import java.awt.*; // basic AWT classes
import java.awt.event.*; // event classes
import javax.swing.*; // Swing classes
```

```
/* Straightforward demo of Swing event handling.
 * Create a window with a single button that prints a message when clicked.

 * Version 1 with named inner class to handle button events.
 */
public class ButtonDemo1 {
// inner class to handle button events
private static class MyButtonListener implements ActionListener {
 final String id;
 private int nEvents = 0;

 /** Create a new MyButtonListener */
 public MyButtonListener(String id) {
 this.id = id;
 }

 /* Respond to events generated by the button by printing the
 * command and number of times the event has been triggered.
 * @param e the event created by the button when it was clicked.
 */
 @Override
 public void actionPerformed(ActionEvent e) {
 nEvents++;
 System.out.println(id + " " + e.getActionCommand() + " " + nEvents);

 }
}
public static void main(String[] args) {
 // Create a new window and set it to exit from the application when closed.

 JFrame frame = new JFrame("Button Demo");
 frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
 // Create a new button with the label "Hit me" and the string "OUCH!" to be

 // returned as part of each action event.
 JButton button = new JButton("Hit me");
 button.setActionCommand("OUCH!");
 button.addActionListener(new MyButtonListener("Listener1"));
 // Experiment: add two ButtonListeners
// button.addActionListener(new MyButtonListener("Listener2"));
// button.addActionListener(new MyButtonListener("Listener3"));
 // Experiment: add a single ButtonListener twice
// MyButtonListener mbl = new MyButtonListener("Listener4");
// button.addActionListener(mbl);
// button.addActionListener(mbl);
```

```
 // Functional version
// button.addActionListener(evt -> System.out.println("Listener5 OUCH!"));

 // Add a button to the window and make it visible.
 frame.add(button);
 frame.pack();
 frame.setVisible(true);
}
}
```

A single button listener can handle several buttons, so how does one tell which is which? An `ActionEvent` has a `getActionCommand` method that returns an "action command" string, in this case, the button name. Button names, by default, are "Default," so it is better to set button names to specific strings, which remain the same even if the UI and other button names change.

13.7 Serialization

`HashSet` is a serializable class. To **Serialize** a class, convert an object's state to a byte stream that can be reverted to a copy of the original object. Serializing requires a version number of type `long`. After a serialized object has been written into a file, it can be read from the file and deserialized. Java issues a warning if `Serializable` is extended without a version number. Luckily, Eclipse generates one automatically.

See this example of serialization:

```
public class Employee implements java.io.Serializable {
    public String name;
    public String address;
    public transient int SSN;
    public int number;
    public void mailCheck() {
        System.out.println("Mailing a check to " + name + " " + address);
    }
}
```

Fields not meant to be serialized are marked as **Transient**.

In general, the structure of a GUI Application can be summarized in two steps:

1. Place components in a **JPanel**, a Java container that stores groups of components
2. Add the new container to a **JFrame**, the main window in Java.

The container stores components and governs their positions, sizes, and resizing behavior. Once components are added to their container, `pack()` calculates

all component sizes and calls the layout manager to set component locations. Remember `pack();` omitting it can lead to (for lack of a better term) funky layouts. Use `SimpleLayoutMain.java` when in doubt.

13.8 Java GUI Libraries

Swing is the leading Java GUI library. There are many advantages to using Swing:

- It paints GUI components itself, pixel-by-pixel
- It does not delegate to the OS window system, though it emulates the look and feel of several platforms
- Swing is platform-independent - cross-platform compatibility is key to covering all operating system and browser options available
- It has an expanded set of widgets and features made possible by object-oriented design

Swing component objects all have a **Preferred Size** that they would like to be: just large enough to fit their contents. Some other layout managers, such as `FlowLayout`, also choose to size the components inside them to the preferred size. Others, such as `BorderLayout` or `GridLayout`, disregard the preferred size convention and use some other scheme to size components.

Abstract Windowing Toolkit (AWT) is Sun's initial GUI library. It maps Java code to each OS's windowing system. Despite this feature, AWT is otherwise clunky to use and offers a limited set of widgets.

JavaFX is the most recent GUI library. It was built to replace Swing eventually and uses a declarative framework.

13.9 Nested Classes

A **Nested Class** is a class defined in another class. **Inner Objects**, or objects within nested classes, can access the fields and methods of their outer method. Event listeners are often defined as inner classes inside a GUI. The syntax is as follows:

```
public class OuterClass {
    ...
    // Instance nested class (inner class)
    // A different class for each instance of Outer private class InnerClass { ... }
    // Static nested class
    // One class, shared by all instances of Outer
    static private class NestedClass { ... }
}
```

If a nested class is private, only the outer class can see it and create objects of the nested type. In that case, each inner object is associated with the outer object that created it so that the inner object can access or modify the outer object. The inner class can then refer to the outer object as `OuterClass.this`.

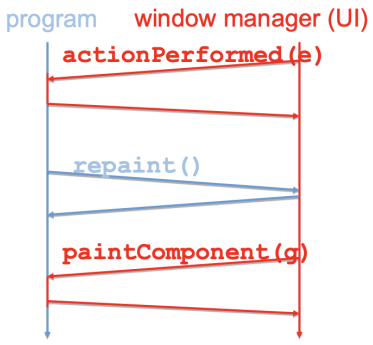
An **Anonymous Inner Class** is a textcolorBlueVioletuse-once class defined directly in the new expression. In an anonymous inner class, specify the base class to be extended or interface to be implemented, and override or implement needed methods. If the class gets complicated, the anonymous clause can be dropped. The syntax is as follows:

```
    button.addActionListner(new ActionListner() {
        public void actionPerformed(ActionEvent e) {
            doSomething()
            \\ implementation of the method for anonymous class
        }
    }
)
```

13.10 Program Thread and UI Thread

Programs and their UI run in concurrent threads. All UI actions happen in the UI thread, including callbacks like `paintComponent` and `actionPerformed`. After event handling, `repaint` can be called if `paintComponent` needs to run, but do not try to draw anything from inside the event handler. The series of actions is as follows:

```
\item \verb+Repaint()+ returns immediately
\item \verb+Painting+ occurs when the window manager is ready
\item \verb+paintComponent()+ is called
```

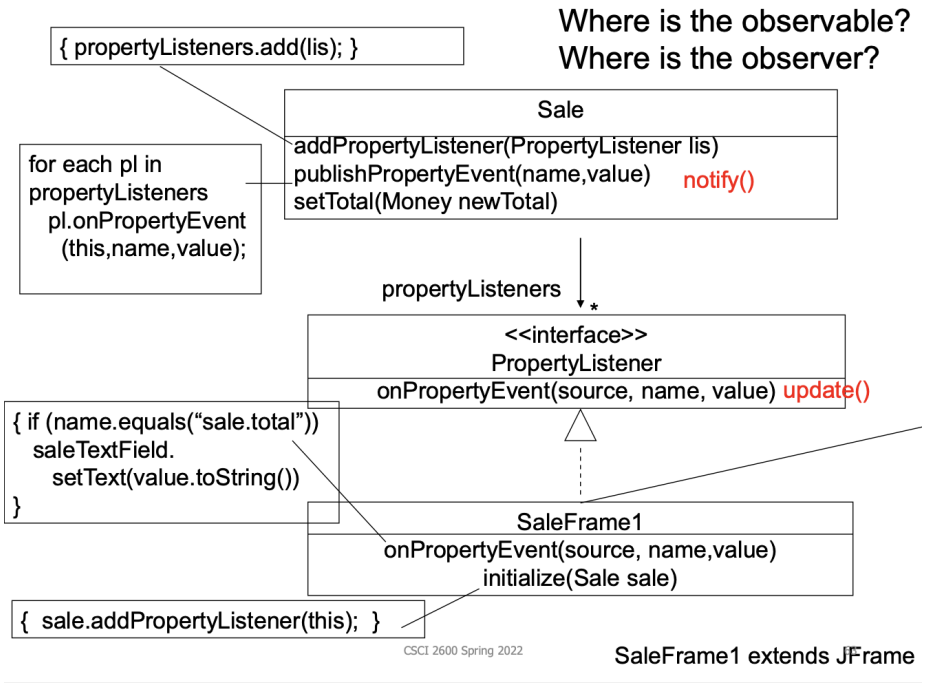


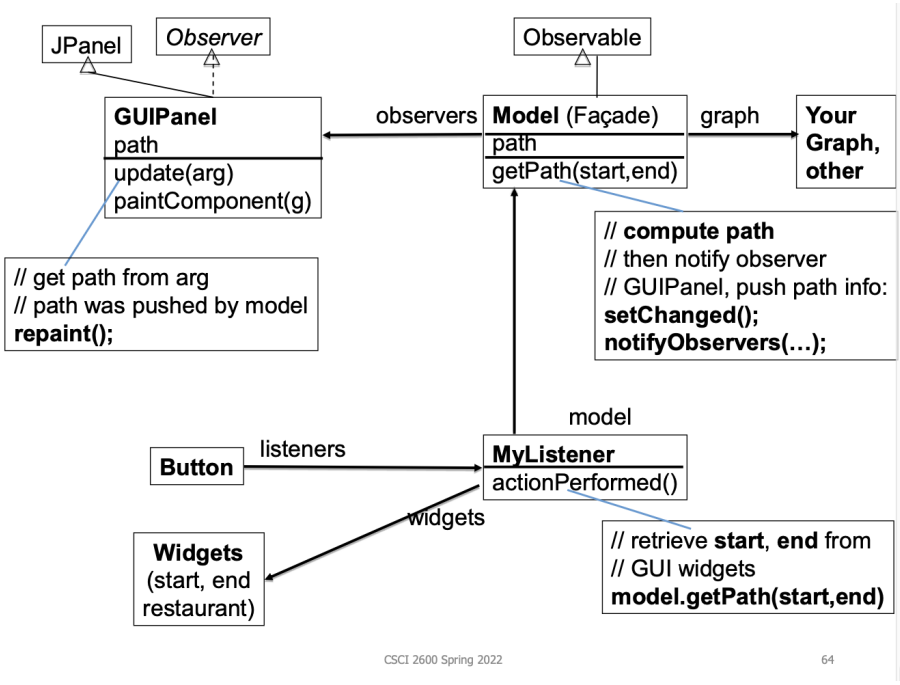
In the UI Thread, event handlers should only do a little work; if the event handler gets busy, the interface will appear to freeze up. On the other hand, if there is a lot to do, the event handler should handle some events that the program thread will notice, and then do the heavy work behind the scenes.

13.11 Components, Events, Listeners, and the Observer Pattern

The **Model View Controller (MVC)** design pattern, another name for the observer design pattern comes up a lot. One possible design for a GUI using MVC could be:

- A **Model** or observable class, such as `Sale`
- When `Sale` changes, it notifies its observers
- The **Component**, is an observer class that extends JFrame. A component can also be initialized by implementing some Listener interface.




JPanel
Observer
Observable
GUIPanel
path
update(arg)
paintComponent(g)
observers
Model (Façade)
path
getPath(start,end)
graph
Your Graph, other
// get path from arg
// path was pushed by model
repaint();
// compute path
// then notify observer
// GUIPanel, push path info:
setChanged();
notifyObservers(…);
model
Button
listeners
MyListener
actionPerformed()
widgets
Widgets
(start, end
restaurant)
// retrieve start, end from
// GUI widgets
model.getPath(start,end)
CSCI 2600 Spring 2022
64

14 Software Processes

Some software lifecycle activities require analysis, design, implementation, integration, testing, verification, deployment, and maintenance. A software development process puts all of these things together, but how? In what order?

14.1 Software Life cycle and Artifacts

The **Software Lifecycle** is the process of building and maintaining a software system. The activities from inception to end-of-life can take months or years. Each activity has specific goals. It defines a clear set of steps, produces an artifact (tangible item), allows for review, and specifies actions to perform in the next activity.

Prescribed activities create valuable artifacts:

1. **Requirements Analysis** produces requirements documents and supplementary specifications
2. The **Design Process** produces design models and artifacts
3. **Implementation** produces readable, modular code
4. **Testing** produces test suites as artifacts
5. **Maintenance** continues software development processes at a smaller scale throughout the whole system for the software's lifecycle

14.2 Steps

14.2.1 Step 1: Requirements Analysis and the Use-Case Model

A **Use-Case Model** is a model of how different types of users interact with a system to solve a problem using text stories called scenarios. A user can also be called an actor or a role at this stage.

A dice game on a gaming platform is an excellent example of a use case. The main success scenario is:

1. Player (actor) picks up the two dice and rolls them
2. If the player rolls a seven or "doubles" (the dice match), then the player wins
3. If not, then another player gets the chance to roll

Use Cases can take many forms. First, we have the **Brief Format**, a one-paragraph summary, usually for the main success scenario. Next is the **Casual Format**, which consists of multiple paragraphs covering various scenarios. Finally, the **Fully Dressed Format** includes all possible steps and variations the system is capable of supporting. Use cases are developed iteratively and often go through each stage in turn.

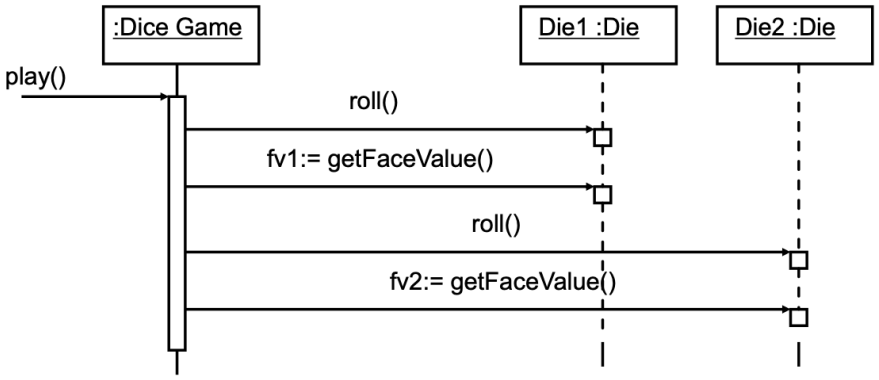
14.2.2 Step 2: Design Process

The design process is to be taken seriously; iterating over multiple possible designs until developers and clients agree on UI and specifications is the best way to guarantee success and reduce production costs over the long run.

Part of the **Design Process** includes creating interaction, class, and UML diagrams, all of which are shown below:

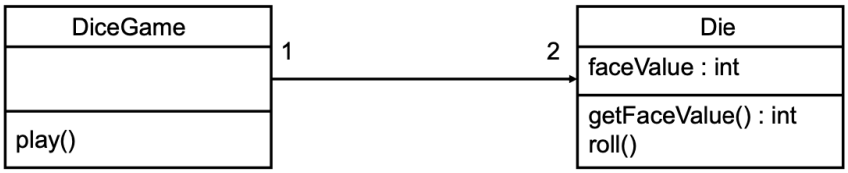
Example, Step 2:
Design: Interaction Diagram

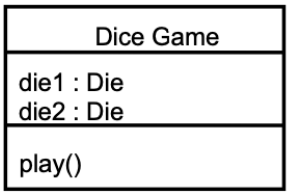
- Dynamic object design



Example, Step 2:
Design: Class Diagram

- Static design





14.2.3 Step 3: Implementation

The implementation process is based on the requirements and designs defined above and should be the shortest step. Coding according to the dice game specs should produce something like this:

```
/* a mutable class ... */
class DiceGame {
    private Die die1 = new Die();
    private Die die2 = new Die();

    // Abstraction Function: ...
    // Rep invariant: ...

    // @modifies:
    // @effects:
    public void play() {
        die1.roll();
        int fv1 = die1.getFaceValue();
        ...
    }
}
```

14.2.4 Step 4: Testing

Write Unit Tests after specifications, but before code. Then, once those are squared away, move on to System Tests. Then, after implementation is done, Integration Tests. There is a helpful chapter in this book called Testing, which readers are encouraged to consult before this one.

14.2.5 Step 5: Maintenance

This step is iterative and lasts for as long as the code exists. Maintenance involves Bug Fixes, Enhancements, and Refactoring until the system's end of life.

Other Engineering disciplines have similar processes. For example, Civil engineering has:

- Requirements: architectural design
- Design: engineering blueprints
- Implementation: Construction
- Testing: Inspections throughout construction and after completion
- Maintenance: inspections, repair, and upgrades

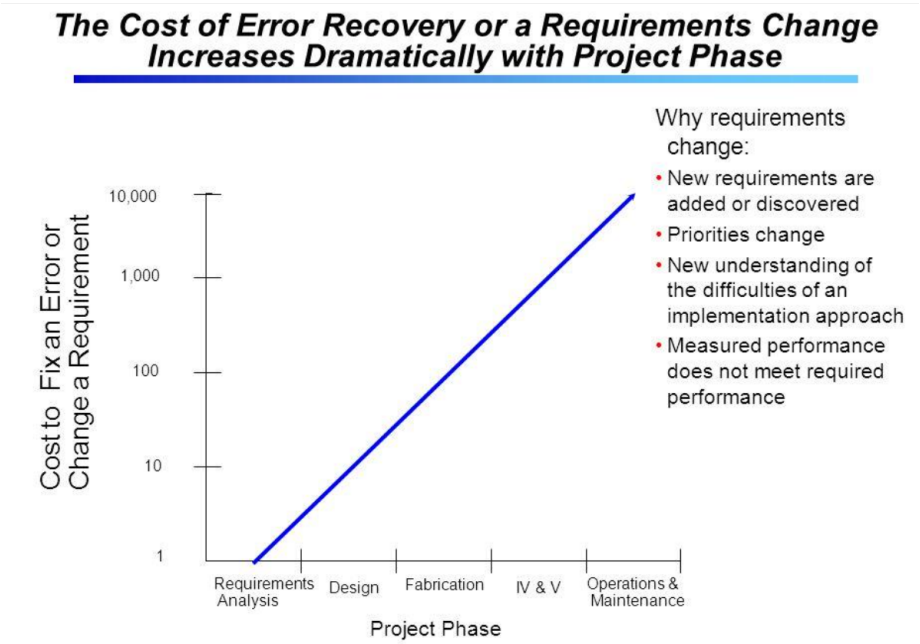
The main differences between software engineering and other disciplines are that requirements changes do not creep into construction; changing requirements in the middle of an extensive physical system is costly, while it is easy to do in code. Most engineering design and construction also use tried and true materials and techniques because it is a large physical system. Engineers are looking for well-built systems rather than innovation.

In contrast, it is easy to innovate in software because breaking things usually has little to no consequence in the real world - disasters are caught at or before the testing phase. Additionally, construction projects are usually (mostly) on time, while software projects tend to have much more variable timelines.

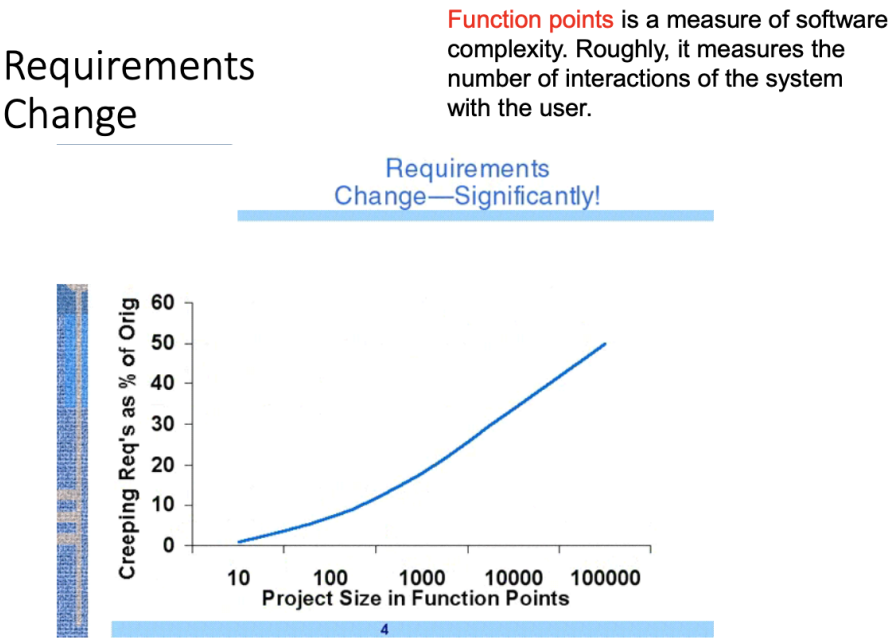
14.3 Requirements

Requirements analysis is complicated. Some significant causes of project failure are poor user input, incomplete requirements, and changing requirements. Some essential tools are requirements documents and use cases.

Requirements can be complex because they change; errors can be found in original requirements, so they must conform with what works. The customer also needs to evolve to match new requirements. Technical, scheduling, and cost problems can arise here because system requirements change as the world does. It is also often difficult to understand the implications of software design changes and other requirements. This following graph illustrates this relationship well:



There are a few ways to measure complexity. For example, there are **Function Points**, roughly the number of user interactions within the system, **Cyclomatic Complexity**, the number of predicate nodes in a control flow graph, and lines of code. The following graph illustrates some of this well:



Requirements can be classified in a few different ways. For example, the **FURPS+ Model** is an acronym for Functionality, Usability, Reliability, Performance, and Supportability. The + stands for design constraints, implementation requirements, and anything else specific to the system.

14.4 Software Development Processes

The software process puts together software lifecycle activities, including requirements analysis, design, implementation, testing, deployment, and maintenance. It also forces attention to these activities and their artifacts.

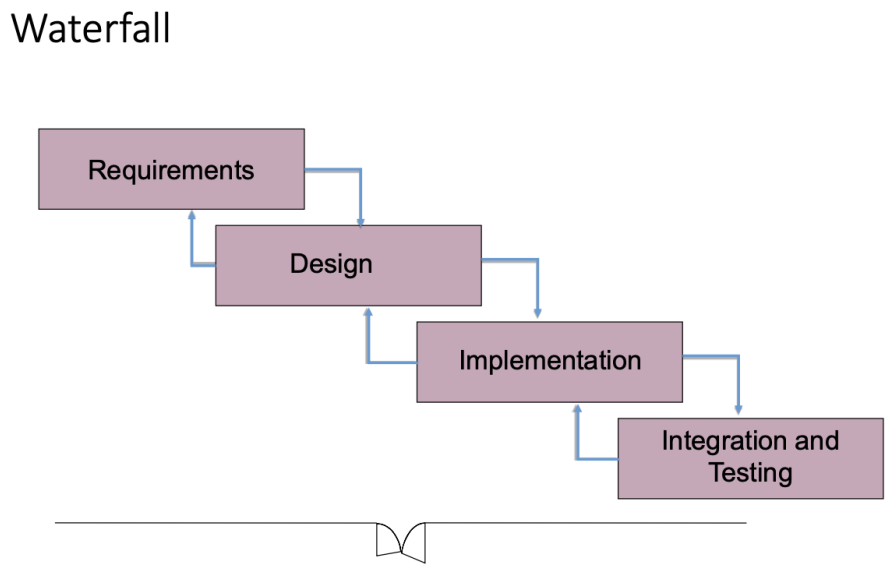
There are a few different software development strategies. The benefits of having a software process are the provided framework to work within, a management tool for unruly systems, and the forced attention to essential activities and artifacts. However, With a process, developers risk focusing too much on process artifacts, hurting their team and slowing development. Sometimes, a process requires substantial expertise and a learning curve, which can further slow things down.

14.4.1 Code-And-Fix or Ad-Hoc Process

The **Code-And-Fix** or **Ad-Hoc** process is familiar to those readers who have taken RPI's data structures: developers write some code, make up inputs, and debug accordingly. Some advantages of this are that there is little or no overhead; programmers can dive right in and see progress quickly.

This process is useful for small, short-lived projects, but is dangerous for most other projects. This process may ignore important tasks and artifacts such as design or testing. Sometimes, it is not clear when to start or stop an activity. In addition, it can be hard to review code with the ad-hoc method, so scales poorly to multiple people.

14.4.2 Waterfall Process

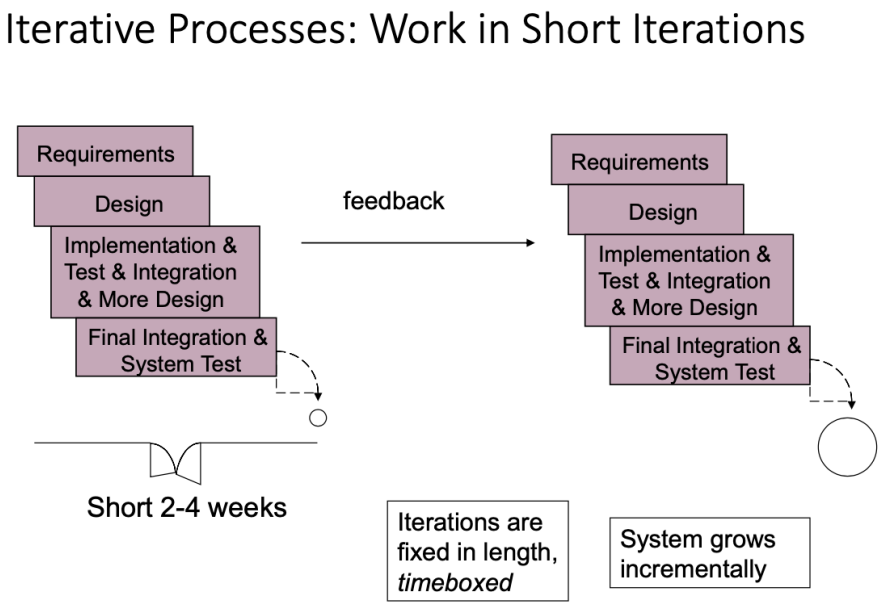


With the **Waterfall Process** process, developers tackle all planning up front with an orderly, easy-to-follow sequential process. Stages are well-defined, so reviews at each stage determine if the product is ready to advance. Thus, the waterfall process is ideal for experienced teams on short-term projects and projects with very well-understood requirements.

There are some drawbacks to using the Waterfall process, though. Waterfall assumes requirements are clear and well-understood, and most planning is done upfront. Because of that assumption, its rigid and sequential nature does not embrace change well. It can be costly to "swim upstream" back to an earlier phase because iteration is limited to the previous or next step.

There is also no sense of progress until the very end - Integration occurs then, which defies the "integrate early and often" rule. This practice increases inflexibility because there is no feedback until the very end. At its worst, the product may ultimately not match the customer's need. Reviews are also massive affairs due to inertia. The writers of this book do not recommend the waterfall method.

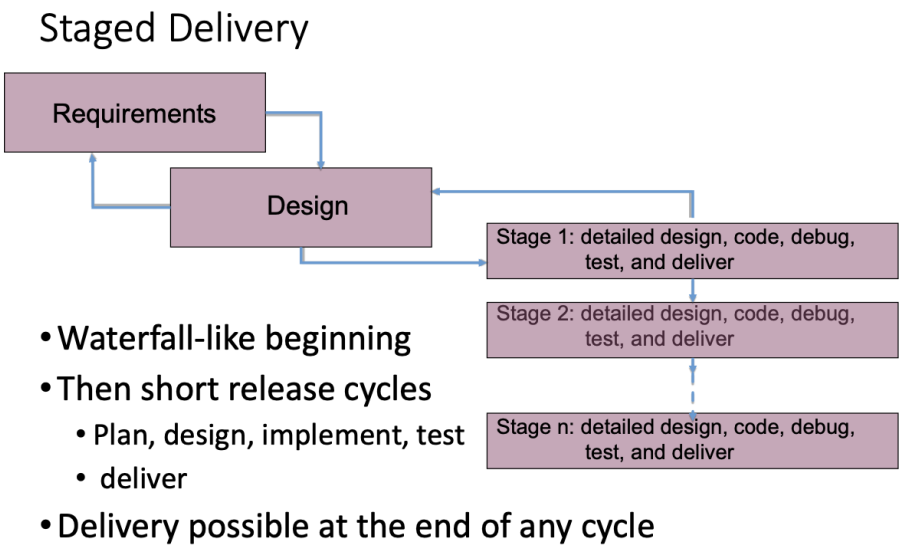
14.4.3 Iterative Process



The iterative process accommodates and embraces change. It provides constant feedback, so problems are visible early. The iterative process is especially apt at the beginning of any project when requirements are still fluid. Additionally, the most significant risks are usually addressed first, so as costs increase, risks decrease.

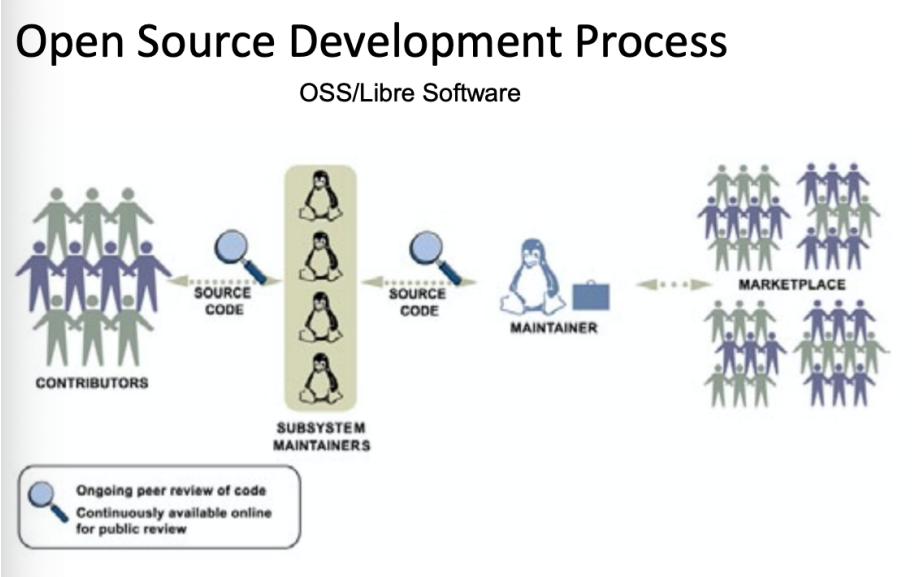
However, the iterative process could be better; it can take a lot of planning and management, and there are frequent changes in tasks. In addition, this process requires customer and contract flexibility, as well as developers able to assess risk accurately.

14.4.4 Staged Delivery



With **Staged Delivery**, developers can ship their product at the end of any release, which looks like a success to customers, even if all original goals are not met. Intermediate deliveries show progress and customers are happy, leading to feedback on how to best attack new issues. Problems are also visible early, facilitating shorter, more predictable release cycles. Staged delivery is practical, widely used, and successful, though it requires the flexibility of management, developers, and users.

14.4.5 Open Source Development



Open Source Development uses a distributed model, where individuals or small teams of contributors are responsible for the development and maintenance of code. Contributed features are integrated into a single body of code by one or more maintainers, which ensures that newly submitted code meets the overall vision and standards set for the project. The feature development lifecycle begins with an idea for a new project feature or enhancement, which is then proposed to project developers. There is online discussion among contributors, and the features are released as alpha software.

Feature requests are generally tracked and prioritized using processes visible to the rest of the development community, ensuring a common understanding of features and their relative priority, development status, associated bugs, blockers, and planned release.

The open-source development process encourages transparency and distributed collaboration, as many eyes make for fewer bugs. In addition, because code submissions can come from anyone, most projects have formal procedures to track code ownership when a patch is submitted.

The OSS development model has an interwoven development cycle with constant feature development - its structure ensures early and frequent release and peer review. Some open-source successes include the Linux Kernel, GNU toolchain, LibreOffice, Python, and the repository this text is hosted on.

15 Appendix

15.1 Halting Problem

The halting problem is a classic problem in computer science that asks whether it is possible to determine whether the program will run indefinitely or eventually halt, given a program and its input. In other words, it asks whether there is an algorithm that can determine whether a given program will run forever or will eventually stop.

The halting problem is important because it highlights the limitations of what can be computed by a computer. It was first proposed by mathematician Alan Turing in 1936 and is considered a fundamental problem in the theory of computation.

Here are some resources to learn more about the halting problem:

Wikipedia: `https://en.wikipedia.org/wiki/Halting\_problem`

Khan Academy:
`https://www.khanacademy.org/computing/computer-science/theory-of-computation/undecidability-`

Brilliant: `https://brilliant.org/wiki/halting-problem/`

Coursera:
`https://www.coursera.org/lecture/automata-theory/the-halting-problem-YbYOc`

15.2 Rice's Theorem

Rice's Theorem is a fundamental result in computer science that states that it is impossible to design an algorithm that can determine whether a given program will ever halt or run forever unless the program is trivial. In other words, it is impossible to design an algorithm to determine whether a given program will eventually halt or run forever, unless the program is so simple that it can be easily analyzed by hand.

Rice's Theorem is important because it highlights the limitations of what can be computed by a computer. It was first proposed by mathematician Henry Gordon Rice in 1953 and is considered a fundamental result in the theory of computation.

Here are some resources to learn more about Rice's Theorem:

Wikipedia: `https://en.wikipedia.org/wiki/Rice%27s\_theorem`

Khan Academy:

https://www.khanacademy.org/computing/computer-science/theory-of-computation/undecidability-

Brilliant: https://brilliant.org/wiki/rices-theorem/

Coursera:
https://www.coursera.org/lecture/automata-theory/rices-theorem-E5Q5F

15.3 Russel's Paradox

Russell's Paradox is a paradox that arises in set theory when trying to define a set that includes all sets that do not contain themselves as elements. The paradox arises because such a set cannot be a member of itself, since it contains all sets that do not contain themselves.

Russell's Paradox is important because it highlights the limitations of defining certain types of sets. It was first proposed by mathematician Bertrand Russell in 1901 and is considered a fundamental problem in the foundations of mathematics.

Here are some resources to learn more about Russell's Paradox:

Wikipedia: https://en.wikipedia.org/wiki/Russell%27s_paradox

Khan Academy:
https://www.khanacademy.org/math/set-theory/axiomatic-set-theory/russells-paradox/a/russells

Brilliant: https://brilliant.org/wiki/russells-paradox/

Coursera:
https://www.coursera.org/lecture/set-theory/russells-paradox-hYv0g

15.4 Java ADTs

Here is an interesting and helpful article on ADTs, written by the developers at TechVidvan in 2022:

https://techvidvan.com/tutorials/java-abstract-data-type/

15.5 Dafny

Here is a lecture by Will Sonnex and Sophia Drossopoulou of Microsoft on some of the uses and syntax of Dafny.

http://www.doc.ic.ac.uk/~scd/Dafny_Material/Lectures.pdf

15.6 OpenJML

Here is the source code for the OpenJML example in the Specifications section:

`https://www.openjml.org/examples/binary-search.html`

15.7 Design Patterns

All example java design patterns are from fluffycat.com, a website Dr. Kuzmin has used extensively in PSoft, which is now hosted on the Wayback Machine.

Thanks to Brian K. for pointing out that the singleton design pattern example was not type-safe.

Fluffycat:
`https://web.archive.org/web/20080208110123/http://www.fluffycat.com/`

Wayback Machine: `https://web.archive.org/`

15.8 Gang of Four

The Gang of Four (GOF) is a nickname for the four authors of the book "Design Patterns: Elements of Reusable Object-Oriented Software," published in 1994. The authors of the book are Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

The book is considered a classic in software engineering and has significantly influenced the development of object-oriented programming. It presents a catalog of common design patterns that can be used to solve common software design problems, and it is considered a valuable resource for software designers and developers.

Here are some resources to learn more about the Gang of Four and their work:

Wikipedia: `https://en.wikipedia.org/wiki/Design\_Patterns`

The Gang of Four Website: `https://www.gofpatterns.com/`

Head First Design Patterns: A Brain-Friendly Guide:
`https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124`

15.9 Software Errors

This article details some gruesome and cringeworthy software errors whose effects were felt worldwide.

`https://raygun.com/blog/costly-software-errors-history/`

15.10 Induction

Some definitions and explorations of induction were provided by Mary Barnes and Sue Gordon of The University of Sydney in their paper titled Mathematical Induction:

`https://docplayer.net/21121808-Mathematical-induction-mary-barnes-sue-gordon.html`

15.11 Testing

Examples and early iterations of graphics originated from Imperva's article on black box testing:

`https://www.imperva.com/learn/application-security/black-box-testing/`

15.12 Github

Let us remember where this whole project started. Dr. Konstantin Kuzmin provided the code examples used in lectures as fodder for this textbook. Many of these examples appear in these pages or on the website.

Students are encouraged to look through the rest of the repository to solidify their understanding of these concepts.

`https://github.com/KCony/PSoftExamples`

Writer Notes

Add more examples in every chapter

Add a Dafny "cheat sheet"

Get more concrete definitions in Refactoring to supplement the examples

Work on GUI Programming again so it flows better

make verbatim text smaller and wider in Design Patterns if possible

Fix link to microsoft lecture in appendix

Add links around the website

Fix chapter nesting in website