15SE203
OBJECT ORIENTED ANALYSIS AND DESIGN

Unit I Introduction

Complexity

- A physician, a civil engineer, and a computer scientist were arguing about what was the oldest profession in the world. The physician remarked, “Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world.” The civil engineer interrupted, and said, “But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth from out of the chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profession in the world.” The computer scientist leaned back in her chair, smiled, and then said confidently, “Ah, but who do you think created the chaos?”

Complexity

- The more complex the system, the more open it is to total breakdown
- users of software systems rarely think twice about asking for equivalent changes.
- Besides, they argue, it is only a simple matter of programming.

Complexity

- The complexity of software results in projects that are late, over budget, and deficient in their stated requirements.
- We often call this condition the software crisis, but frankly, a malady that has carried on this long must be called normal.

Complexity

- There are simply not enough good developers around to create all the new software that users need.
- Furthermore, a significant number of the development personnel in any given organization must often be dedicated to the maintenance or preservation of geriatric software.
- Given the indirect as well as the direct contribution of software to the economic base of most industrialized countries, and considering the ways in which software can amplify the powers of the individual, it is unacceptable to allow this situation to continue.

The Structure of Complex Systems

- If we open our eyes to the world about us, we will observe successful systems of significant complexity.
- Some of these systems are the works of humanity, such as the Space Shuttle, the England/France tunnel, and large business organizations.
- Many even more complex systems appear in nature, such as the human circulatory system and the structure of a habanero pepper plant.

//Complexity follows Hierarchical Order

The Structure of Complex Systems

- Here we see the hierarchic nature of a complex system.
- A personal computer functions properly only because of the collaborative activity of each of its major parts.
- Not only are complex systems hierarchic, but the levels of this hierarchy represent different levels of abstraction, each built upon the other, and each understandable by itself.
- At each level of abstraction, we find a collection of devices that collaborate to provide services to higher layers.

The Structure of a Personal Computer

- A personal computer is a device of moderate complexity.
- Most are composed of the same major elements: a central processing unit (CPU), a monitor, a keyboard, and some sort of secondary storage device, usually either a CD or DVD drive and hard disk drive.
- We may take any one of these parts and further decompose it.
- For example, a CPU typically encompasses primary memory, an arithmetic/logic unit (ALU), and a bus to which peripheral devices are attached.
- Each of these parts may in turn be further decomposed: An ALU may be divided into registers and random control logic, which themselves are constructed from even more primitive elements, such as NAND gates, inverters, and so on.

The Structure of Plants and Animals

- In botany, scientists seek to understand the similarities and differences among plants through a study of their morphology, that is, their form and structure.
- Plants are complex multi cellular organisms, and from the cooperative activity of various plant organ systems arise such complex behaviors as photosynthesis and transpiration.
- Plants consist of three major structures (roots, stems, and leaves).
- Each of these has a different, specific structure.
- For example, roots encompass branch roots, root hairs, the root apex, and the root cap.

The Structure of Plants and Animals

- All parts at the same level of abstraction interact in well-defined ways.
- For example, at the highest level of abstraction, roots are responsible for absorbing water and minerals from the soil.
- Roots interact with stems, which transport these raw materials up to the leaves.
- There are always clear boundaries between the outside and the inside of a given level.
- For example, we can state that the parts of a leaf work together to provide the functionality of the leaf as a whole and yet have little or no direct interaction with the elementary parts of the roots
- the mutual cooperation of meaningful collections of these agents do we see the higher-level functionality of a plant.
- The science of complexity calls this emergent behavior: The behavior of the whole is greater than the sum of its parts

The Structure of Plants and Animals

- The field of zoology, we note that multicellular animals exhibit a hierarchical structure similar to that of plants: Collections of cells form tissues, tissues work together as organs, clusters of organs define systems (such as the digestive system), and so on.
- The fundamental building block of all animal matter is the cell, just as the cell is the elementary structure of all plant life.
- For example, plant cells are enclosed by rigid cellulose walls, but animal cells are not. Notwithstanding these differences, however, both of these structures are undeniably cells.
- This is an example of commonality that crosses domains.
- A number of mechanisms above the cellular level are also shared by plant and animal life.
- For example, both use some sort of vascular system to transport nutrients within the organism, and both exhibit differentiation by sex among members of the same species.

Complexity in Traditional Systems

- **Software Complexity**
  - some software systems are not complex
  - The applications that are specified, constructed, maintained, and used by the same person, usually the amateur programmer or the professional developer working in isolation.
  - Such systems tend to have a very limited purpose and a very short life span.
  - We can afford to throw them away and replace them with entirely new software rather than attempt to reuse them, repair them, or extend their functionality.

- The industrial-strength software that exhibit a very rich set of behaviors, as, for example,
  - in reactive systems that drive or are driven by events in the physical world, and for which time and space are scarce resources;
  - applications that maintain the integrity of hundreds of thousands of records of information while allowing concurrent updates and queries;
  - systems for the command and control of real-world entities, such as the routing of air or railway traffic.

Reactive systems: one that requires update; e.g. ATM Machines

- Software systems such as these tend to have a long life span
- that simplify the creation of domain-specific applications
- programs that mimic some aspect of human intelligence.
- It is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of its design.
- The complexity of such systems exceeds the human intellectual capacity.

Software Is Inherently Complex

- The complexity of software is an essential property, not an accidental one" [3].
- We observe that this inherent complexity derives from four elements:

1. the complexity of the problem domain,
2. the difficulty of managing the development process,
3. the flexibility possible through software,
4. the problems of characterizing the behavior of discrete systems.

The Complexity of the Problem Domain

- The problems we try to solve in software often involve elements of inescapable complexity, in which we find a myriad of competing, perhaps even contradictory, requirements.
  - communication gap
  - the requirements of a software system often change during its development
  - large software system is a capital investment
    - *maintenance when we correct errors*
    - *evolution when we respond to changing requirements*
    - it is *preservation when we* continue to use extraordinary means to keep an ancient and decaying piece of software in operation.

The Difficulty of Managing the Development Process

- The fundamental task of the software development team is to engineer the illusion of simplicity
  - write less code by inventing clever and powerful mechanisms that
  - reusing frameworks of existing designs and code
  - we use a team of developers, and ideally we use as small a team as possible

The Flexibility Possible through Software

- Software offers the ultimate flexibility, so it is possible for a developer to express almost any kind of abstraction
- Uniform codes and standards exist in the software industry

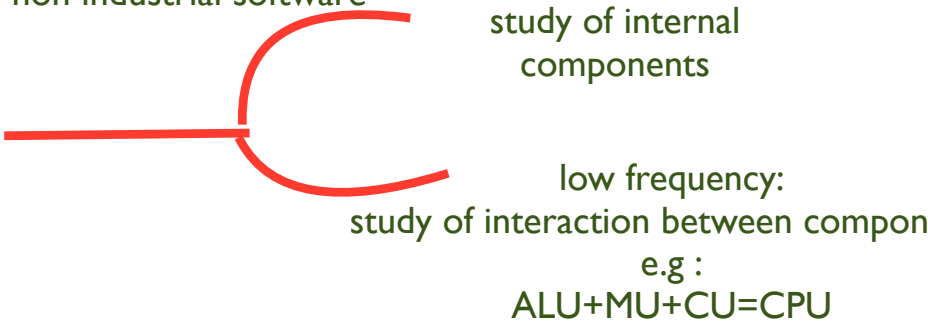
The Problems of Characterizing the Behavior of Discrete Systems

- we have a system with discrete states
- we say that a system is described by a continuous function, we are saying that it can contain no hidden surprises.
  - discrete states cannotbe modeled by continuous functions.

The Five Attributes of a Complex System

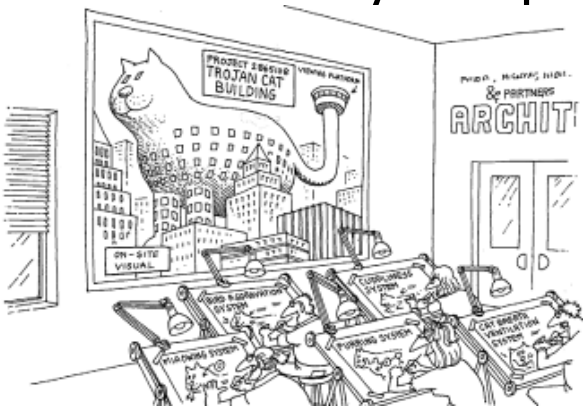
- Considering the nature of this complexity, we conclude that there are five attributes common to all complex systems.
  - **Hierarchic Structure**
  - **Relative Primitives**
  - **Separation of Concerns**
  - **Common Patterns**
  - **Stable Intermediate Forms**

** refers to non industrial software



Hierarchic Structure

- Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached.



- The architecture of a complex system is a function of its components as well as the hierarchic relationships among these components.

Relative Primitives

- The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.
- What is primitive for one observer may be at a much higher level of abstraction for another.

Separation of Concerns

- hierarchic systems *decomposable because they can be divided into* identifiable parts; he calls them *nearly decomposable because their parts are not* completely independent. This leads us to another attribute common to all complex systems:
- Intra component linkages are generally stronger than inter component linkages. This fact has the effect of separating the high-frequency dynamics of the components— involving the internal structure of the components—from the low frequency dynamics—involving interaction among components. [

Common Patterns

- Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements.
- In other words, complex systems have common patterns. These patterns may involve the reuse of small components, such as the cells found in both plants and animals, or of larger structures, such as vascular systems, also found in both plants and animals.

Stable Intermediate Forms

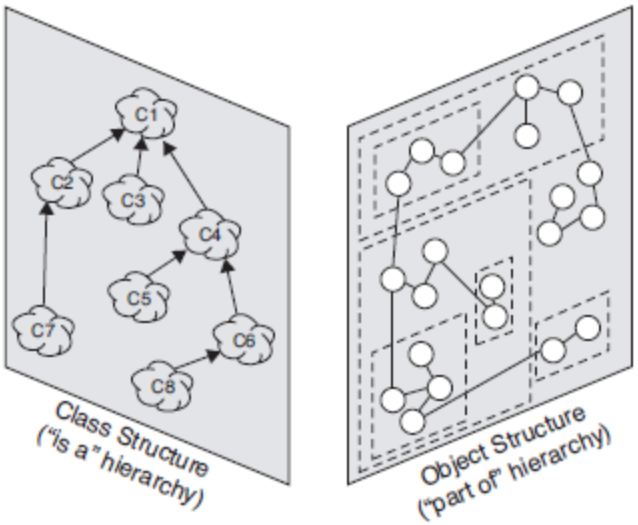
- A complex system that works is invariably found to have evolved from a simple system that worked. . . . A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.

The Canonical Form of a Complex System

- Hierarchy
  - For example, an aircraft may be studied by decomposing it into its propulsion system, flight-control system, and so on. This decomposition represents a structural, or “part of” hierarchy.

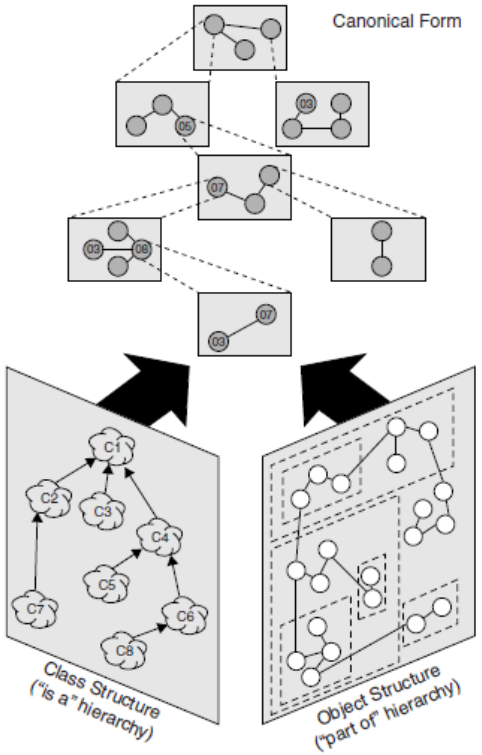
- For example, a turbofan engine is a specific kind of jet engine, and a Pratt and Whitney TF30 is a specific kind of turbofan engine. Stated another way, a jet engine represents a generalization of the properties common to every kind of jet engine; a turbofan engine is simply a specialized kind of jet engine, with properties that distinguish it, for example, from ramjet engines. “is a”

The Canonical Form of a Complex System



The Key Hierarchies of Complex Systems

These hierarchies the *class structure and the object structure of* the system, respectively



The Canonical Form of a Complex System

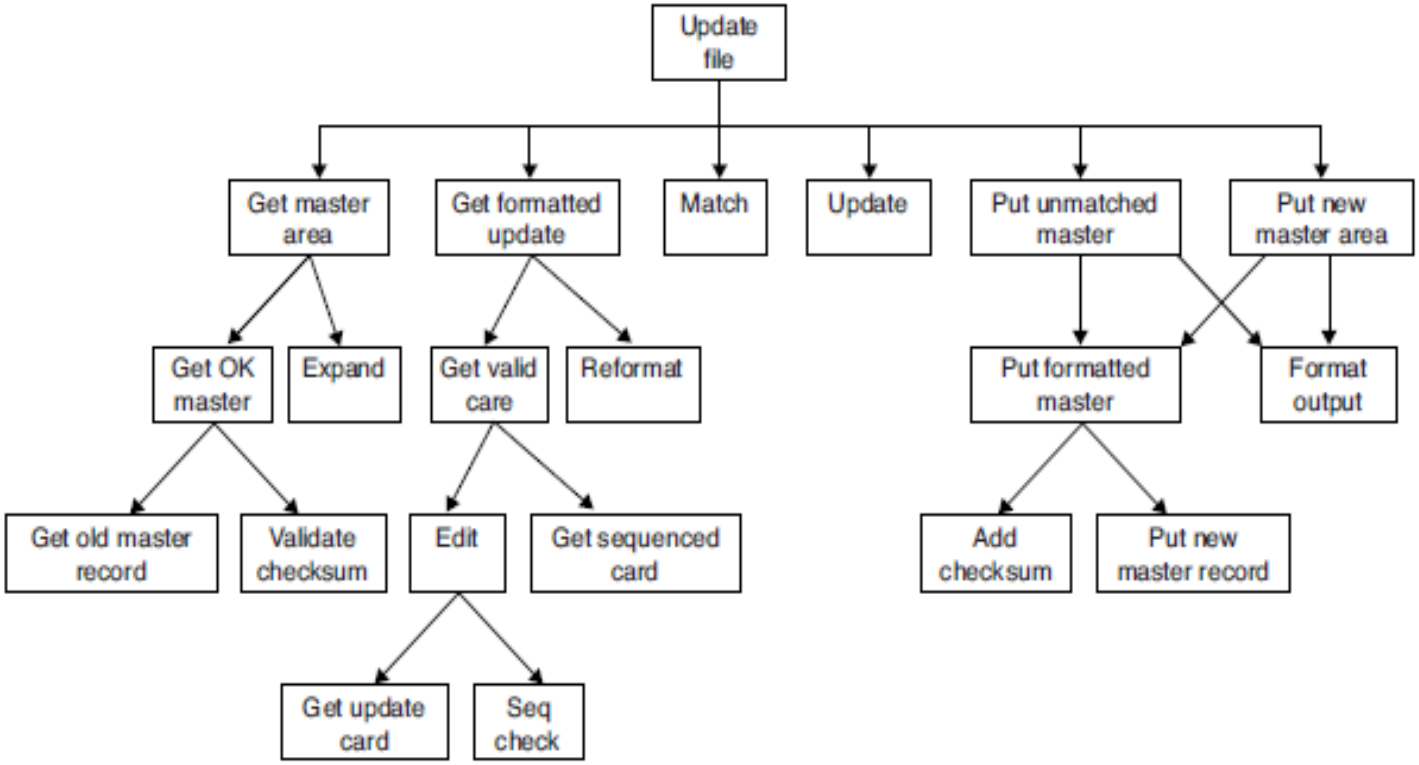
The class and object structures of a system as its *architecture.*

Bringing Order to Chaos

- **The Role of Decomposition**
  - When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently.
  - Parnas observes, intelligent decomposition directly addresses the inherent complexity of software by forcing a division of a system's state space

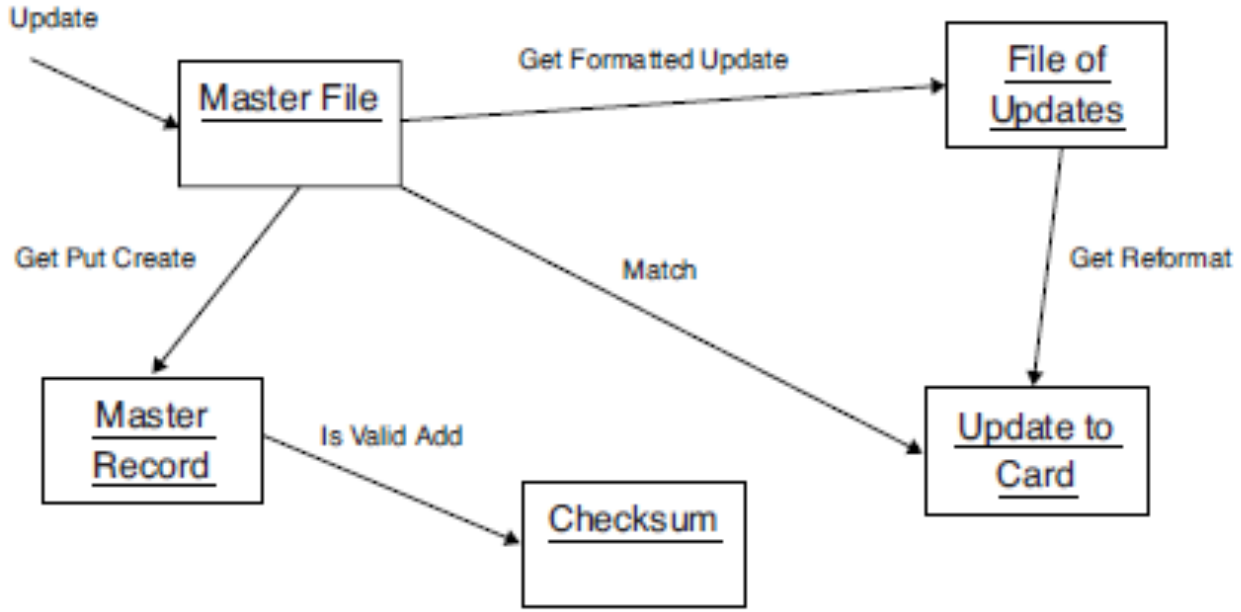
Algorithmic Decomposition

– Most of us have been formally trained in the dogma of top-down structured design, and so we approach decomposition as a simple matter of algorithmic decomposition, wherein each module in the system denotes a major step in some overall process.



Object-Oriented Decomposition

– We suggest that there is an alternate decomposition possible for the same problem.



- an object is simply a tangible entity that exhibits some well-defined behavior.
- Objects do things, and we ask them to perform what they do by sending them messages.
- Because our decomposition is based on objects and not algorithms, we call this an *object-oriented decomposition.*

Algorithmic versus Object-Oriented Decomposition

- The algorithmic view highlights the ordering of events, and the object-oriented view emphasizes the agents that either cause action or are the subjects on which these operations act.

The Role of Abstraction

- The span of absolute judgment and the span of immediate memory impose severe limitations on the amount of information that we are able to receive, process and remember.
- By organizing the stimulus input simultaneously into several dimensions and successively into a sequence of chunks, we manage to break . . . this informational bottleneck

The Role of Hierarchy

- explicitly recognizing the class and object hierarchies within a complex software system.
- The object structure is important because it illustrates how different objects collaborate with one another through patterns of interaction that we call *mechanisms.*
- *The class structure is equally important because it highlights common* structure and behavior within a system

The Role of Hierarchy

- Identifying the hierarchies within a complex software system is often not easy because it requires the discovery of patterns among many objects, each of which may embody some tremendously complicated behavior.
- Once we have exposed these hierarchies, however, the structure of a complex system, and in turn our understanding of it, becomes vastly simplified.

On Designing Complex Systems

- The conception of a design for a new structure can involve as much a leap of the imagination and as much a synthesis of experience and knowledge as any artist is required to bring to his canvas or paper.
- And once that design is articulated by the engineer as artist, it must be analyzed by the engineer as scientist in as rigorous an application of the scientific method as any scientist must make
- The programming challenge is a large-scale exercise in applied abstraction and thus requires the abilities of the formal mathematician blended with the attitude of the competent engineer

- **The Meaning of Design**
  - Satisfies a given (perhaps informal) functional specification
  - Conforms to limitations of the target medium
  - Meets implicit or explicit requirements on performance and resource usage
  - Satisfies implicit or explicit design criteria on the form of the artifact
  - Satisfies restrictions on the design process itself, such as its length or cost,or the tools available for doing the design

- The purpose of design is to create a clean and relatively simple internal structure, sometimes also called an architecture. . . . A design is the end product of the design process
- Design involves balancing a set of competing requirements.
- The products of design are models that enable us to reason about our structures, make trade-offs when requirements conflict, and in general, provide a blueprint for implementation.

The Importance of Model Building

- The building of models has a broad acceptance among all engineering disciplines, largely because model building appeals to the principles of decomposition, abstraction, and hierarchy

The Elements of Software Design Methodologies

- design methods do bring some much-needed discipline to the development process.
- The software engineering community has evolved dozens of different design methodologies, which we can loose
- Despite their differences, all of these have elements in common. Specifically, each includes the following:
  - Notation : The language for expressing each model
  - Process : The activities leading to the orderly construction of the system's models
  - Tools : The artifacts that eliminate the tedium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed

- A sound design method is based on a solid theoretical foundation yet offers degrees of freedom for artistic innovation.

The Models of Object-Oriented Development

- Object-oriented analysis and design is the method that leads us to an object oriented decomposition.
- By applying object-oriented design, we create software that is resilient to change and written with economy of expression.
- We achieve a greater level of confidence in the correctness of our software through an intelligent separation of its state space.
- Ultimately, we reduce the risks inherent in developing complex software systems.

The Object Model

- Object-oriented technology is built on a sound engineering foundation, whose elements we collectively call the *object model of development or* simply the *object model.*
- *The object model encompasses the principles of* abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence.

The Evolution of the Object Model

- History of software engineering notice two sweeping trends:
  - The shift in focus from programming-in-the-small to programming-in-the large
  - The evolution of high-order programming languages

The Generations of Programming Languages

- Wegner has classified some of the more popular high-order programming languages in generations arranged according to the language features they first introduced
- First-generation languages (1954–1958)
  - FORTRAN I Mathematical expressions
  - ALGOL 58 Mathematical expressions
  - Flowmatic Mathematical expressions
  - IPL V Mathematical expressions

- Second-generation languages (1959–1961)
  - FORTRAN II Subroutines, separate compilation
  - ALGOL 60 Block structure, data types
  - COBOL Data description, file handling
  - Lisp List processing, pointers, garbage collection

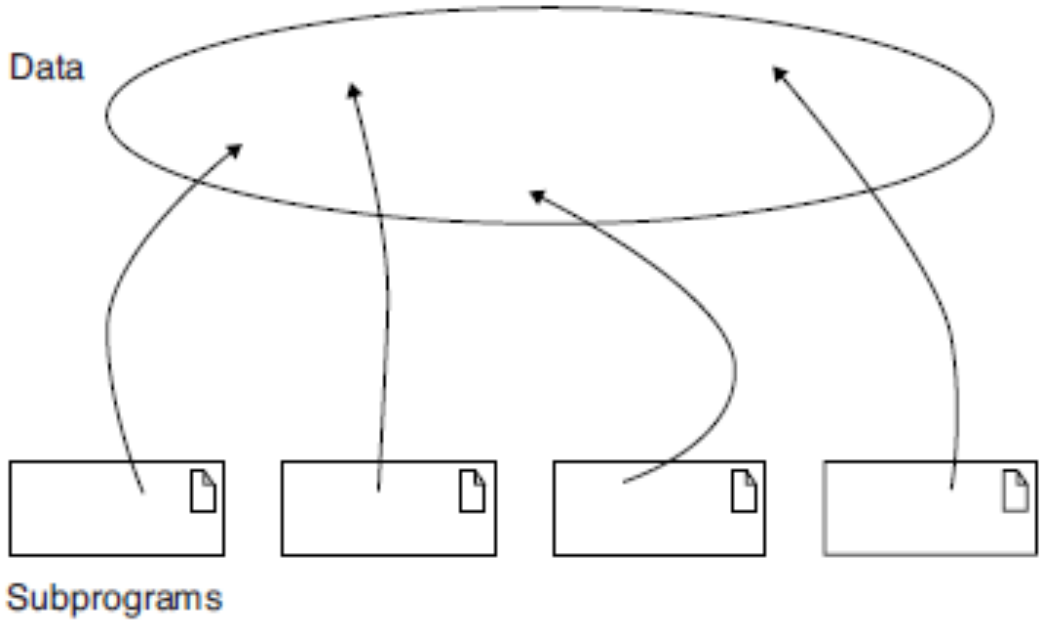
- Third-generation languages (1962–1970)
  - PL/1 FORTRAN + ALGOL + COBOL
  - ALGOL 68 Rigorous successor to ALGOL 60
  - Pascal Simple successor to ALGOL 60
  - Simula Classes, data abstraction

- The generation gap (1970–1980)
- Many different languages were invented, but few endured.
- However, the following are worth noting:
  - C Efficient; small executables
  - FORTRAN 77 ANSI standardization

- Let’s expand on Wegner’s categories.
- Object-orientation boom (1980–1990, but few languages survive)
  - Smalltalk 80 Pure object-oriented language
  - C++ Derived from C and Simula
  - Ada83 Strong typing; heavy Pascal influence
  - Eiffel Derived from Ada and Simula

- Emergence of frameworks (1990–today)
- Much language activity, revisions, and standardization have occurred, leading to programming frameworks.
  - Visual Basic — Eased development of the graphical user interface (GUI) for Windows applications
  - Java — Successor to Oak; designed for portability
  - Python — Object-oriented scripting language
  - J2EE — Java-based framework for enterprise computing
  - .NET — Microsoft's object-based framework
  - Visual C# — Java competitor for the Microsoft .NETFramework
  - Visual Basic .NET — Visual Basic for the Microsoft .NET Framework

The Topology of First- and Early Second-Generation Programming Languages

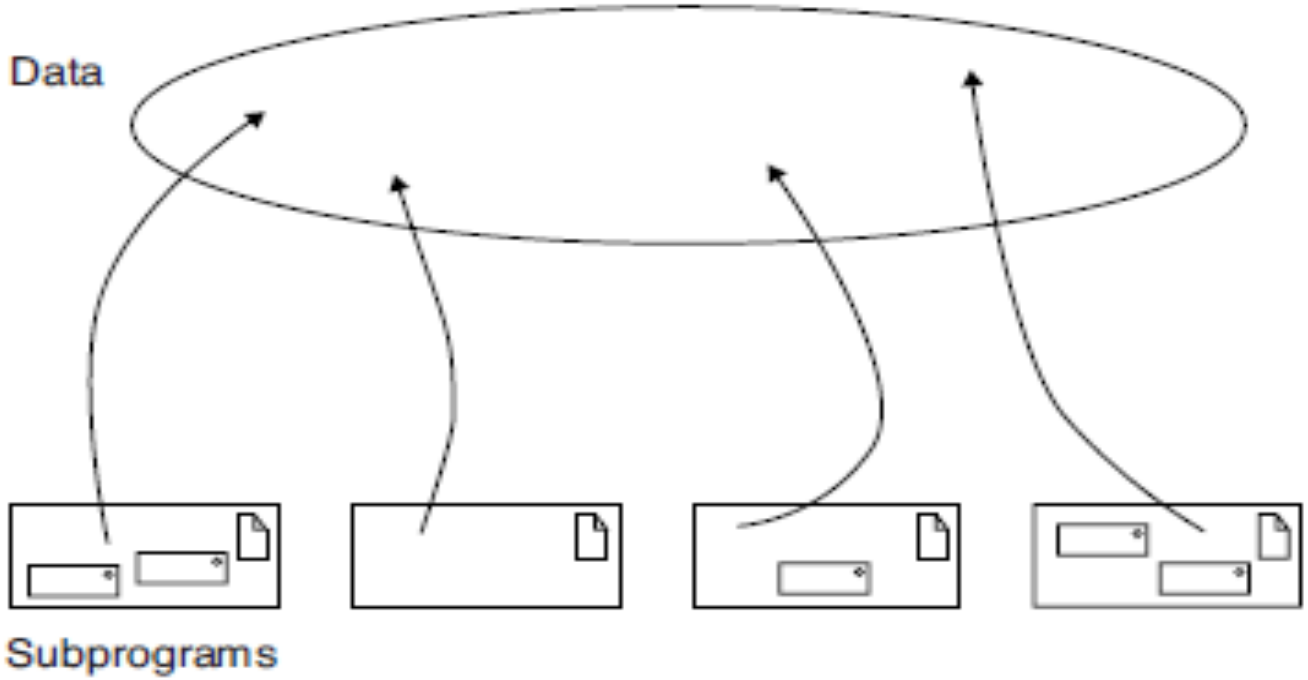


The Topology of First- and Early Second-Generation Programming Languages

The Topology of First- and Early Second-Generation Programming Languages

- The basic physical building block of all applications is the subprogram
- Flat physical structure, consisting only of global data and subprograms
- Arrows in this figure indicate dependencies of the subprograms on various data
- Logically separate different kinds of data from one another
- An error in one part of a program can have a devastating ripple effect across the rest of the system
- When modifications are made to a large system, it is difficult to maintain the integrity of the original design

The Topology of Late Second- and Early Third-Generation Programming

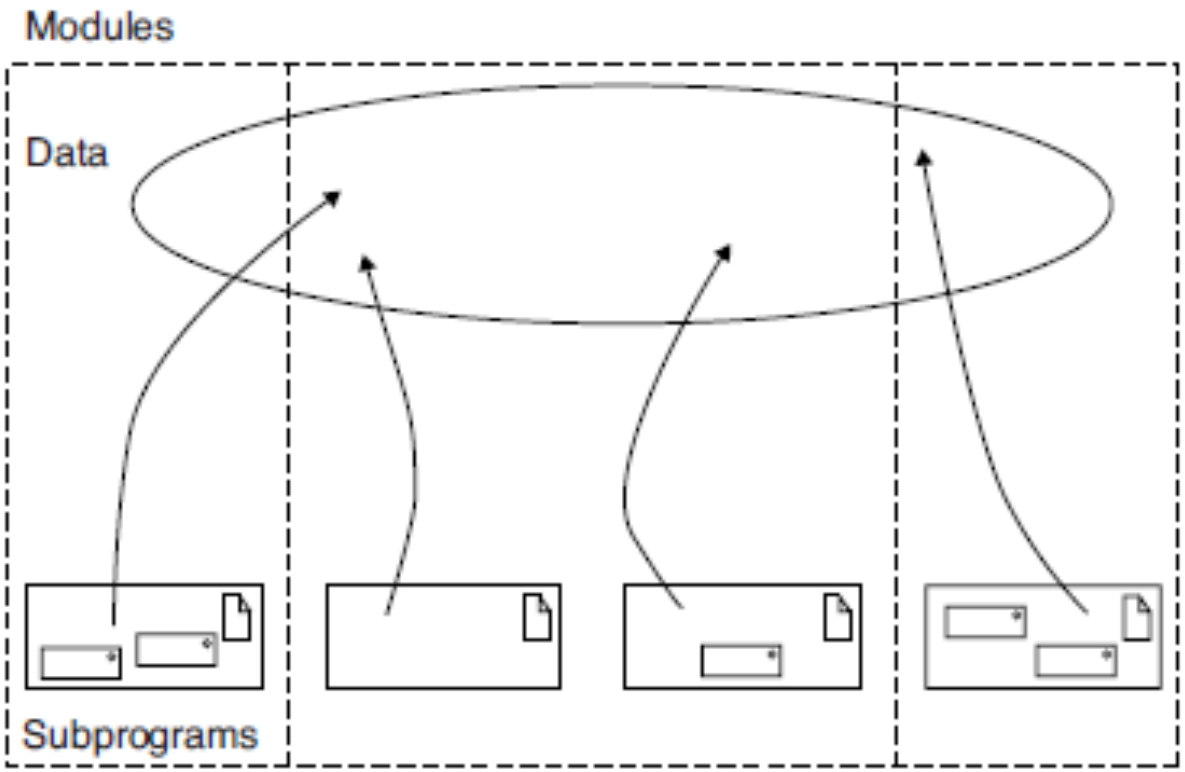


- The Topology of Late Second- and Early Third-Generation Programming Languages

The Topology of Late Second- and Early Third-Generation Programming

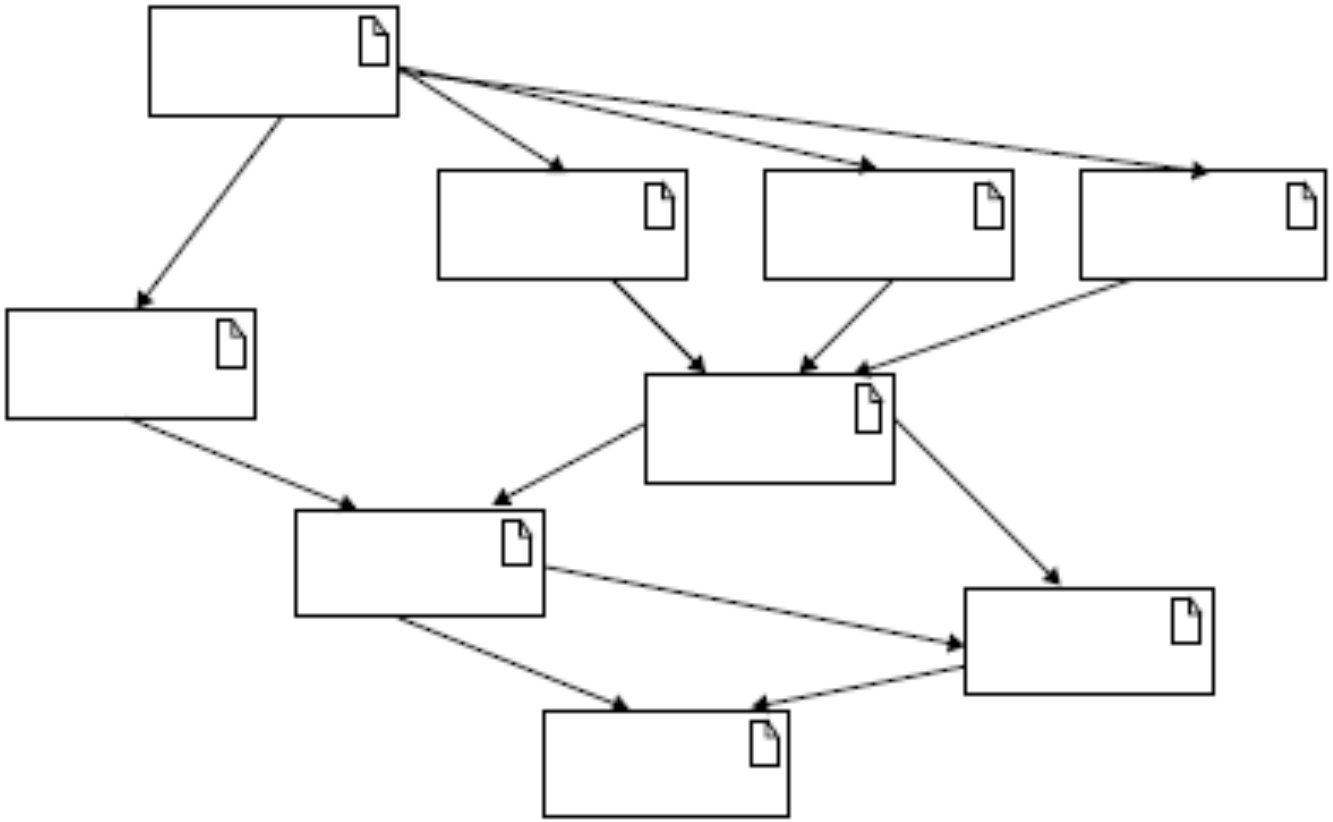
- The first software abstraction, now called the ‘procedural’ abstraction, grew directly out of this pragmatic view of software
- The realization that subprograms could serve as an abstraction mechanism had three important consequences.
- First, languages were invented that supported a variety of parameter-passing mechanisms.
- Second, the foundations of structured programming were laid, manifesting themselves in language support for the nesting of subprograms and the development of theories regarding control structures and the scope and visibility of declarations.
- Third, structured design methods emerged, offering guidance to designers trying to build large systems using subprograms as basic physical building blocks

The Topology of Late Third-Generation Programming Languages



- Larger programming projects meant larger development teams, and thus the need to develop different parts of the same program independently.
- The answer to this need was the separately compiled module, which in its early conception was little more than an arbitrary container for data and subprograms,

The Topology of Object-Based and Object-Oriented Programming Languages

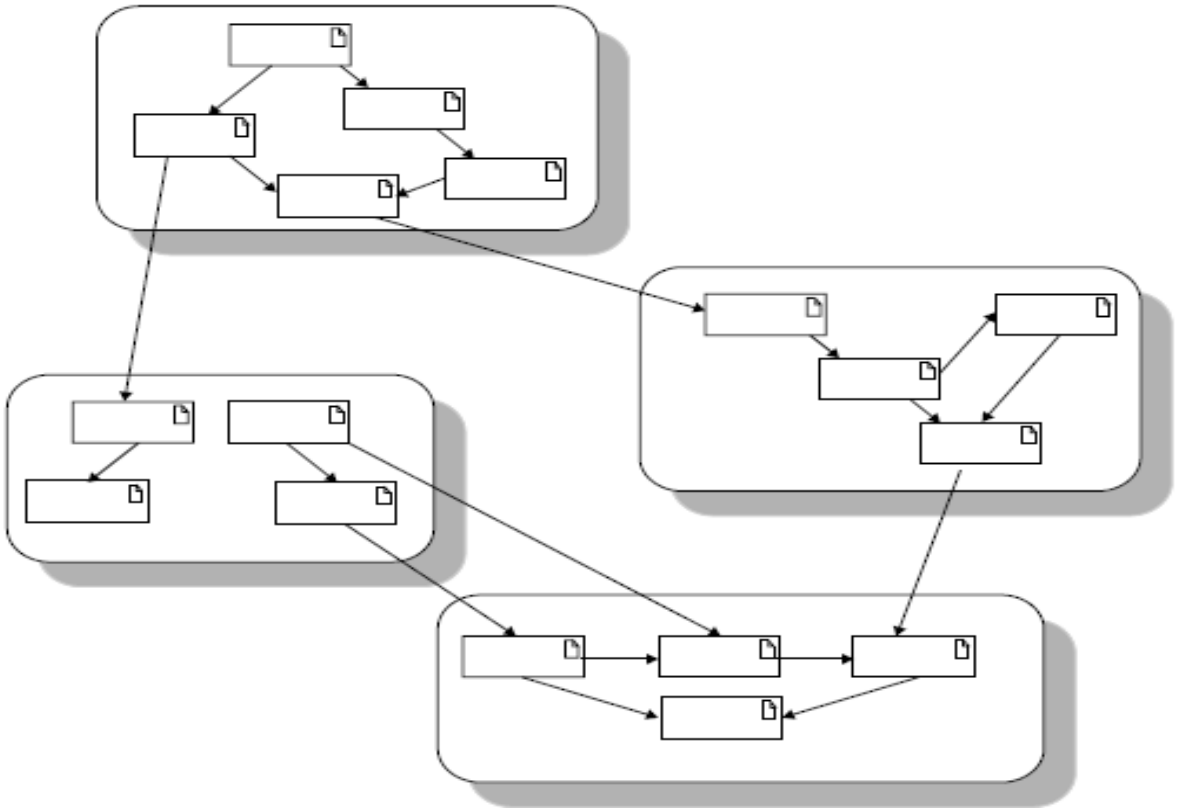


The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

The Topology of Object-Based and Object-Oriented Programming Languages

- A logical collection of classes and objects
- If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns
- For very complex systems, we find that classes, objects, and modules provide an essential yet insufficient means of abstraction.
- Fortunately, the object model scales up. In large systems, we find clusters of abstractions built in layers on top of one another.
- At any given level of abstraction, we find meaningful collections of objects that collaborate to achieve some higher-level behavior.

The Topology of Object-Based and Object-Oriented Programming Languages



The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

Foundations of the Object Model

- The Object Model, the object model has proven to be a unifying concept in computer science, applicable not just to programming languages but also to the design of user interfaces, databases, and even computer architectures
- An object orientation helps us to cope with the complexity inherent in many different kinds of systems
- A Smalltalk programmer uses methods, a C++ programmer uses virtual member functions, and a CLOS programmer uses generic functions.
- An Object Pascal programmer talks of a type coercion;
- An Ada programmer calls the same thing a type conversion;
- A C# or Java programmer would use a cast.

Foundations of the Object Model

- Stefik and Bobrow define objects as “entities that combine the properties of procedures and data since they perform computations and save local state”
- Jones further clarifies this term by noting that “in the object model, emphasis is placed on crisply characterizing the components of the physical or abstract system to be modeled by a programmed system. . . . Objects have a certain ‘integrity’ which should not—in fact, cannot—be violated. An object can only change state, behave, be manipulated, or stand in relation to other objects in ways appropriate to that object.
- Stated differently, there exist invariant properties that characterize an object and its behavior.

Object-Oriented Programming

- Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Object-Oriented Programming

- There are three important parts to this definition:

  (1) Object-oriented programming uses objects, not algorithms, as its fundamental logical building blocks (the “part of” hierarchy);

  (2) each object is an instance of some class; and

  (3) classes may be related to one another via inheritance relationships (the “is a” hierarchy)

Object-Oriented Programming

- A language is object-oriented if and only if it satisfies the following requirements:
  - It supports objects that are data abstractions with an interface of named operations and a hidden local state.
  - Objects have an associated type [class].
  - Types [classes] may inherit attributes from supertypes [superclasses]

Object-Oriented Design

- Object-oriented design is a method of design encompassing the process of object oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

Object-Oriented Design

- There are two important parts to this definition: object-oriented design

  (1) leads to an object-oriented decomposition and

  (2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, in addition to the static and dynamic aspects of the system.

Object-Oriented Analysis

- Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

- the products of object-oriented analysis serve as the models from which we may start an object-oriented design;
- the products of object-oriented design can then be used as blueprints for completely implementing a system using object-oriented programming methods.

Elements of the Object Model

- Jenkins and Glasgow observe that “most programmers work in one language and use only one programming style. They program in a paradigm enforced by the language they use. Frequently, they have not been exposed to alternate ways of thinking about a problem, and hence have difficulty in seeing the advantage of choosing a style more appropriate to the problem at hand”

- Bobrow and Stefik define a programming style as “a way of organizing programs on the basis of some conceptual model of programming and an appropriate language to make programs written in the style clear”

- They further suggest that there are five main kinds of programming styles, listed here with the kinds of abstractions they employ:
  - Procedure-oriented Algorithms
  - Object-oriented Classes and objects
  - Logic-oriented Goals, often expressed in a predicate calculus
  - Rule-oriented If–then rules
  - Constraint-oriented Invariant relationships

- For all things object-oriented, the conceptual framework is the object model.
- There are four major elements of this model:
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy

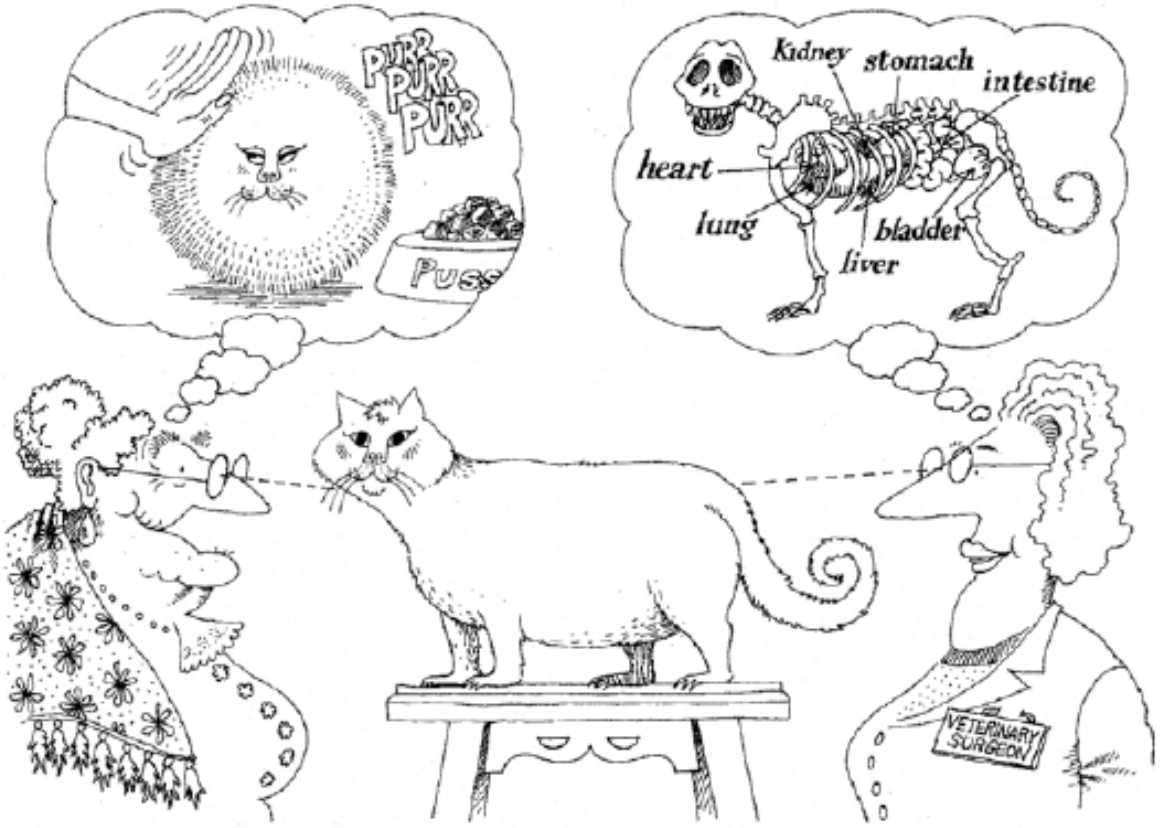
  By *major, we mean that a model without any one of these elements is not object oriented.*

- There are three minor elements of the object model:
  - Typing
  - Concurrency
  - Persistence
- By *minor, we mean that each of these elements is a useful, but not essential, part* of the object model.

The Meaning of Abstraction

- An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

The Meaning of Abstraction



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

- From the most to the least useful, these kinds of abstractions include the following:

1. Entity abstraction

   An object that represents a useful model of a problem domain or solution domain entity

2. Action abstraction

   An object that provides a generalized set operations, all of which perform the same kind of function

3. Virtual machine abstraction

   An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations

4. Coincidental abstraction

   An object that packages a set of operations that have no relation to each other

Examples of Abstraction

** temperature, location
and setpoint are private and unaccessible outside of class
Active Temperature Sensor

Abstraction: Temperature Sensor
Important Characteristics: temperature location
Responsibilities: report current temperature calibrate

Abstraction of a Temperature Sensor

Abstraction: Active Temperature Sensor
Important Characteristics: temperature location setpoint
Responsibilities: report current temperature calibrate establish setpoint

Abstraction of an Active Temperature Sensor

The Meaning of Encapsulation

- Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Examples of Encapsulation

Encapsulation hides the details of the implementation of an object.

Abstraction: Heater
Important Characteristics: location status
Responsibilities: turn on turn off provide status

Related Candidate Abstractions: Heater Controller, Temperature Sensor

The Meaning of Modularity

"The act of partitioning a program into individual components can reduce its complexity to some degree. . . . Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well-defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program"

Modularity packages abstractions into discrete units.

- Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- Two additional technical issues can affect modularization decisions.
- First, since modules usually serve as the elementary and indivisible units of software that can be reused across applications, a developer might choose to package classes and objects into modules in a way that makes their reuse convenient.
- Second, many compilers generate object code in segments, one for each module.
- Therefore, there may be practical limits on the size of individual modules.

The Meaning of Hierarchy

- Hierarchy is a ranking or ordering of abstractions.
- The two most important hierarchies in a complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

Class

Object

Abstractions form a hierarchy.

Hierarchy

- Semantically, inheritance denotes an “is a” relationship. For example, a bear “is a” kind of mammal, a house “is a” kind of tangible asset, and a quick sort “is a” particular kind of sorting algorithm
  - ***Single Inheritance***
  - ***Multiple Inheritance***
  - ***Aggregation***

The Meaning of Typing

- A type is a precise characterization of structural or behavioral properties which a collection of entities all share
- Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

Strongly Typed language:
Case-sensitive

Weakly Typed langauge:
Case-insensitive

- There are a number of important benefits to be derived from using strongly typed languages:
  - Without type checking, a program in most languages can ‘crash’ in mysterious ways at runtime.
  - In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable.
  - Type declarations help to document programs.
  - Most compilers can generate more efficient object code if types are declared.

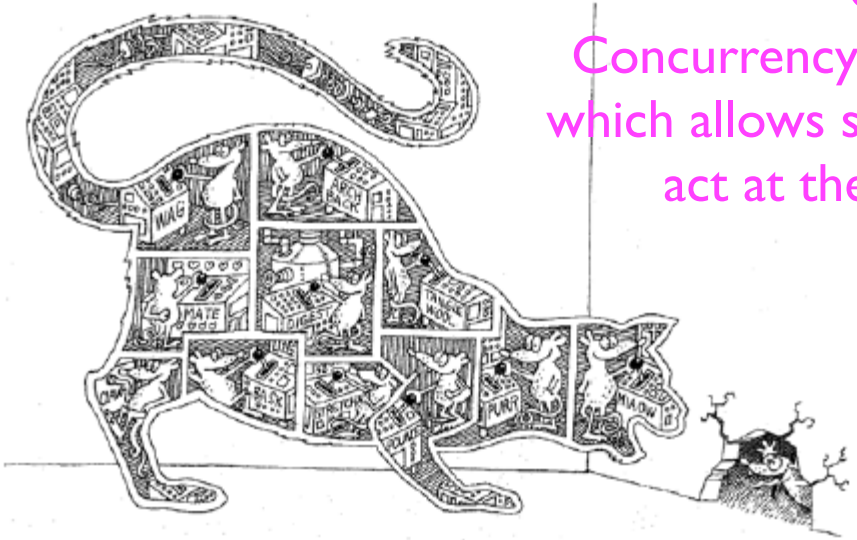
Static and Dynamic Typing

- The concepts of *strong and weak typing and static and dynamic typing are* entirely different.
- Strong and weak typing refers to *type consistency, whereas* static and dynamic typing refers to the *time when names are bound to types.*
- *Static* typing (also known as *static binding or early binding) means that the types of all* variables and expressions are fixed at the time of compilation; dynamic typing (also known as *late binding) means that the types of all variables and expressions* are not known until runtime

The Meaning of Concurrency

- Concurrency is the property that distinguishes an active object from one that is not active.

OR
Concurrency is the property which allows several objects to act at the same time



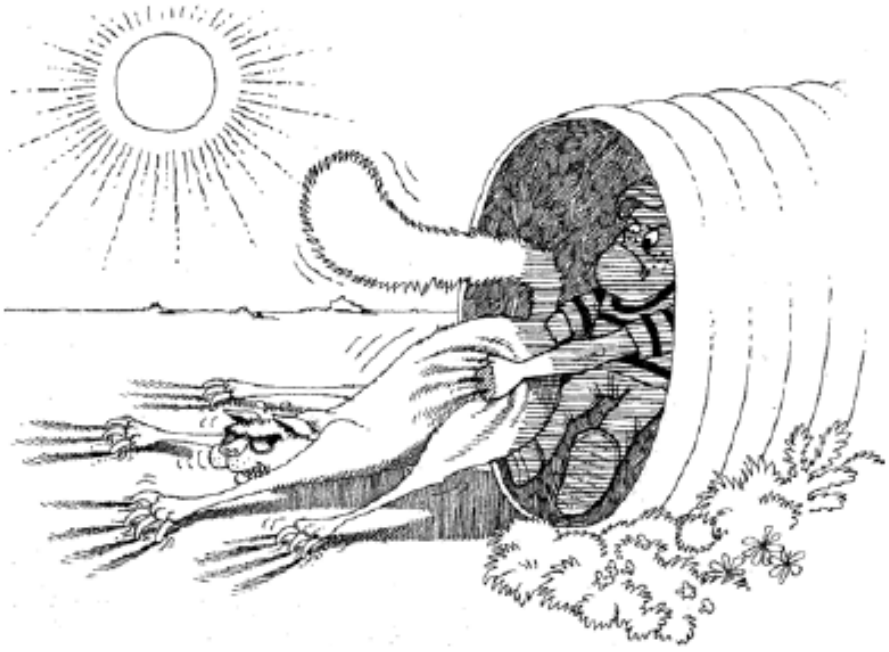
Concurrency allows different objects to act at the same time.

- We also distinguish between *heavyweight and lightweight concurrency.*
- *A heavyweight* process is one that is typically independently managed by the target operating system and so encompasses its own address space. e.g, Struct
- A lightweight process usually lives within a single operating system process along with other lightweight processes, which share the same address space. e.g, Union
- Communication among heavyweight processes is generally expensive, involving some form of interprocess communication; communication among lightweight processes is less expensive and often involves shared data.

The Meaning of Persistence

- Persistence is the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created).

Object continues to exist post deallocation of space



Persistence saves the state and class of an object across time or space.

Applying the Object Model

*introduces novelty

- object model is fundamentally different from the models embraced by the more traditional methods of structured analysis, structured design, and structured programming
- This does not mean that the object model abandons all of the sound principles and experiences of these older methods.
- Rather, it introduces several novel elements that build on these earlier models.
- Thus, the object model offers a number of significant benefits that other models simply do not provide.
- Most importantly, the use of the object model leads us to construct systems that embody the five attributes of well-structured complex systems

Benefits of the Object Model

- First, the use of the object model helps us to exploit the expressive power of object-based and object-oriented programming languages.
- Second, the use of the object model encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frameworks
- Third, the use of the object model produces systems that are built on stable intermediate forms, which are more resilient to change.

Classes and Objects

- we use object-oriented methods to analyze or design a complex software system, our basic building blocks are classes and objects
- objects have a permanence and identity apart from any operations on them.

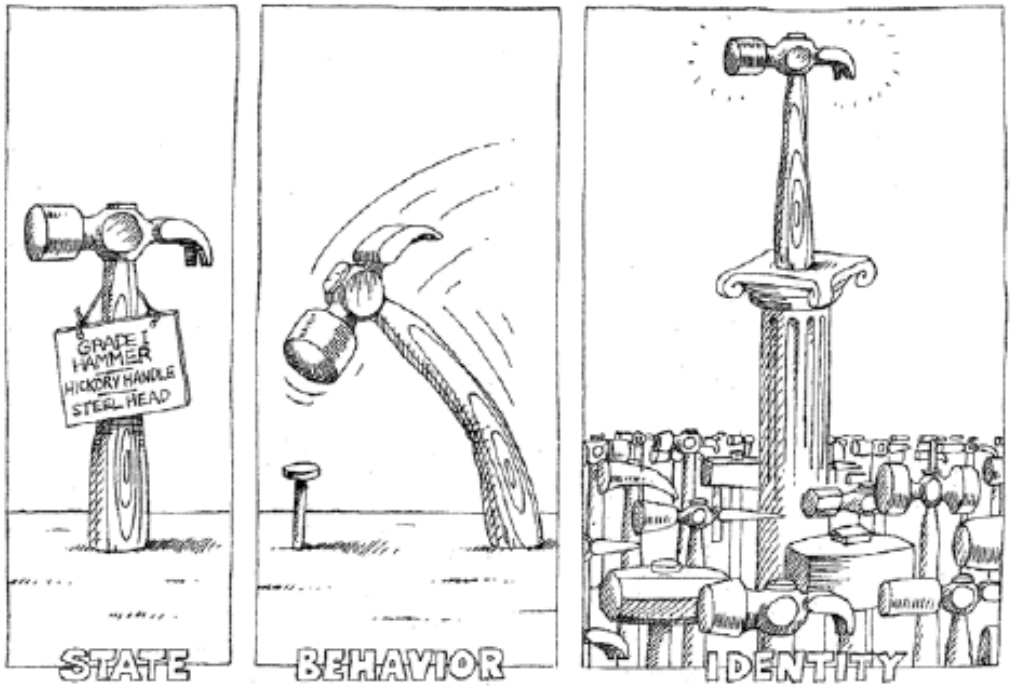
What Is and What Isn't an Object

- an object is any of the following:
  - A tangible and/or visible thing
  - Something that may be comprehended intellectually
  - Something toward which thought or action is directed

- Jacobson et al. define *control objects as "the ones that unite courses of events and thus will carry on communication* with other objects" [62].
- This leads us to the more refined definition of Smith and Tockey, who suggest that "an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain"

- Some objects may have crisp conceptual boundaries yet represent intangible events or processes.
- For example, a chemical process in a manufacturing plant may be treated as an object because it has a crisp conceptual boundary, interacts with certain other objects through a well-ordered collection of operations that unfolds over time, and exhibits a well-defined behavior.

An object is an entity that has state, behavior, and identity. The structure and behavior of similar objects are defined in their common class. The terms *instance* and *object are interchangeable.*

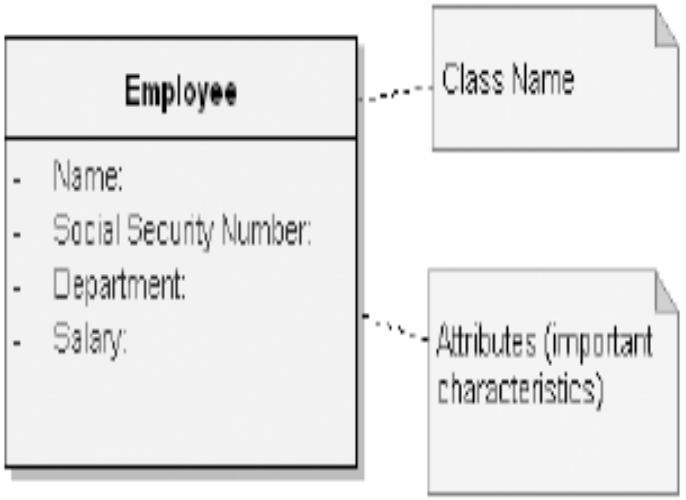


An object has state, exhibits some well-defined behavior, and has a unique identity.

State

The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

Properties: Static
Values: Dynamic (Current)



Employee Class with Attributes

Employee Objects Tom and Kaitlyn

Behavior

- Behavior is how an object acts and reacts, in terms of its state changes and message passing. e.g, push() and pop() in a Stack
- In other words, the behavior of an object represents its outwardly visible activity.
- An operation is some action that one object performs on another in order to elicit a reaction.
- For example, a client might invoke the operations append and pop to grow and shrink a queue object, respectively.
- A client might also invoke the operation length, which returns a value denoting the size of the queue object but does not alter the state of the queue itself.
- Message passing is one part of the equation that defines the behavior of an object; our definition for behavior also notes that the state of an object affects its behavior as well.

Operations

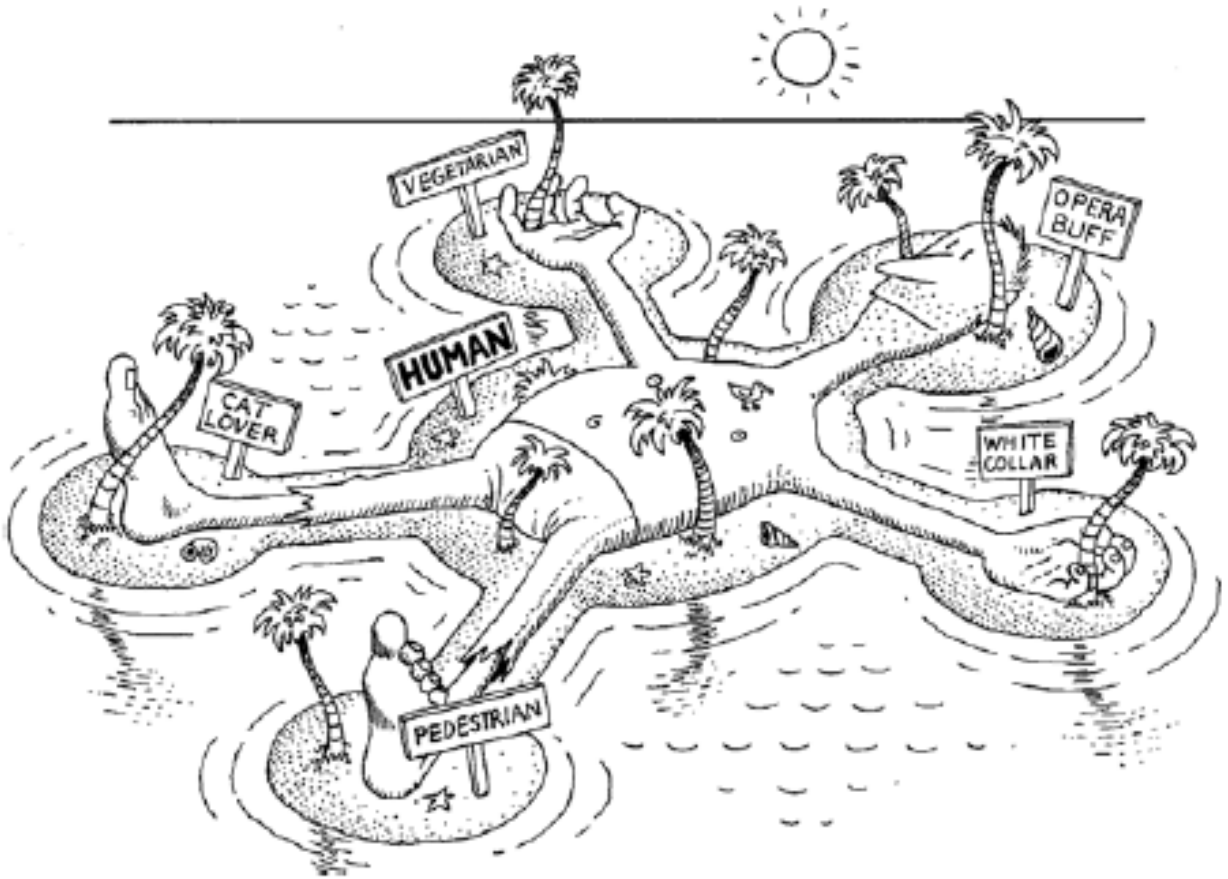
- An operation denotes a service that a class offers to its clients. In practice, we have found that a client typically performs five kinds of operations on an object.
- The three most common kinds of operations are the following:
  - Modifier: an operation that alters the state of an object
  - Selector: an operation that accesses the state of an object but does not alter the state
  - Iterator: an operation that permits all parts of an object to be accessed in some well-defined order

Access+Alter

- Two other kinds of operations are common; they represent the infrastructure necessary to create and destroy instances of a class.
  - Constructor: an operation that creates an object and/or initializes its state
  - Destructor: an operation that frees the state of an object and/or destroys the object itself

Roles and Responsibilities

- A role is a mask that an object wears [8] and so defines a contract between an abstraction and its clients.
- “Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports”



Objects can play many different roles.

Objects as Machines

- The existence of state within an object means that the order in which operations are invoked is important.
- we may classify objects as either active or passive.
- An active object is one that encompasses its own thread of control, whereas a passive object does not.
- Active objects are generally autonomous, meaning that they can exhibit some behavior without being operated on by another object.
- Passive objects, on the other hand, can undergo a state change only when explicitly acted on.
- In this manner, the active objects in our system serve as the roots of control.

Identity

- Khoshafian and Copeland offer the following definition for identity: “Identity is that property of an object which distinguishes it from all other objects”
- “most programming and database languages use variable names to distinguish temporary objects, mixing addressability and identity. Most database systems use identifier keys to distinguish persistent objects, mixing data value and identity.”
- The failure to recognize the difference between the name of an object and the object itself is the source of many kinds of errors in object oriented programming.

Relationships among Objects

- The relationship between any two objects encompasses the assumptions that each makes about the other, including what operations can be performed and what behavior results.
- We have found that two kinds of object relationships are of particular interest in object-oriented analysis and design, namely:
- 1. Links
- 2. Aggregation

Links

- The term *link derives from Rumbaugh et al., who define it as a* loosely coupled
- *"physical or conceptual* connection between objects"
- An object collaborates with other objects through its links to these objects.
- Stated another way, a link denotes the specific association through which one object (the client) applies the services of another object (the supplier), or through which one object may navigate to another.

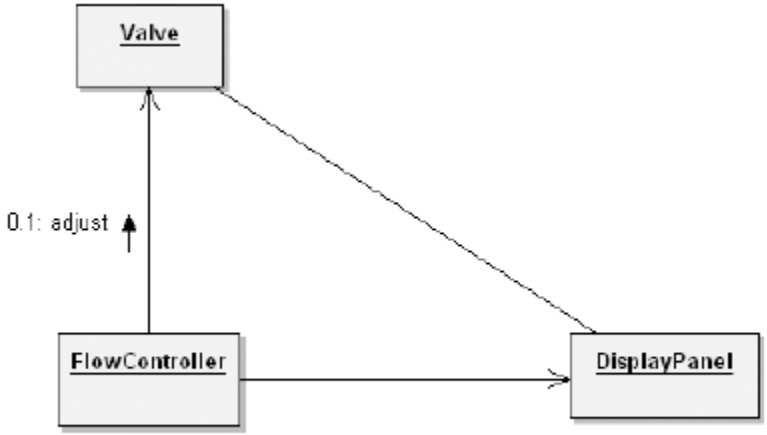


Figure 3–5 Links

Visibility

- Consider two objects, A and B, with a link between the two. In order for A to send a message to B, B must be visible to A in some manner.
- During our analysis of a problem, we can largely ignore issues of visibility, but once we begin to deviseconcrete implementations, we must consider the visibility across links
- Because our decisions here dictate the scope and access of the objects on each side of a link.

Synchronization

- Whenever one object passes a message to another across a link, the two objects are said to be synchronized.
- when one active object has a link to a passive one, we must choose one of three approaches to synchronization.

  1. Sequential: The semantics of the passive object are guaranteed only in the presence of a single active object at a time. Single sender(active)—>Single receiver(passive)

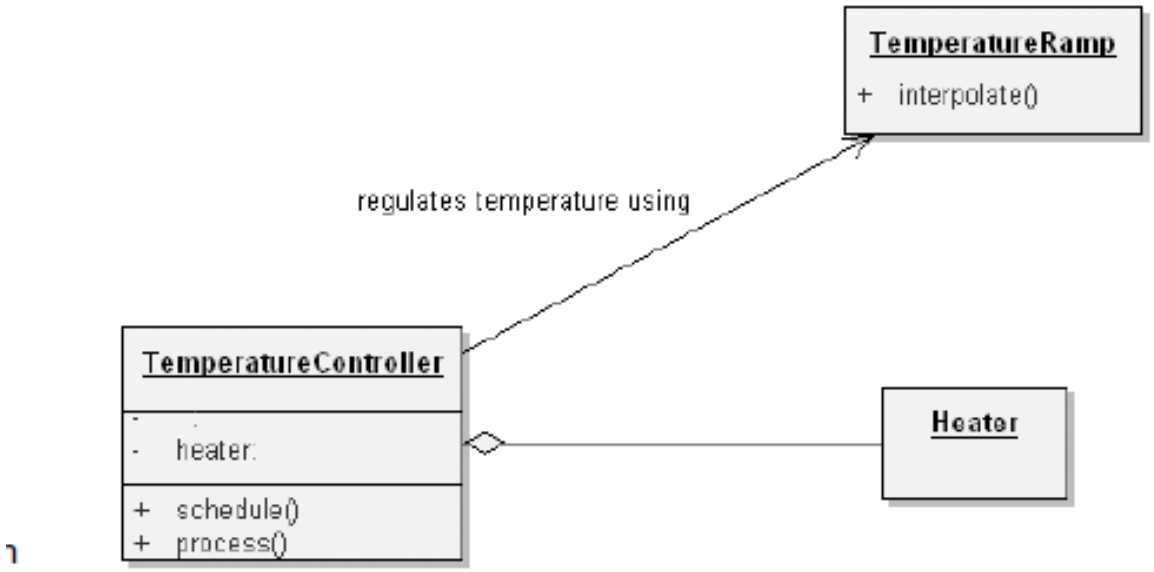
  2. Guarded: The semantics of the passive object are guaranteed in the presence of multiple threads of control, but the active clients must collaborate to achieve mutual exclusion.

  3. Concurrent: The semantics of the passive object are guaranteed in the presence of multiple threads of control, and the supplier guarantees mutual exclusion.

Aggregation

tightly coupled

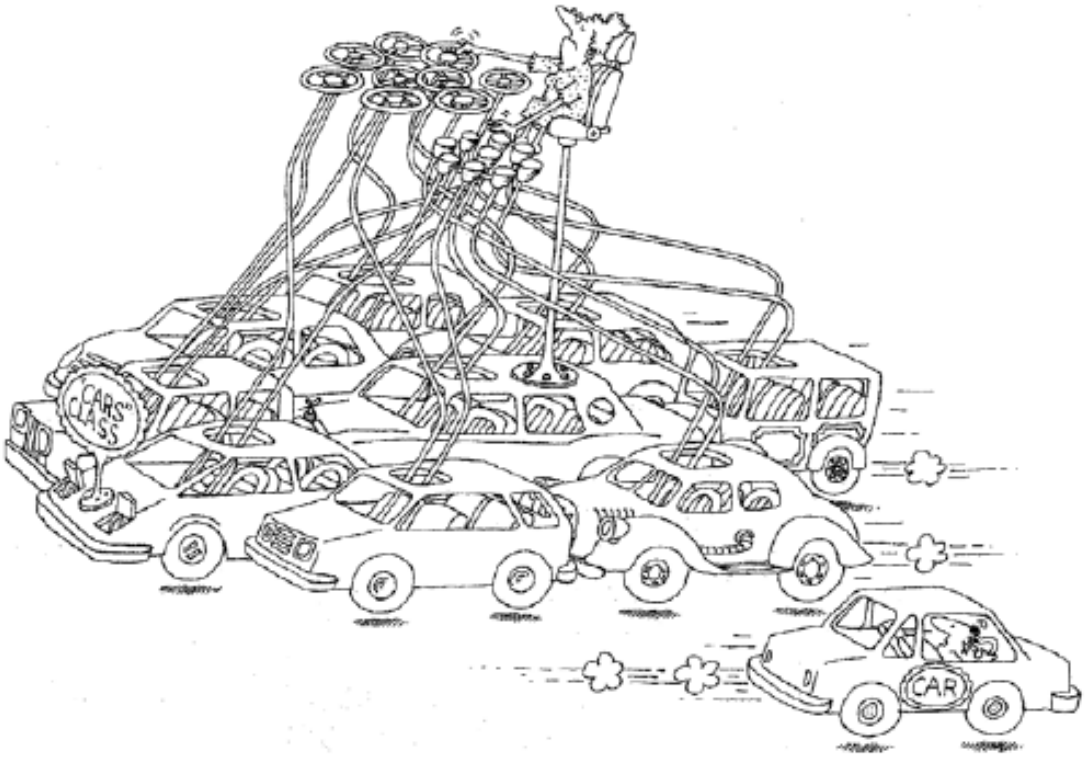
- Whereas links denote peer-to-peer or client/supplier relationships, aggregation denotes a whole/part hierarchy, with the ability to navigate from the whole (also called the *aggregate) to its parts*



- There are clear trade-offs between links and aggregation.
- Aggregation is sometimes better because it encapsulates parts as secrets of the whole.
- Links are sometimes better because they permit looser coupling among objects.
- Intelligent engineering decisions require careful weighing of these two factors.

The Nature of a Class

- *Webster's Third New International Dictionary defines a class* as "a group, set, or kind marked by common attributes or a common attribute; a group division, distinction, or rating based on quality, degree of competence, or condition"
- In the context of object-oriented analysis and design, we define a class as follows:
- A class is a set of objects that share a common structure, common behavior, and common semantics.
- A single object is simply an instance of a class.



A class represents a set of objects that share a common structure and a common behavior.

- What isn't a class?

  An object is not a class. Objects that share no common structure and behavior cannot be grouped in a class because, by definition, they are unrelated except by their general nature as objects.

Interface and Implementation

- The interface of a class provides its outside view and therefore emphasizes the abstraction while hiding its structure and the secrets of its behavior.
- This interface primarily consists of the declarations of all the operations applicable to instances of this class, but it may also include the declaration of other classes, constants, variables, and exceptions as needed to complete the abstraction.
- By contrast, the implementation of a class is its inside view, which encompasses the secrets of its behavior.
- The implementation of a class primarily consists of the implementation of all of the operations defined in the interface of the class.

- We can further divide the interface of a class into four parts:

  1. Public: a declaration that is accessible to all clients

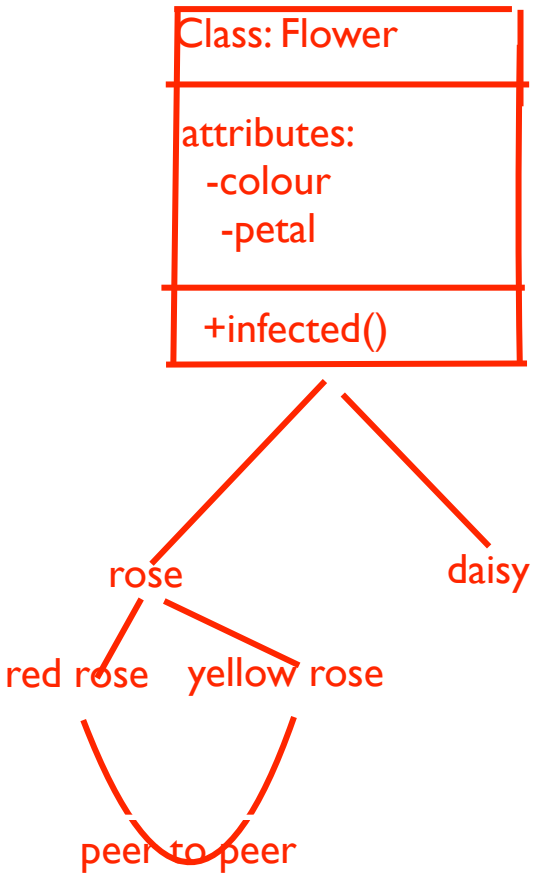
  2. Protected: a declaration that is accessible only to the class itself and its subclasses

  3. Private: a declaration that is accessible only to the class itself

  4. Package: a declaration that is accessible only by classes in the same package

Relationships among Classes

- Consider for a moment the similarities and differences among the following classes of objects: flowers, daisies, red roses, yellow roses, petals, and ladybugs.
- We can make the following observations.
  - A daisy is a kind of flower.
  - A rose is a (different) kind of flower.
  - Red roses and yellow roses are both kinds of roses.
  - A petal is a part of both kinds of flowers.
  - Ladybugs eat certain pests such as aphids, which may be infesting certain kinds of flowers.


Class: Flower
attributes:
-colour
-petal
+infected()
rose
daisy
red rose
yellow rose
peer to peer

1.) Relationship among Objects:

a.) Link
b.)Visibility
c.)Synchronization
d.) Aggregation

2. Implementation

3. Interface

4.)Relationship between Classes:

1. Association
2. Semantic Dependencies
3. Multiplicity
4. Inheritance

a.) Single inheritance
b.) Multiple Inheritance
c.) Polymorphism

5. Aggregation
6. Dependencies

1. “is a “
2. “part of”
3. association

- We establish relationships between two classes for one of two reasons.
- First, a class relationship might indicate some sort of sharing.
- For example, daisies and roses are both kinds of flowers, meaning that both have brightly colored petals, both emit a fragrance, and so on.
- Second, a class relationship might indicate some kind of semantic connection.
- Thus, we say that red roses and yellow roses are more alike than are daisies and roses, and daisies and roses are more closely related than are petals and flowers.

- In all, there are three basic kinds of class relationships [22].
- The first of these is generalization/ specialization, denoting an[1] "is a" relationship.
- For instance, a rose is a kind of flower, meaning that a rose is a specialized subclass of the more general class, flower.
- The second is whole/part, which denotes a[2] "part of" relationship.
- Thus, a petal is not a kind of a flower; it is a part of a flower
- The third is[3] association, which denotes some semantic dependency among otherwise unrelated classes, such as between ladybugs and flowers.

Association

- Of these different kinds of class relationships, associations are the most general but also the most semantically weak.
- The identification of associations among classes is often an activity of analysis and early design, at which time we begin to discover the general dependencies among our abstractions.

- ***Semantic Dependencies***
  - an association only denotes a semantic dependency and does not state the direction of this dependency (unless otherwise stated, an association implies bidirectional navigation, as in our example),
  - nor does it state the exact way in which one class relates to another (we can only imply these semantics by naming the role each class plays in relationship with the other).
  - However, these semantics are sufficient during the analysis of a problem, at which time we need only to identify such dependencies.

- Each instance of Wheel relates to one Vehicle, and each instance of Vehicle may have many Wheels (noted by the *).

Association

- ***Multiplicity***
  - In practice, there are three common kinds of multiplicity across an association:
    1. One-to-one 1....1
    2. One-to-many 1....*
    3. Many-to-many *....*
  - A one-to-one relationship denotes a very narrow association.a one-to-one relationship between the class Sale and the class CreditCardTransaction

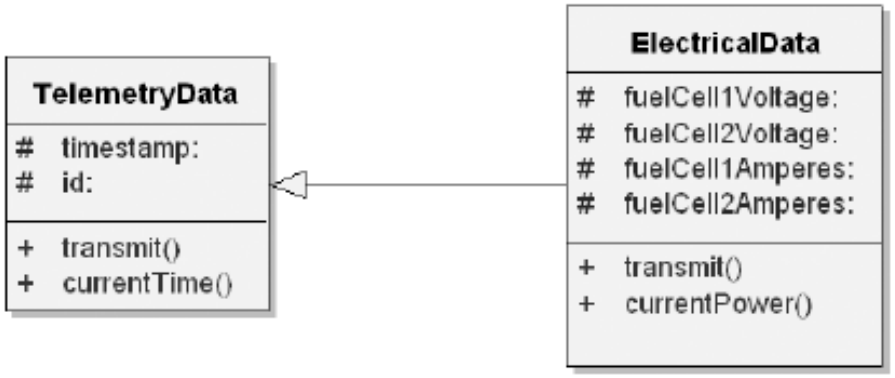
- A one-to-one relationship denotes a very narrow association.
- A one-to-one relationship between the class Sale and the class CreditCardTransaction
- Each sale has exactly one corresponding credit card transaction, and each such transaction corresponds to one sale.
- Many-to-many relationships are also common.
- For example, each instance of the class Customer might initiate a transaction with several instances of the class Sales Person, and each such salesperson might interact with many different customers.

Inheritance

- Inheritance, perhaps the most semantically interesting of these concrete relationships, exists to express generalization/specialization relationships.
- An alternate approach to inheritance involves a language mechanism called *delegation, in which objects delegate their behavior to related objects.*

A subclass may inherit the structure and behavior of its superclass.

- A slightly better way to capture our decisions would be to declare one class for each kind of telemetry data.
- In this manner, we could hide the representation of each class and associate its behavior with its data.
- Still, this approach does not address the problem of redundancy.

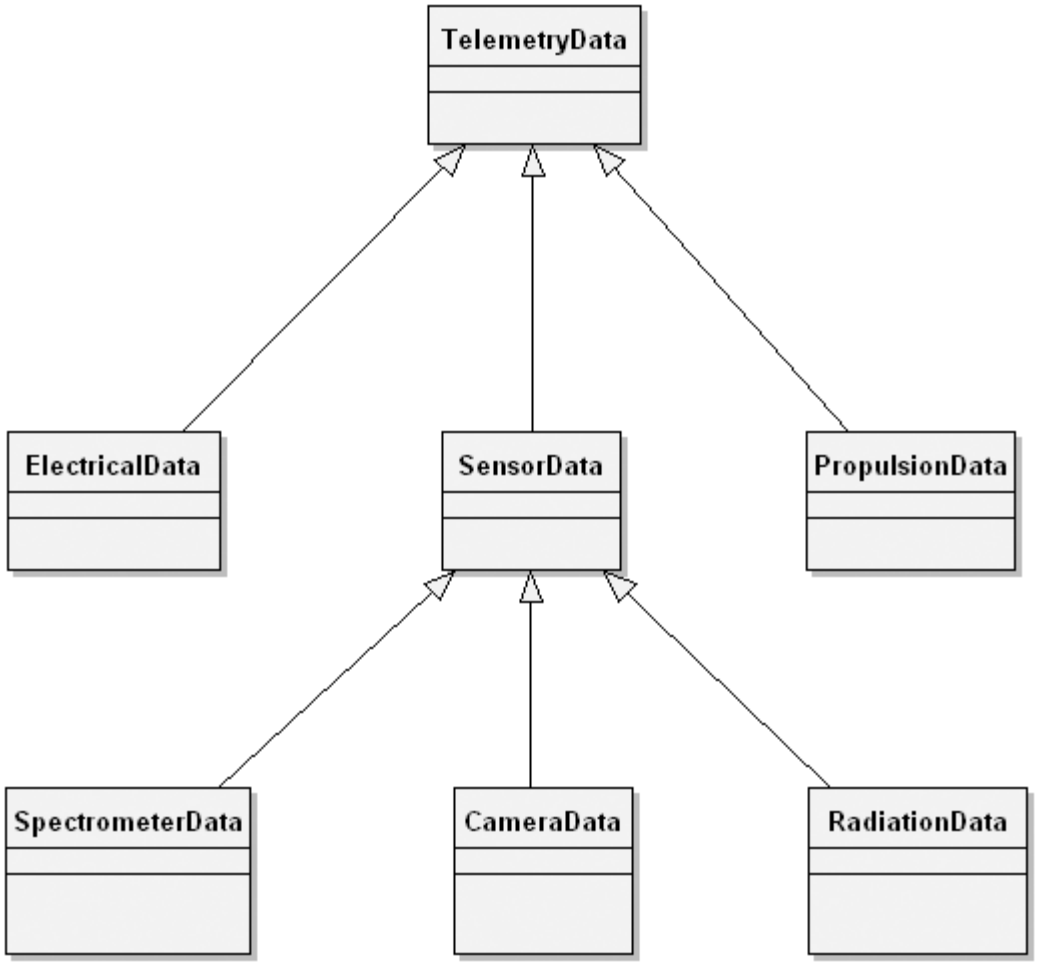


ElectricalData Inherits from the Superclass TelemetryData

Single Inheritance

- Simply stated, inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.
- Inheritance therefore defines an “is a” hierarchy among classes, in which a subclass inherits from one or more superclasses.
- This is in fact the litmus test for inheritance.
- Given classes A and B, if A is not a kind of B, then A should not be a subclass of B.

- A subclass typically augments or restricts the existing structure and behavior of its superclasses.
- A subclass that augments its superclasses is said to use inheritance for extension.
- In contrast, a subclass that constrains the behavior of its superclasses is said to use inheritance for restriction.
- In practice, it is not always so clear whether or not a subclass augments or restricts its superclass; in fact, it is common for a subclass to do both.



Single Inheritance

Polymorphism

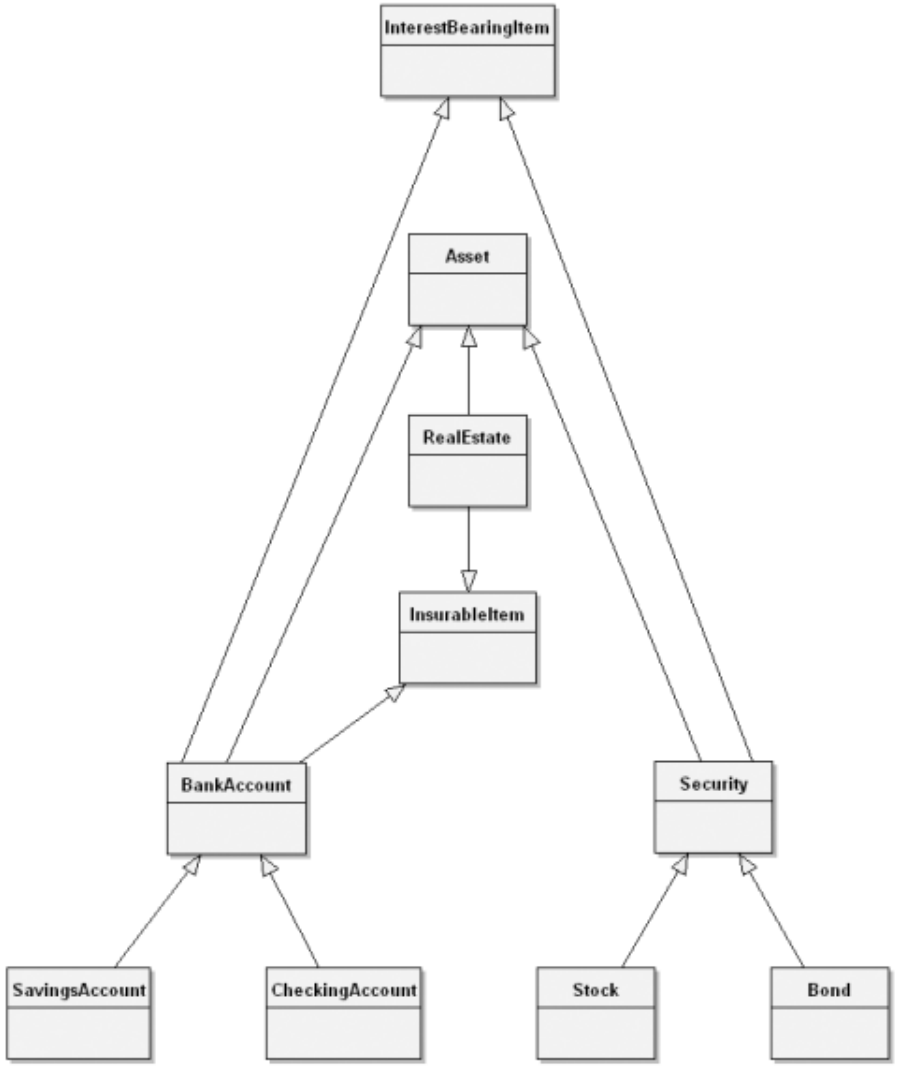
- Polymorphism is a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common superclass.
- Any object denoted by this name is thus able to respond to some common set of operations in different ways.
- With polymorphism, an operation can be implemented differently by the classes in the hierarchy.
- In this manner, a subclass can extend the capabilities of its superclass or override the parent's operation,

Multiple Inheritance

- With single inheritance, each subclass has exactly one superclass.
- However, as Vlissides and Linton point out, although single inheritance is very useful, “it often forces the programmer to derive from one of two equally attractive classes. This limits the applicability of predefined classes, often making it necessary to duplicate code. For example, there is no way to derive a graphic that is both a circle and a picture; one must derive from one or the other and reimplement the functionality of the class that was excluded”

- Designing a suitable class structure involving inheritance, and especially involving multiple inheritance, is a difficult task.
- This is often an incremental and iterative process.
- Two problems present themselves when we have multiple inheritance:
  - How do we deal with name collisions from different superclasses?
  - How do we handle repeated inheritance?

- Name collisions are possible when two or more different superclasses use the same name for some element of their interfaces, such as instance variables and methods.
- It mean to inherit two operations with the same name?
- This in fact is the key difficulty with multiple inheritance: Clashes may introduce ambiguity in the behavior of the multiply inherited subclass.



Multiple Inheritance

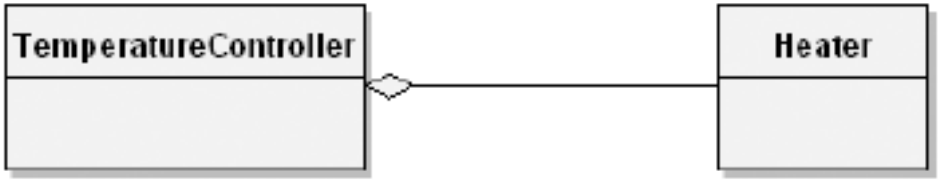
- There are three basic approaches to resolving this kind of clash.
- First, the language semantics might regard such a clash as illegal and reject the compilation of the class.
- Second, the language semantics might regard the same name introduced by different classes as referring to the same attribute.
- Third, the language semantics might permit the clash but require that all references to the name fully qualify the source of its declaration.

- There are various approaches to dealing with the problem of repeated inheritance.
- First, we can treat occurrences of repeated inheritance as illegal.
- Second, we can permit duplication of superclasses but require the use of fully qualified names to refer to members of a specific copy.
- Third, we can treat multiple references to the same class as denoting the same class.
- Different languages handle this approach differently.

- The existence of multiple inheritance gives rise to a style of classes called *mixins.*
- Mixins derive from the programming culture surrounding the language Flavors:
- One would combine (mix in) little classes to build classes with more sophisticated behavior.
- “A mixin is syntactically identical to a regular class, but its intent is different. The purpose of such a class is solely to . . . [add] functions to other flavors [classes]—one never creates an instance of a mixin”

Aggregation

- A class that is constructed primarily by inheriting from mixins and does not add its own structure or behavior is called an *aggregate class.*
- We also need aggregation relationships, which provide the whole/part relationships manifested in the class's instances.
- Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes.



- The class TemperatureController denotes the whole, and the class Heater is one of its parts.
- This corresponds exactly to the aggregation relationship among the instances of these classes

Dependencies

- Aside from inheritance, aggregation, and association, there is another group of relationships called *dependencies.*
- *A dependency indicates that an element on one* end of the relationship, in some manner, depends on the element on the other end of the relationship.
- This alerts the designer that if one of these elements changes, there could be an impact to the other. There are many different kinds of dependency relationships

The Interplay of Classes and Objects

- Classes and objects are separate yet intimately related concepts.
- Specifically, every object is the instance of some class, and every class has zero or more instances.
- For practically all applications, classes are static; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program.
- Similarly, the class of most objects is static, meaning that once an object is created, its class is fixed.
- In sharp contrast, however, objects are typically created and destroyed at a furious rate during the lifetime of an application.

Relationships between Classes and Objects

- For example, consider the classes and objects in the implementation of an air traffic control system.
- Some of the more important abstractions include planes, flight plans, runways, and air spaces.
- By their very definition, the meanings of these classes and objects are relatively static.
- They must be static, for otherwise one could not build an application that embodied knowledge of such commonsense facts as that planes can take off, fly, and then land, and that two planes should not occupy the same space at the same time.
- Conversely, the instances of these classes are dynamic.
- At a fairly slow rate, new runways are built, and old ones are deactivated.

The Role of Classes and Objects in Analysis and Design

- During analysis and the early stages of design, the developer has two primary tasks:
    1. Identify the classes that form the vocabulary of the problem domain
    2. Invent the structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem
- Collectively, we call such classes and objects the *key abstractions of the problem,* and we call these cooperative structures the *mechanisms of the implementation.*

On Building Quality Classes and Objects

- Ingalls suggests that “a system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework” [51].
- With object-oriented development, these parts are the classes and objects that make up the key abstractions of the system, and the framework is provided by its mechanisms.

Measuring the Quality of an Abstraction

- How can one know if a given class or object is well designed?
- We suggest five meaningful metrics:
  - Coupling
  - Cohesion
  - Sufficiency
  - Completeness
  - Primitiveness

measure quality of abstraction

Coupling

- Coupling is a notion borrowed from structured design, but with a liberal interpretation it also applies to object -oriented design.
- Stevens, Myers, and Constantine define coupling as "the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules"

Strong couping: 1. highly inter-related 2. harder to understand
Weak coupling : 1. complexity can be reduced by designing systems

cohesion

- The idea of cohesion also comes from structured design.
- Simply stated, cohesion measures the degree of connectivity among the elements of a single module (and for object-oriented design, a single class or object).
- The least desirable form of cohesion is coincidental cohesion, in which entirely unrelated abstractions are thrown into the same class or module.
- For example, consider a class comprising the abstractions of dogs and spacecraft, whose behaviors are quite unrelated.
- The most desirable form of cohesion is functional cohesion, in which the elements of a class or module all work together to provide some well-bounded behavior.

sufficient

minimum amt. of interface involved, violation detected early

- By sufficient, we mean that the class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction.
- To do otherwise renders the component useless.
- For example, if we are designing the class Set, it is wise to include an operation that removes an item from the set, but our wisdom is futile if we neglect an operation that adds an item.
- In practice, violations of this characteristic are detected very early; such shortcomings rise up almost every time we build a client that must use this abstraction.

complete

covers all aspects of abstraction

- By complete, we mean that the interface of the class or module captures all of the meaningful characteristics of the abstraction.
- Whereas sufficiency implies a minimal interface, a complete interface is one that covers all aspects of the abstraction.
- A complete class or module is thus one whose interface is general enough to be commonly usable to any client.
- Completeness is a subjective matter, and it can be overdone.
- Providing all meaningful operations for a particular abstraction overwhelms the user and is generally unnecessary since many high-level operations can be composed from low-level ones.

Primitive

- Primitive operations are those that can be efficiently implemented only if given access to the underlying representation of the abstraction
- An operation is indisputably primitive if we can implement it only through access to the underlying representation.
- An operation that could be implemented on top of existing primitive operations, but at the cost of significantly more computational resources, is also a candidate for inclusion as a primitive operation.

Choosing Operations

- Crafting the interface of a class or module is plain hard work.
- Typically, we make a first attempt at the design of a class, and then, as we and others create clients, we find it necessary to augment, modify, and further refine this interface.
- Eventually, we may discover patterns of operations or patterns of abstractions that lead us to invent new classes or to reorganize the relationships among existing ones.

Functional Semantics

- Within a given class, it is our style to keep all operations primitive, so that each exhibits a small, well-defined behavior.
- We call such methods *fine-grained. We* also tend to separate methods that do not communicate with one another.
- Halbert and O'Brien offer the following criteria to be considered when making such a decision.

1. classes should be well-defined
2. reusability, complexity and implementation knowledge

- Reusability:
  - Would this behavior be more useful in more than one context?
- Complexity:
  - How difficult is it to implement the behavior?
- Applicability:
  - How relevant is the behavior to the type in which it might be placed?
- Implementation knowledge:
  - Does the behavior's implementation depend on the internal details of a type?

Time and Space Semantics

Time reqd. to complete a structure **Space reqd. to complete a structure**

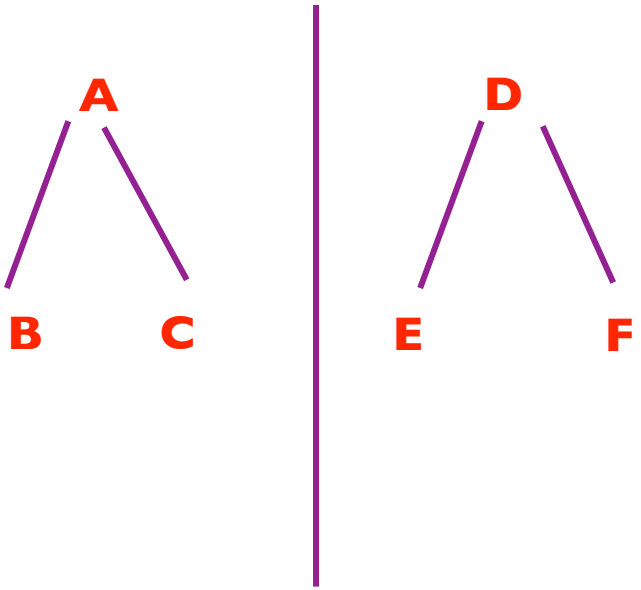
- Once we have established the existence of a particular operation and defined its functional semantics, we must decide on its time and space semantics.
- This means that we must specify our decisions about the amount of time it takes to complete an operation and the amount of storage it needs
- Such decisions are often expressed in terms of best, average, and worst cases, with the worst case specifying an upper limit on what is acceptable.

Choosing Relationships

- Choosing the relationships among classes and among objects is linked to the selection of operations.
- If we decide that object X sends message M to object Y, then either directly or indirectly, Y must be accessible to X; otherwise, we could not name the operation M in the implementation of X.
- By accessible, we mean the ability of one abstraction to see another and reference resources in its outside view.

** *The Law of Demeter*

- Law of Demeter, which states that “the methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only”



methods involved in A does not depend on those of D/E/F

methods of B are highly related to those of A

Choosing Implementations

- Only after we stabilize the outside view of a given class or object do we turn to its inside view.
- This perspective involves two different decisions: a choice of representation for a class or object and the placement of the class or object in a module.

- ***Representation***
  - The representation of a class or object should almost always be one of the encapsulated secrets of the abstraction.
  - This makes it possible to change the representation (e.g., to alter the time and space semantics) without violating any of the functional assumptions that clients may have made.
  - One of the more difficult trade-offs when selecting the implementation of a class is between computing the value of an object's state versus storing it as a field.

- ***Packaging***
  - Like the design of classes and objects, module design is not to be taken lightly.
  - As Parnas, Clements, and Weiss note with regard to information hiding, "Applying this principle is not always easy. It attempts to minimize the expected cost of software over its period of use and requires that the designer estimate the likelihood of changes. Such estimates are based on past experience and usually require knowledge of the application area as well as an understanding of hardware and software technology"