

Syllabus

Elective Course-II-17PCSE07 OBJECT ORIENTED ANALYSIS AND DESIGN

Credits: 4

Course Objectives:

- Describe Object Oriented Analysis and Design concepts to solve many real life problems and to develop Software.
- Helps to prepare Object Oriented Analysis and Design documents for a given problem using Unified Modeling Language

UNIT – I

Introduction: Role of Analysis and Design in Software Development – Meaning of Object Orientation - Overview of Various OOAD Methodologies - Goals of UML. **Use case Modeling:** Actors and Use Cases - Use Case Relationships - Writing Use Cases formally - Choosing the System Boundary - Finding Actors - Finding Use Cases - Use of Use Cases for Validation and Verification - Use Case Realization.

UNIT - II

Concept: The Object Model - The Evolution of the Object Model - Foundations of the Object Model - Elements of the Object Model - Applying the Object Model. **Classes and Object:** The Nature of an Object - Relationships among Objects - The Nature of a Class - Relationships among Classes - The Interplay of Classes and Objects - On Building Quality Classes and Objects **Classification:** The importance of proper classification - Identifying classes and objects - Key abstractions and Mechanisms.

UNIT - III

Notations: The Unified Modeling Language - Package Diagrams - Component Diagrams - Deployment Diagrams - Use Case Diagrams - Activity Diagrams.

UNIT – IV

Class Diagrams: Sequence Diagrams - Interaction Overview Diagrams - Composite Structure Diagrams - State Machine Diagrams - Timing Diagrams - Object Diagrams - Communication Diagrams.

UNIT – V

Applications: System Architecture: Satellite-Based Navigation - Control System: Traffic Management - Web Application: Vacation Tracking System - Data Acquisition: Weather Monitoring Station.

TEXT BOOKS

1. Mahesh P. Matha, "Object – Oriented Analysis and Design Using UML" , PHI Learning Private Limited, New Delhi, 2008.
2. Grady Booch Robert A. Maksimchuk Michael W. Engle Bobbi J. Young, Ph.D. Jim Conallen Kelli A. Houston "Object-Oriented Analysis and Design with Applications" Third Edition, Pearson Education, Inc., April 2007.

REFERENCE BOOKS

1. Martin Fowler, Kendall Scott, "UML Distilled, A Brief Guide to the Standard Object Modeling Languages", Second Edition, Pearson Education, 2000.
2. James Rumbaugh et al, " Object - Oriented Modeling and Design With UML" second Edition, Pearson Education, 2007.

UNIT - I

INTRODUCTION & MODELING




Object-oriented analysis and design

- Object-oriented analysis and design (OOAD) is a popular technical approach for
 - analyzing,
 - designing an application, system, or business
 - by applying the object oriented paradigm and
 - visual modeling throughout the development life cycles for better communication and product quality.
- Object-oriented programming (OOP) is a method
 - based on the concept of “objects”,
 - which are data structures that contain data,
 - in the form of fields,
 - often known as attributes;
 - and code, in the form of procedures,
 - often known as methods.

some of OOD concepts that seem relevant to the UML:

- **Class and object,**
- **Message, Operation, Method,**
- **Encapsulation,**
- **Abstraction,**
- **Inheritance,**
- **Polymorphism.**

- 
- The **three major methodologies** and their modeling notations developed by **Rumbaugh *et al.***, **Booch** and **Jacobson** which are the origins of the Unified Modeling Language.
 - Each method has its strengths.
 - **Rumbaugh** method is well-suited for describing the object model or the static structure of the system.
 - The **Jacobson *et al.*** method is good for producing user – driven analysis models.
 - The **Booch** method produces detailed object-oriented design models.

Booch Methodology

- It is a widely used object oriented method that helps us design our system using the object paradigm.
- It covers the analysis and design phases of an object oriented system.
- We start with class & object diagrams in analysis phase and refine these diagrams in various steps.
- The Booch method consists of the following *diagrams* :
 - *Class diagrams,*
 - *Object diagrams,*
 - *State Transition diagrams,*
 - *Module diagrams*
 - *Process diagrams,*
 - *Interaction programs.*



Jacobson *et al.* Methodology

- Use Cases.
- Object Oriented Software Engineering.
- Object Oriented Business Engineering.



Use Cases

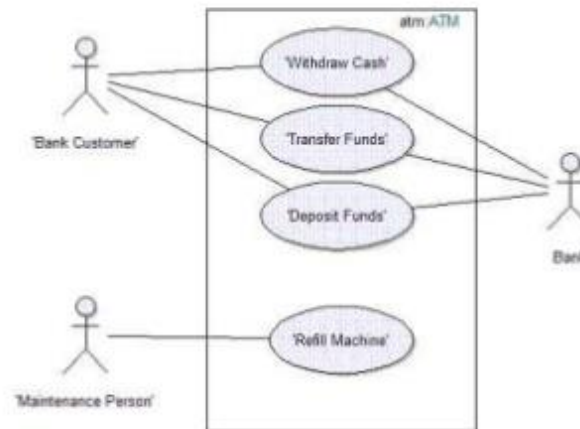
- Understanding system requirements
- Interaction between Users and Systems
- The use case description must contain
 - How and when the use case begins and ends.
 - The Interaction between the use case and its actors, including when the interaction occurs and what is exchanged.
 - How and when the use case will need data stored in the system.
 - Exception to the flow of events
 - How and when concepts of the problem domain are handled.

OOSE

- Object Oriented Software Engineering.
- Objectory (Object Factory for S/w Development) is built around several different models:
 - **Use case model.** The use-case model defines the outside (actors) and inside (use case) of the systems behavior.
 - **Domain object model.** The objects of the “real” world are mapped into the domain object model.
 - **Analysis object model.** The analysis object model presents how the source code (implementation) should be carried out and written.
 - **Implementation model.** The implementation model represents the implementation of the system.
 - **Test model.** The test model constitutes the test plans, specifications, and reports.

Use Case Diagrams

- A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved.



Actor



An **actor instance is someone** or something outside the system that interacts with the system.

- An actor is anything that exchanges data with the system.
- An actor can be a user, external hardware, or another system.



How to Find Actors

- Actors:
 - Supply/use/remove information
 - Use the functionality.
 - Will be interested in any requirement.
 - Will support/maintain the system.
 - The system's external resources.
 - The other systems will need to interact with this one.

Terminologies

- **System boundary**: rectangle diagram representing the boundary between the actors and the system.
- **Use Case Diagram(core relationship)**
Association: communication between an actor and a use case;
Represented by a solid line.



- Generalization: relationship between one general use case and a special use case (used for defining special alternatives)
- Represented by a line with a triangular arrow head toward the parent use case.



- **Include**: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use “include” in stead of copying the description of that behavior.

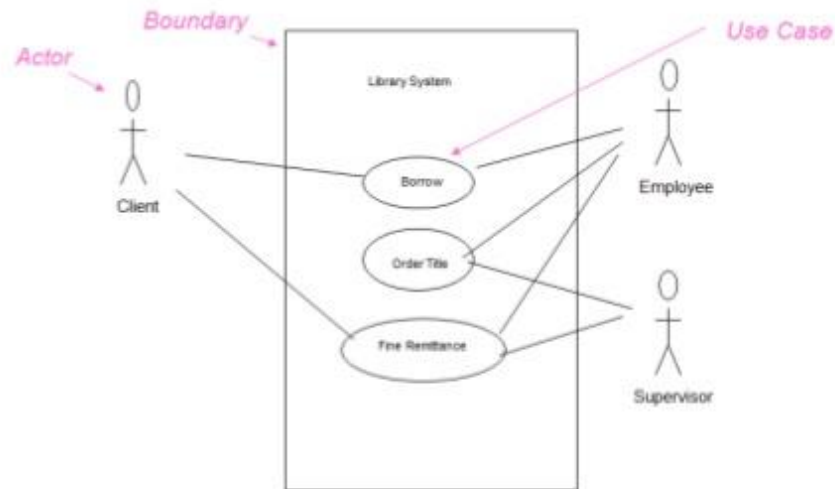
<<include>>
----->

- **Extend**: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares “extension points”.

<<extend>>
----->

Example: Library management System

- A generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system



UNIT II

CONCEPT & OBJECT

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 [\[2\]](#). (By no means is this an exhaustive list of all programming languages.)
-
- 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
-

Object-orientation boom (1980–1990, but few languages survive) Smalltalk 80
object-oriented language

Pure

- 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 .NET Framework
- Visual Basic .NET Visual Basic for the Microsoft .NET Framework

The Topology of First- and Early Second-Generation PLs

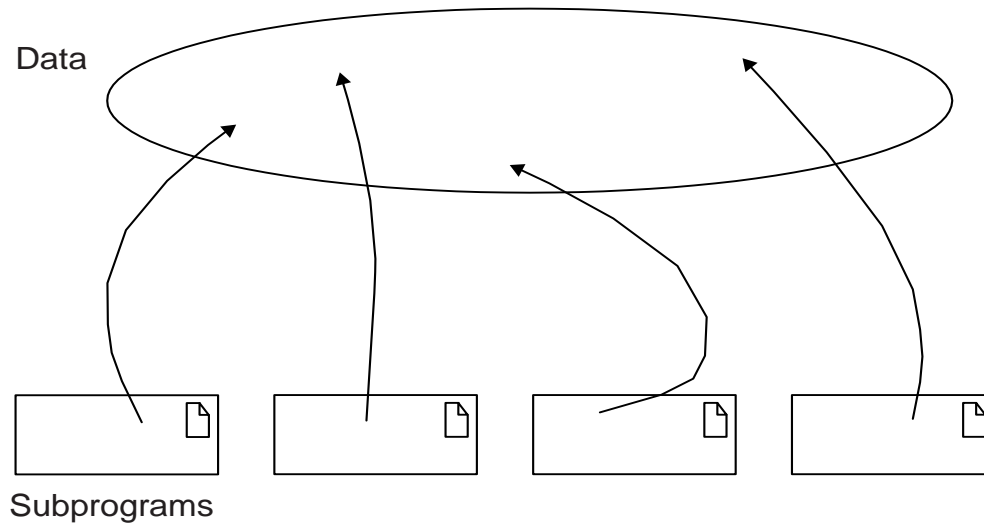


Figure 2-1 The Topology of First- and Early Second-Generation Programming Languages

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

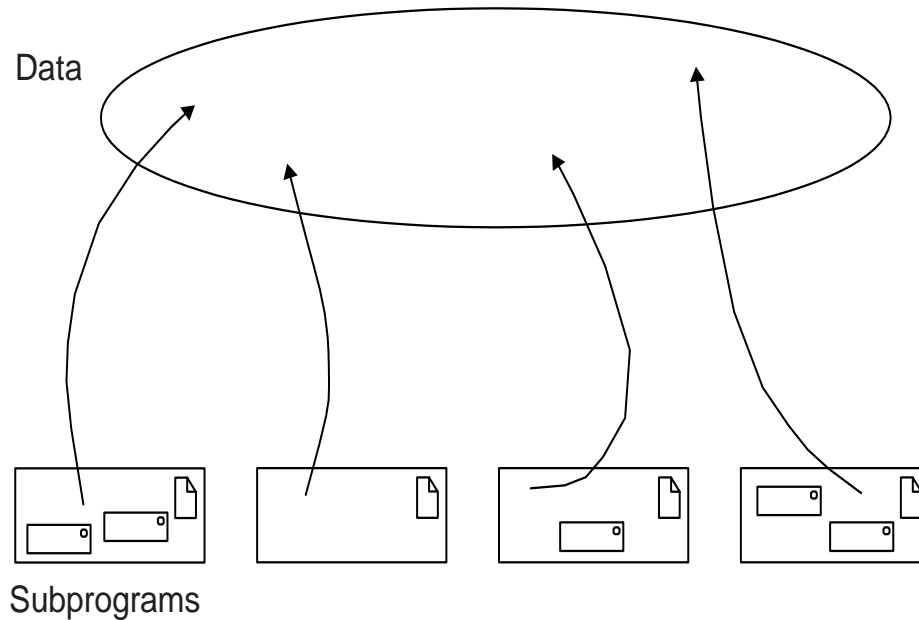


Figure 2-2 The Topology of Late Second- and Early Third-Generation Programming Languages

The Topology of Late Third-Generation PLs

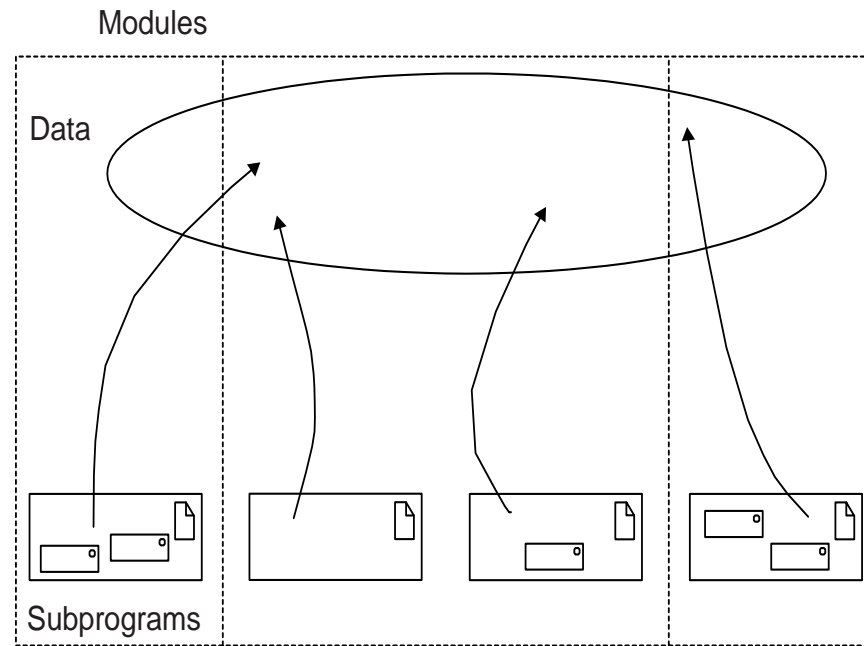


Figure 2-3 The Topology of Late Third-Generation Programming Languages

The Topology of Object-Based and Object-Oriented PLs

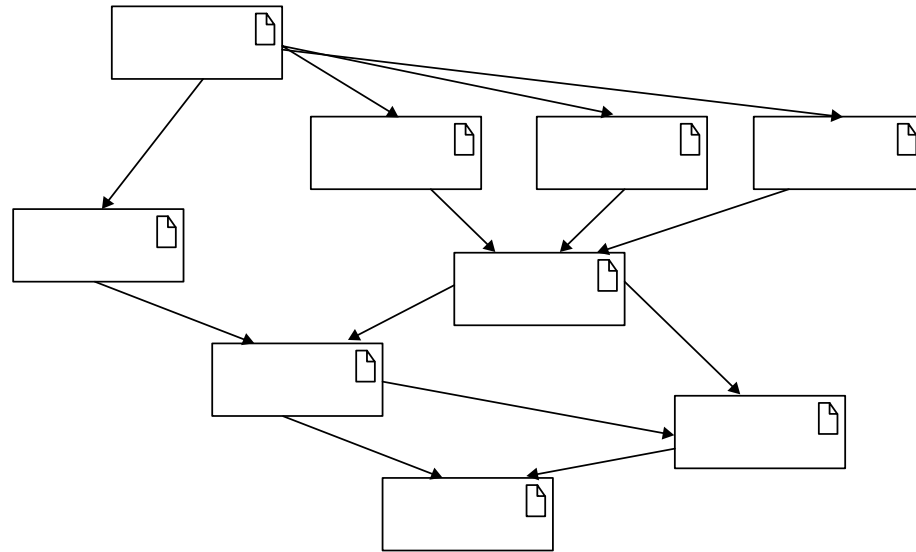


Figure 2-4 The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

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 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 Analysis

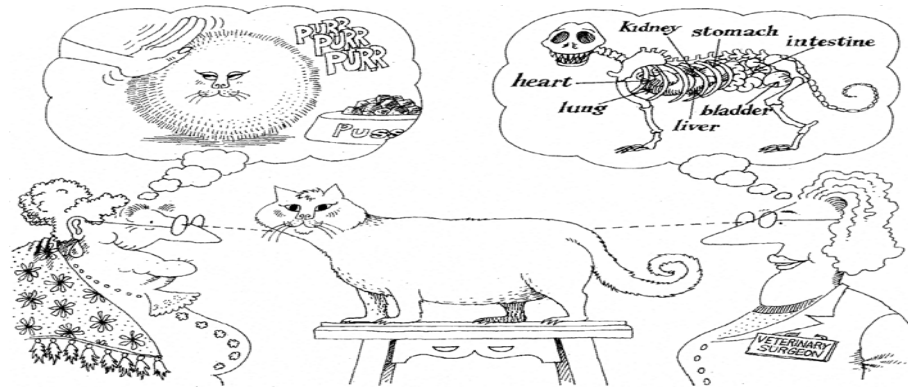
Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

Elements of the Object Model

- 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

The Meaning of Abstraction

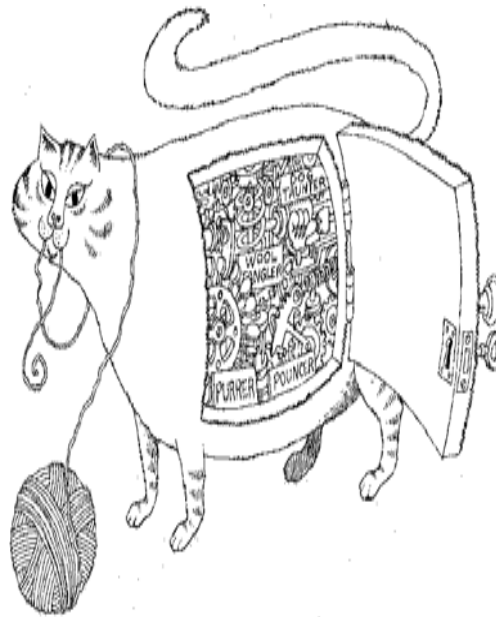
- An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.



Kinds of abstractions include the following:

- Entity abstraction - An object that represents a useful model of a problem domain or solution domain entity
- Action abstraction - An object that provides a generalized set of operations, all of which perform the same kind of function
- 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
- Coincidental abstraction - An object that packages a set of operations that have no relation to each other

The Meaning of Encapsulation



Encapsulation hides the details of the implementation of an object.

- 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.

The Meaning of Modularity



Modularity packages abstractions into discrete units.

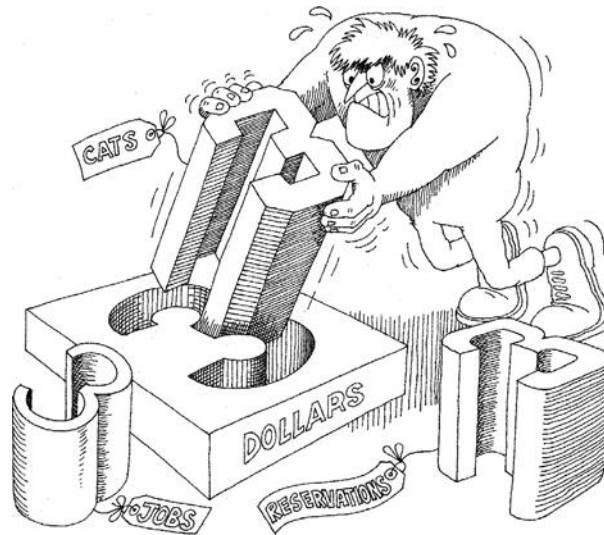
The Meaning of Hierarchy

Hierarchy is a ranking or ordering of abstractions.



The Meaning of Typing

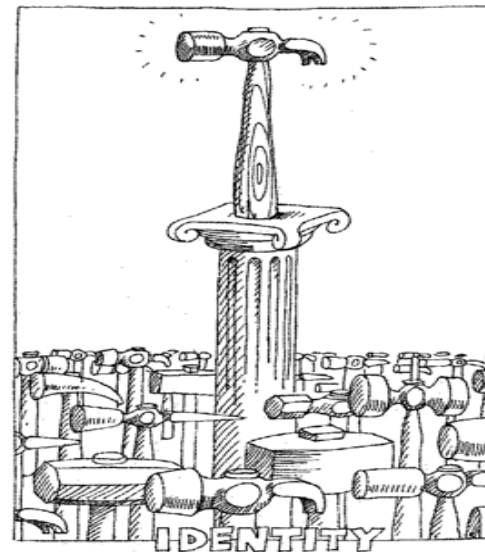
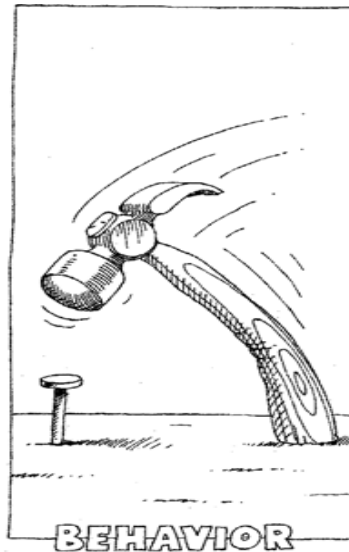
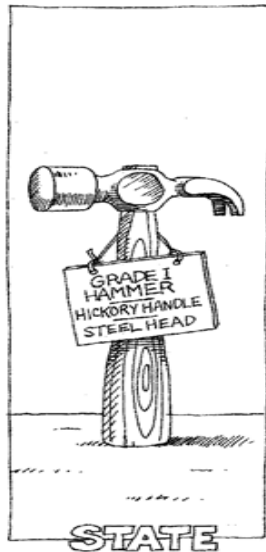
A type is a precise characterization of structural or behavioral properties which a collection of entities all share



Classes and Objects

What Is and What Isn't an Object

- A tangible and/or visible thing
- Something that may be comprehended intellectually
- Something toward which thought or action is directed



- 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.

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.

Behavior is how an object acts and reacts, in terms of its state changes and message passing.

The state of an object represents the cumulative results of its behavior.

Operations

- 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

Roles and Responsibilities

“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”

Relationships among Objects

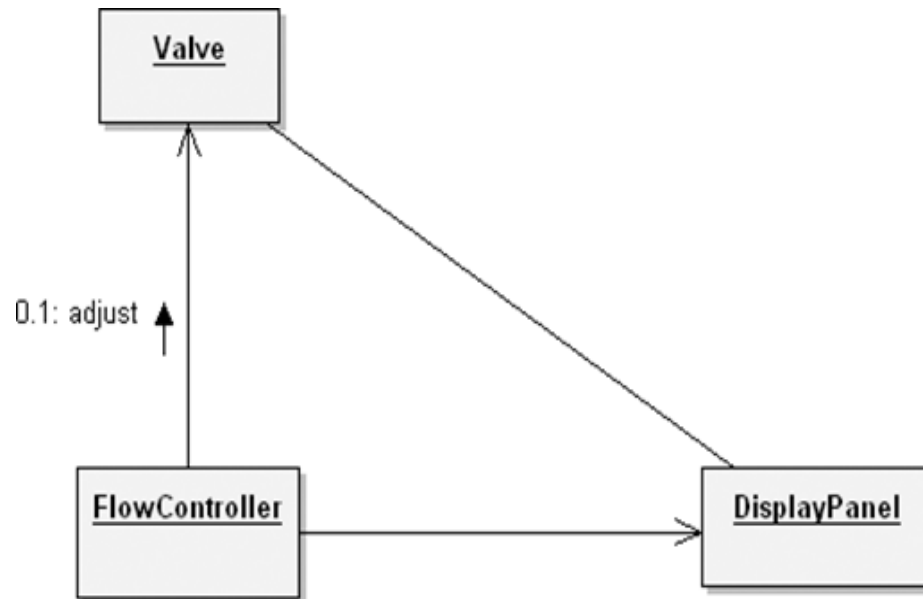
Two kinds of object relationships

- Links
- Aggregation

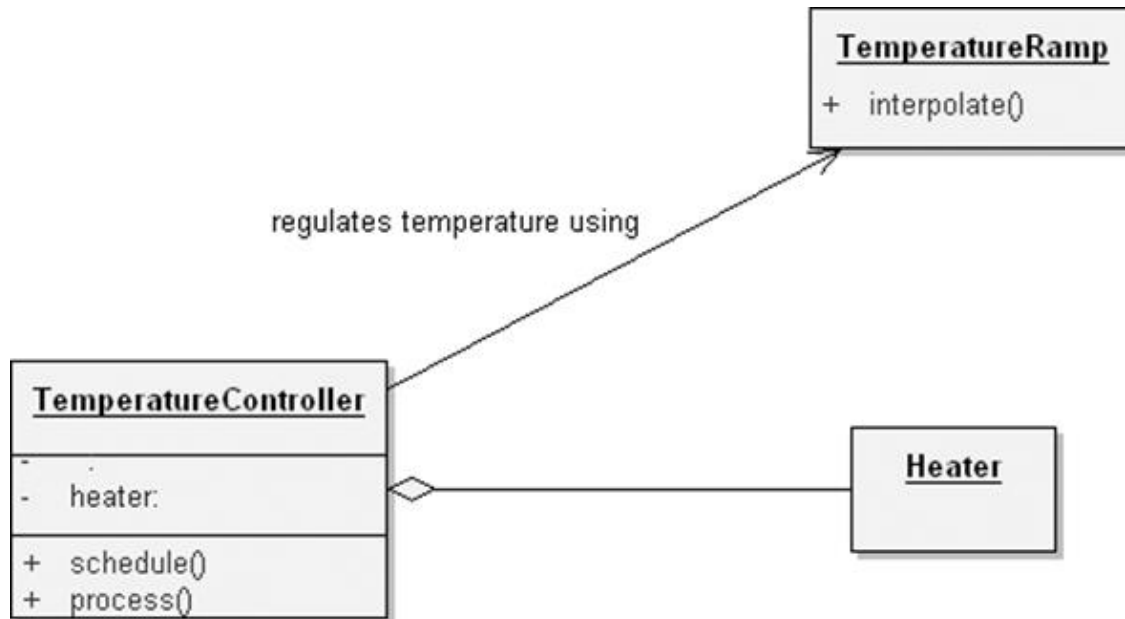
An object may play one of three roles.

- Controller: This object can operate on other objects but is not operated on by other objects. In some contexts, the terms *active object* and *controller* are interchangeable.
- Server: This object doesn't operate on other objects; it is only operated on by other objects.
- Proxy: This object can both operate on other objects and be operated on by other objects. A proxy is usually created to represent a real-world object in the domain of the application.

Links



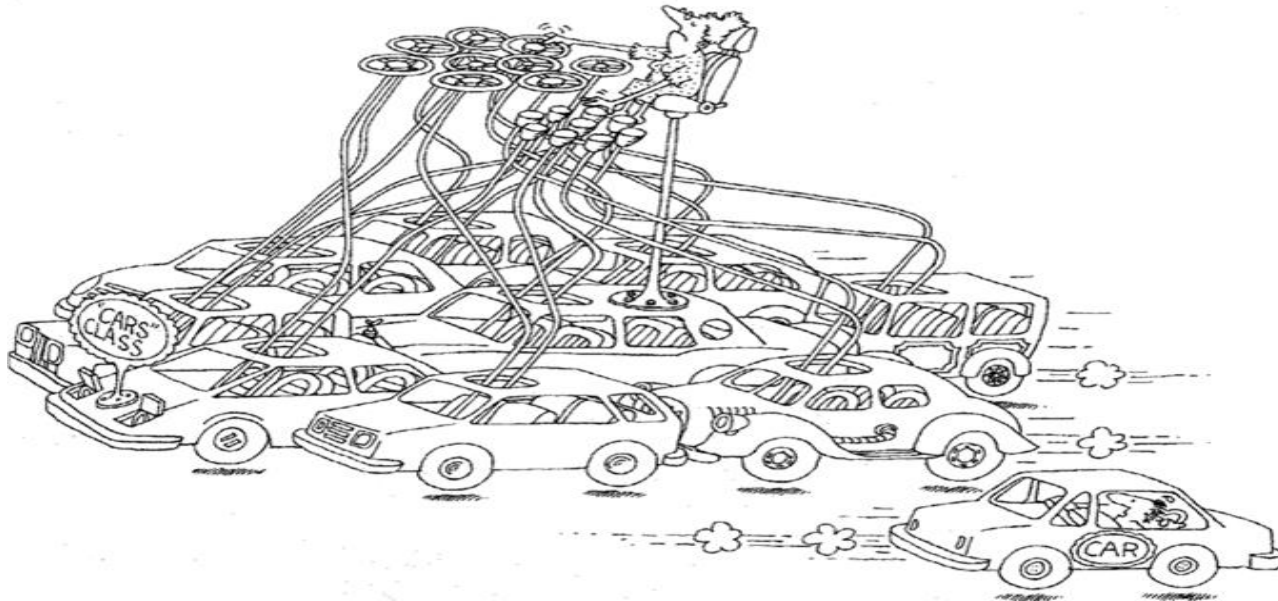
Aggregation



Class

What Is and What Isn't a Class

- A class is a set of objects that share a common structure, common behavior, and common semantics.



Interface and Implementation

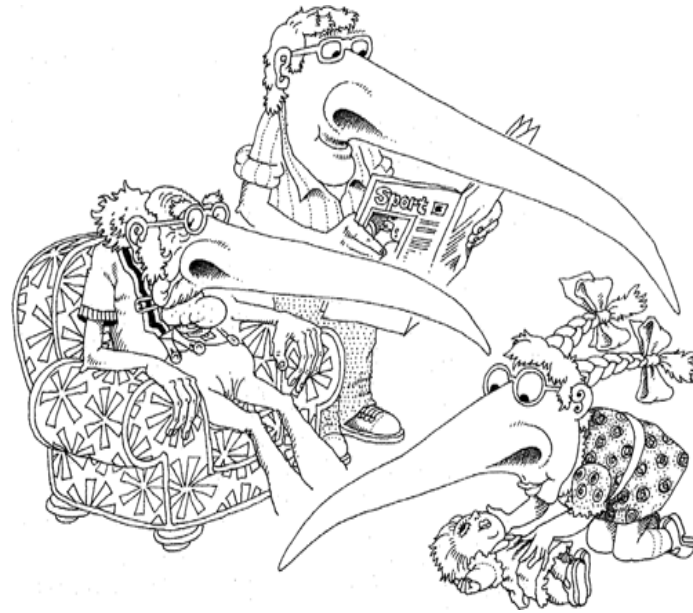
The interface of a class into four parts:

Public: a declaration that is accessible to all clients

- Protected: a declaration that is accessible only to the class itself and its sub-classes
- Private: a declaration that is accessible only to the class itself
- Package: a declaration that is accessible only by classes in the same package

Relationships among Classes

Inheritance



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

Single Inheritance

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

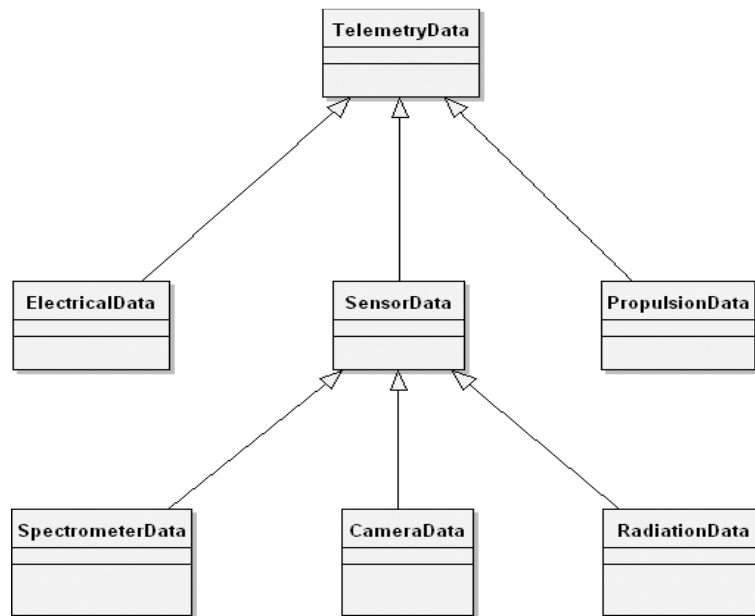


Figure 3–9 Single Inheritance

Multiple 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.

Aggregation

- Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes.

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.

Measuring the Quality of an Abstraction

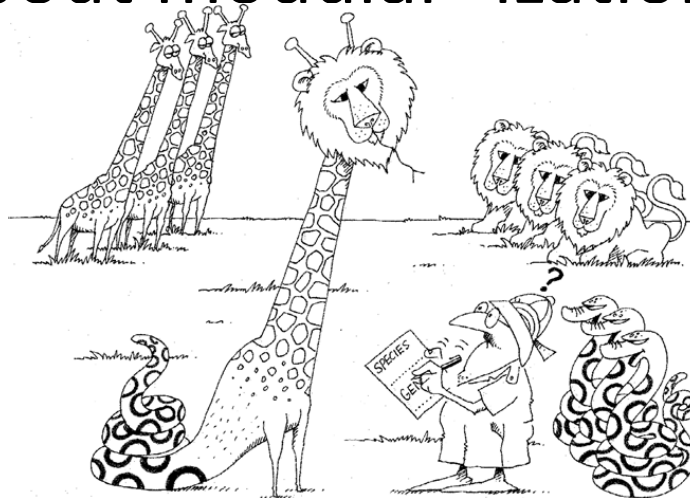
- Coupling
- Cohesion
- Sufficiency
- Completeness
- Primitiveness

Classification

Classification is the means whereby we order knowledge. In object-oriented design, recognizing the sameness among things allows us to expose the commonality within key abstractions and mechanisms and eventually leads us to smaller applications and simpler architectures.

The Importance of Proper Classification

- Classification helps us to identify generalization, specialization, and aggregation hierarchies among classes.
- Classification also guides us in making decisions about modular-ization.



Classification is the means whereby we order knowledge.

Identifying Classes and Objects

Classical and Modern Approaches

Historically, there have been only three general approaches to classification:

- Classical categorization
- Conceptual clustering
- Prototype theor

Object-Oriented Analysis

Classical Approaches

The objects derive primarily from the principles of classical categorization.

- Tangible things - Cars, telemetry data, pressure sensors
- Roles - Mother, teacher, politician
- Events - Landing, interrupt, request
- Interactions - Loan, meeting, intersection

Behavior Analysis

object-oriented analysis focuses on dynamic behavior as the primary source of classes and objects

Domain Analysis

To identify the classes and objects that are common to all applications within a given domain, such as patient record tracking, bond trading, compilers, or missile avionics systems.

Use Case Analysis

The practices of classical analysis, behavior analysis, and domain analysis all depend on a large measure of personal experience on the part of the analyst. For the majority of development projects, this is unacceptable because such a process is neither deterministic nor predictably successful.

CRC Cards

- CRC cards have proven to be a useful development tool that facilitates brainstorming and enhances communication among developers.
- A CRC card is nothing more than a 3x5 index card,³ on which the analyst writes—in pencil—the name of a class (at the top of the card), its responsibilities

Informal English Description

An English description of the problem (or a part of a problem) and then underlining the nouns and verbs. The nouns represent candidate objects, and the verbs represent candidate operations on them.

Structured Analysis

- Structured analysis as a front end to object-oriented design.
- A particular data flow diagram candidate objects may be derived from the following:
 - External entities
 - Data stores
 - Control stores
 - Control transformations
- Candidate classes derive from two sources:
 - Data flows
 - Control flows

Key Abstractions and Mechanisms

- A *key abstraction* is a class or object that forms part of the vocabulary of the problem domain.
- *Mechanism* to describe any structure whereby objects collaborate to provide some behavior that satisfies a requirement of the problem.

Identifying Key Abstractions

The identification of key abstractions is highly domain-specific. As Goldberg states, the “appropriate choice of objects depends, of course, on the purposes to which the application will be put and the granularity of information to be manipulated”

Refining Key Abstractions

The most common reorganizations of a class hierarchy are factoring the common part of two classes into a new class and splitting a class into two new ones

Naming Key Abstractions

Objects should be named with proper noun phrases, such as theSensor or just simply shape.

- Classes should be named with common noun phrases, such as Sensor or Shape.
- The names chosen should reflect the names used and recognized by the domain experts, whenever possible.
- Modifier operations should be named with active verb phrases, such as draw or moveLeft.
- Selector operations should imply a query or be named with verbs of the form “to be,” such as extentOf or isOpen.
- The use of underscores and styles of capitalization are largely matters of personal taste. No matter which cosmetic style you use, at least have your programs be self-consistent.

Identifying Mechanisms

- A mechanical linkage connects the accelerator directly to the fuel injectors.
- An electronic mechanism connects a pressure sensor below the accelerator to a computer that controls the fuel injectors (a drive-by-wire mechanism).
- No linkage exists. The gas tank is placed on the roof of the car, and gravity causes fuel to flow to the engine. Its rate of flow is regulated by a clip around the fuel line; pushing on the accelerator pedal eases tension on the clip, causing the fuel to flow faster (a low-cost mechanism).