Object Oriented Analysis and Design

Object oriented Paradigm

For many years, the term object oriented (OO) was used to denote a software development approach that used one of a number of object-oriented programming languages (e.g., Java, C++). Today, the OO paradigm encompasses a complete view of software engineering.

Basic Principles of Object Orientation

- **Abstraction**
  Reduces and factors out details so that one can focus on a few concepts at a time. It is the process of modelling real-world objects. When you create a class you create an abstraction of a real-world object. Abstraction is useful in managing complexity.
- **Encapsulation (Information Hiding)**
  The property of being a self-contained unit is called encapsulation. Encapsulation, accomplishes data hiding which is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally e.g. just as a refrigerator is used without knowing how the compressor works, one can use a well-designed object without knowing about its internal data members. This principle is useful in improving resilience.
- **Hierarchy**
  Ordering of abstractions into a tree-like structure.
- **Modularity**
  This involves the breaking up of something complex into manageable pieces. It is also useful in managing complexity

Basic Concepts of Object Orientation

- Objects
- Class
- Inheritance
- Polymorphism

Concept 1: Objects

Considering an example of a real world object: a chair is a member (the term **instance** is also used) of a much larger class of objects that we call furniture.
A set of generic **attributes** can be *associated* with every **object** in the **class furniture** e.g. all furniture has cost, dimensions, weight, location, and colour, among many possible attributes. These apply whether we are talking about a table or a chair, or a sofa.
Therefore because **chair is a member of furniture**, **chair** *inherits* **all attributes** defined for the **class furniture**.
Once the class has been defined, the attributes can be reused when new instances (objects) of the class are created.

Objects are thus the basic run-time entities in an object oriented system; they take up space in the memory and have associated addresses.
A programming problem is analyzed in terms of objects and the nature of communication between them and hence program objects should be chosen such that they match closely with the real-world objects.
When a program is executed, the objects interact by sending messages to one another.
For example, if “student” and “teacher” are two objects in a program, then the student object may send a message to the teacher object requesting for his marks scores in a test.

Each object contains data and code to manipulate the data. However, objects can interact without having know-details of one another’s data or code.
Objects in the real world can be characterized by two things, each real world object has:
1. Information (Data)
- has a unique identity
- has a description of its structure
- has a state representing its current condition

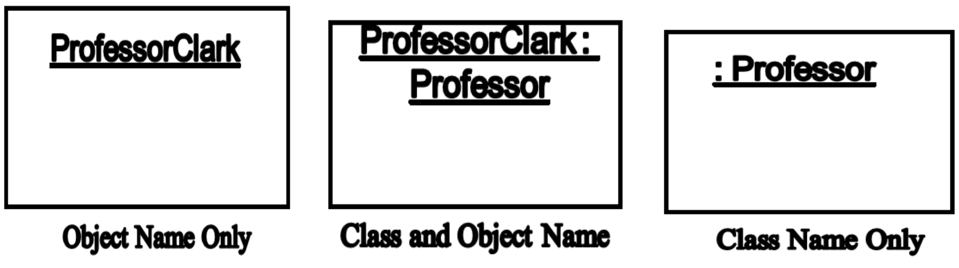
2. Behaviour
- what can an object do?
- what can be done to it?

E.g. a printer has data; serial number, model, speed, memory, status, and behaviour; print file, stop printing, empty queue.

Terminology
The **data** for an object are generally called the **Attributes** of the object while the different **behaviours** of an object are called the **Methods** (or **operations**) of the object.

Encapsulation: Only the instance that owns an item of data is allowed to modify or read it. This means that all of this information is packaged under one name and can be reused as one specification or program component.
e.g. The object **chair** (and all objects in general) **encapsulates:** data (the attribute values that define the chair), operations (the actions that are applied to change the attributes of chair), other objects (composite objects can be defined), constants (set values), and other related information.

An object is represented as rectangles with underlined names:



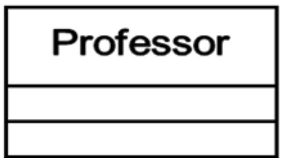
Concept 2: Classes

A class is a description of a set of objects that share the same attributes, operations i.e. an object is an instance of a class. The term class is simply a template for an object. A class is thus a collection of objects of similar type.
It describes what attributes and methods will exist for all instances of the class.
Since it is just a description, it doesn't really exist as such until you declare an instance of the class relationships, and semantics.

A class is an OO concept that **encapsulates** the **data abstractions(attributes)** and **procedural abstractions(methods)** required to describe the content and behavior of some real world entity.
The only way to reach the attributes (and operate on them) is to go through one of the methods that form the external wall.
Therefore, the class encapsulates data (inside the wall) and the processing that manipulates the data (the methods that make up the wall).
This achieves information hiding and reduces the impact of side effects associated with change.

Terminology

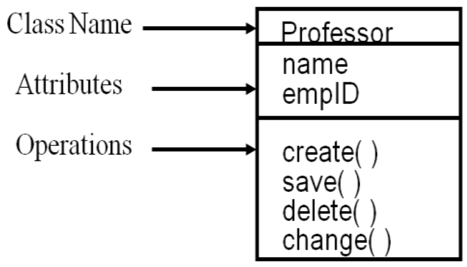
i. Class Name
It is a textual string. A class name must be unique within its enclosing package i.e. every class must have a name that distinguishes it from other classes.
A class may be drawn showing only its name using a compartmented rectangle



ii. Attributes
An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all.
An attribute represents some property of the thing you are modeling that is shared by all objects of that class
An attribute name may be text. In practice, an attribute name is a short noun or noun phrase that represents some property of its enclosing class.
Typically, you capitalize the first letter of every word in an attribute name except the first letter, as in ‘name’ or ‘loadBearing’

iii. Operations
An operation is the implementation of a service that can be requested from any object of the class to affect behaviour i.e. it is an abstraction of something you can do to an object and that is shared by all objects of that class.
A class may have any number of operations or no operations at all

An operation name may be text - a short verb or verb phrase that represents some behaviour of its enclosing class. It may be a question – where data will not me changed or a command where data will be changed.
Typically, you capitalize the first letter of every word in an operation name except the first letter, as in 'move' or 'isEmpty'.



Super Class
It is a class that contains the features common to two or more classes. It is similar to a superset, e.g. agency-staff.

Sub Class
Sub-Class is a class that contains at least the features of its super-class(es).
A class may be a sub-class and a super-class at the same time, e.g. management-staff.

Concept 3: Inheritance

Inheritance is one of the key differentiators between conventional (structured) and OO systems.
A subclass Y inherits all of the attributes and operations associated with its superclass, X. This means that all data structures and algorithms originally designed and implemented for X are immediately available for Y—no further work need be done

- Reuse is thus accomplished directly.
- Any change to the data or operations contained within a superclass is immediately inherited by all subclasses that have inherited from the superclass. Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system.

Inheritance is the process by which objects of one class acquire the properties of objects of another class.
It supports the principle of hierarchical classification. For example, the **grandchild** is a part of the class **parent** which is again a part of the class **grandparent.**

Concept 4: Polymorphism

Polymorphism means the ability to take on more than one form e.g. an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in operation. Polymorphism employs the principle of encapsulation.

Example 1:
Consider the operation of addition.

- For two numbers, the operation will generate a sum
- If the operands are strings, then the operation would produce a third string by concatenation

This means that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context.

Example 2:
Derived classes can redefine the implementation of a method.
Consider a class "Transport"

- One method contained in transport must be move(), because all transport must be able to move
- If we wanted to create a Boat and a Car class, we would certainly want to inherit from the transport class, as all Boats can move and all Cars can move. However both objects move in different ways

Example 3:

- management-staff and agency staff can apply for leave, but possibly in different ways.

Other OO Concepts…

a. Association
A large software system may have thousands of classes and objects and different classes must relate to each other, interact and collaborate to carry out processes.
Relationships between object classes are shown as lines linking objects and may be:

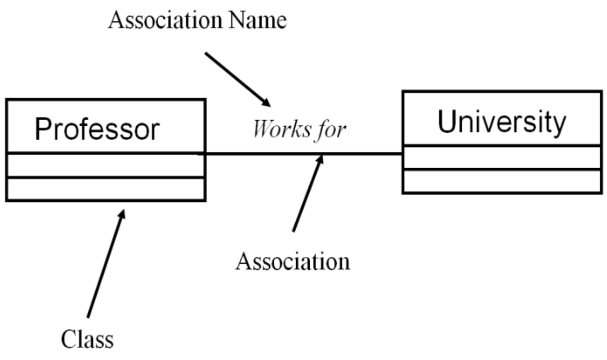
- between classes (relations)
- between objects (links)

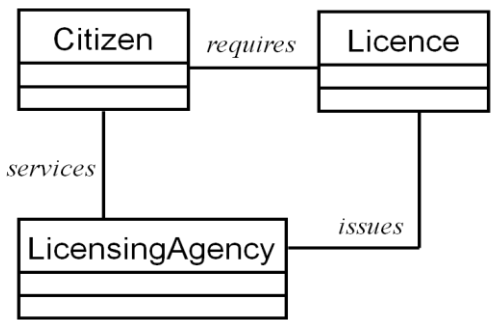
There are different kinds of relations between classes:

- Association
- Aggregation
- Composition
- Generalization
- Dependency

Associations are the simplest form of relation between classes and involves peer-to-peer relations whereby one object is aware of the existence of another object. It is implemented in objects as references.
An association is a relationship between two or more classifiers that involves connection among instances.
Examples of associations between classes A and B: *A is a physical or logical part of B; A is a kind of B; A is contained in B; A is a description of B; A is a member of B; A is an*

organization subunit of B; A uses or manages B; A communicates with B; A follows B; A is owned by B.
Example 1: Person works for the Company.
Example 2: A citizen requires a Licence and a Licensing Agency services the Citizen and issues the Licence.

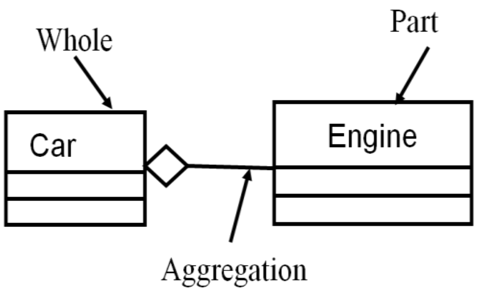


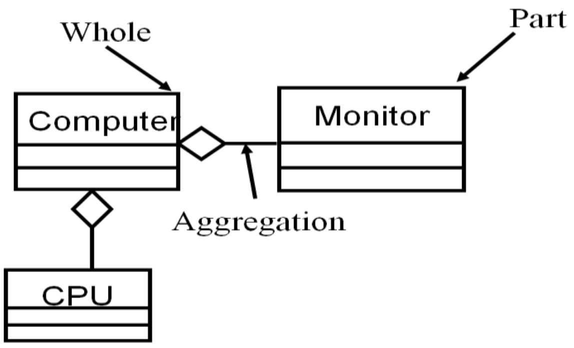


b. Aggregation

Aggregation is a restrictive form of "wholepart" or "part-of" association. Objects are assembled to create a more complex object. The assembly may be physical or logical.
It defines a single point of control for participating objects; the aggregate object coordinates its parts.
Aggregation is represented as a hollow diamond, pointing at the class which is used.
Example:

- A CPU is part of a computer.
- CPU, devices, monitor and keyboard are assembled to create a computer



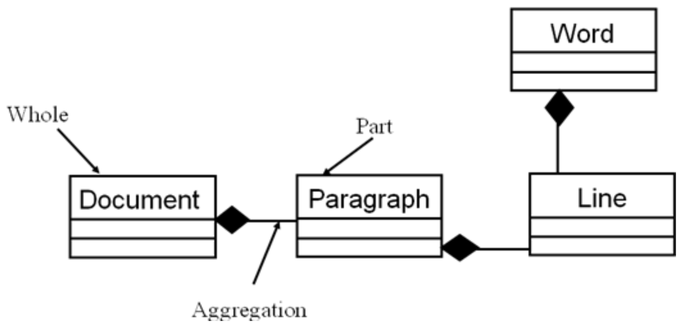


c. Composition

Composition is a stricter form of aggregation

- The lifespan of individual objects depend on the lifespan of the aggregate object
- Parts cannot exist on their own
- There is a create-delete dependency of the parts to the whole

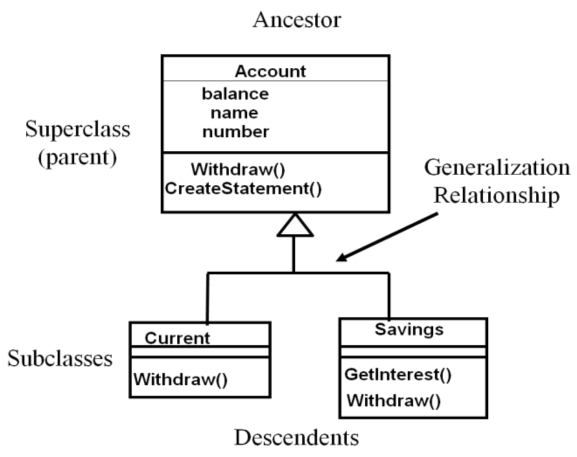
Composition is represented as a filled diamond, pointing at the class which is used.
For Example: A word cannot exist if it is not part of a line; If a paragraph is removed, all lines of the paragraph are removed, and all words belonging to that lines are removed.



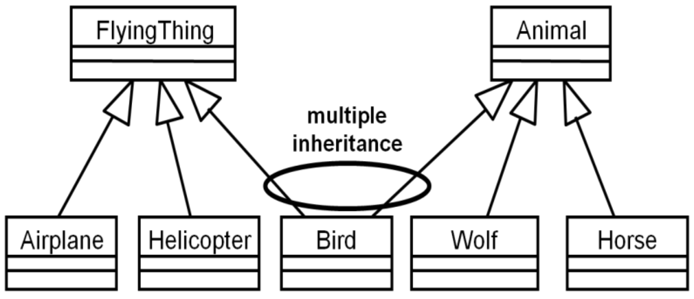
d. Generalization

A generalization is the relationship between a general class and one or more specialized classes.
In a generalization relationship, the specializations are known as **subclass** and the generalized class is known as the **superclass**
A generalization allows the inheritance of the attributes and operations of a superclass by its subclasses. It equivalent to a “kind-of” or “type-of” relationship
For Example: Common features are defined in class Account and Current and Savings (accounts) inherit them.

Single inheritance example:

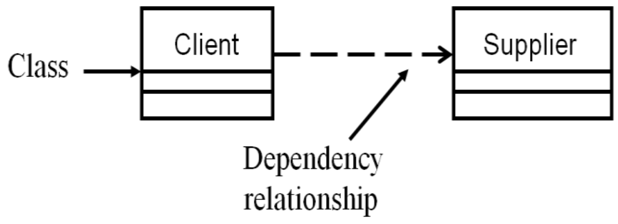


Multiple Inheritance Example:



e. Dependency

This is a relationship between two model elements where a change in one **<u>may</u>** cause a change in the other. It is a non-structural, "using" relationship.



Multiplicity

Multiplicity shows how many objects of one class can be associated with one object of another class. *For Example:* a citizen can apply for one or more licenses, and a license is required by one citizen.

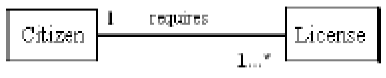
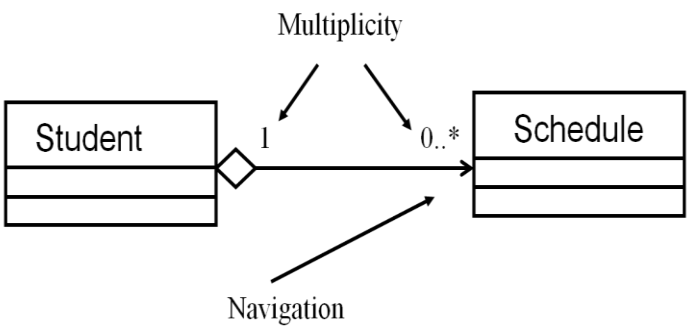


Table 1: Potential multiplicity values

Value	Description
0..1	Zero or one
1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
3	Three only
0..5	Zero to Five
5..15	Five to Fifteen

Navigation

Associations and aggregations are bi-directional by default, but it is often desirable to restrict navigation to one direction. If navigation is restricted, an arrowhead is added to indicate the direction of the navigation.

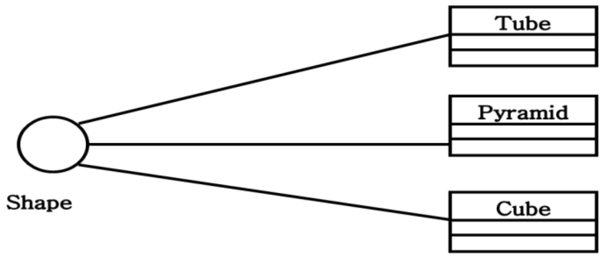


Interface

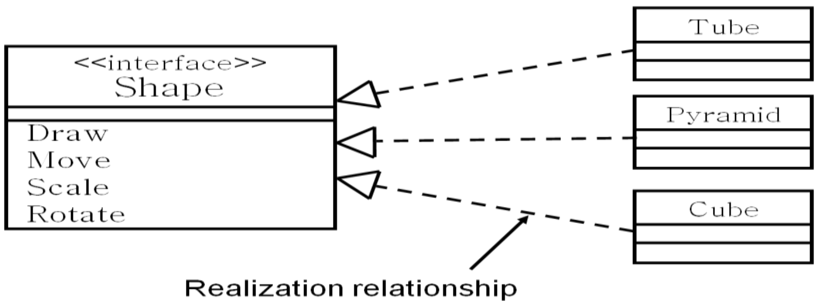
Interfaces formalize polymorphism.

Interface representations include:

a. Elided/Iconic Representation (“lollipop”/circle)



b. Canonical (Class/Stereotype) Representation



Modeling with UML

The Object Management Group (OMG) specification states that:
"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

The UML represents a collection of best engineering practices that have proven successful in the modelling of large and complex systems. It is a very important part of developing object oriented software and the software development process as it has become the de-facto standard for.

The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

A UML diagram is a partial graphic representation of a system's model. The UML model contains documentation that drives the model elements and diagrams. Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. Currently running UML 2.2, 14 UML diagrams represent two different views of a system model:

A. Static (or structural) view: This emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.

B. Dynamic (or behavioural) view: emphasizes the dynamic behaviour of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

Structure Diagrams

These emphasize the things that MUST be present in the system being modelled. It depicts elements that are irrespective of time.

1. Class Diagram models the system structure and contents using design elements such as classes, their attributes and also displays relationships such as containment, inheritance, associations and others.

2. Physical Diagrams

    - Component Diagram describes displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.

- Deployment Diagram displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.

Behavioural Diagrams

3. Use Case Diagram primarily describes the boundary and interaction between the system and users. It displays the relationship among actors and use cases.

4. Interaction Diagrams: describes how objects in the system will interact with each other to get work done.

   - Sequence Diagram displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).

   - Collaboration Diagram displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.

5. State Diagram displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.

6. Activity Diagram displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.

A. Class Diagrams

The interactions between classes fall into four broad categories as previously discussed.

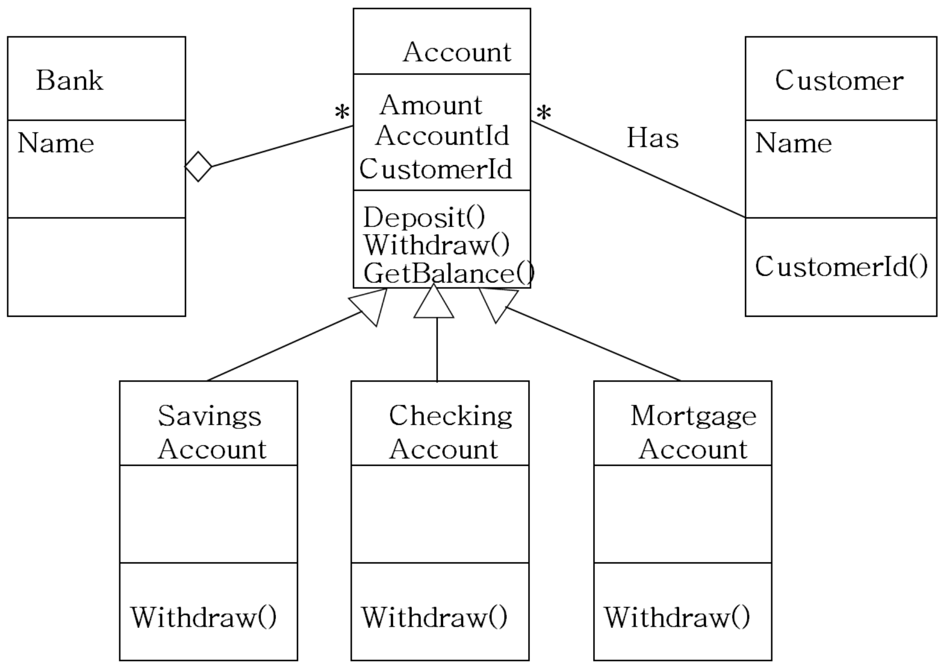
- Association
- Inheritance or also called generalization
- Composition
- Aggregation

Class diagrams show a set of classes, interfaces, and collaborations and their relationships. They are the commonest diagrams found in modeling object-oriented systems. They address the **static design view** of a system and are used at the analysis stage as well as design:

- during requirements analysis to model problem domain concepts
- during system design to model subsystems and interfaces
- during object design to model classes.

Class Diagram syntax is used to draw a plan of the major concepts for anyone to understand – the **Conceptual Model** during **Analysis.** Together with use cases, a conceptual diagram is a powerful technique in requirements analysis.

Example of Class Diagram

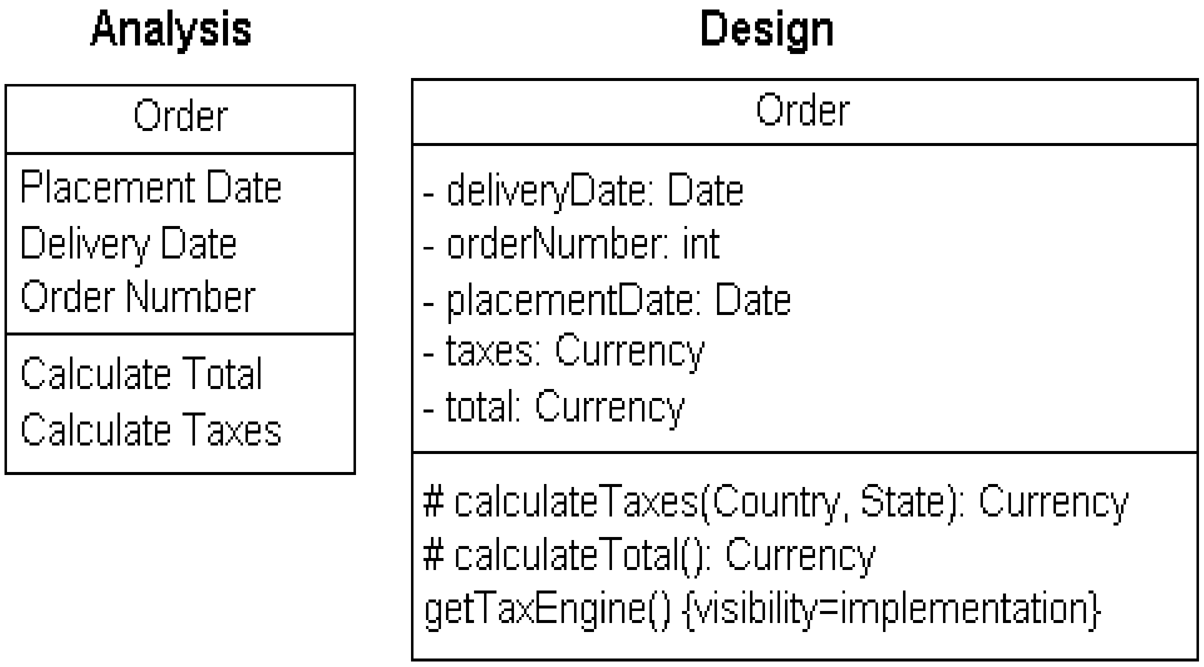


Design Class Model

After building a class (conceptual or Domain) diagram (made using concepts having properties of these concepts), one might realise that no behaviour was allocated to any of these concepts.

Therefore after creating collaboration diagrams, progress of the conceptual model can be done, to build it into a true "Design Class Diagram"

This DCD diagram shows attribute visibility for permanent connections and be a base for final program code.



DCD Notation

The following optional characters in front of class attributes or methods depict meaning as shown below:

- **-** Private (accessible only within the body of the class in which they are declared)
- # Protected (accessible only from within the class in which it is declared, and from within any class derived from the class that declared this member)
- + Public (no restrictions on access)

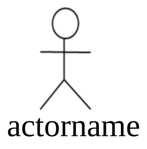
If the method or attribute is listed in italics text, it is **abstract**. If text is underlined, the method or attribute is **static**.

B. Use Case Modelling
This is performed during the analysis phase to help developers understand functional requirements of the system without regard for implementation details. A use-case is a scenario describing a series of events involving actors which describes one functionality the system should have.
A use case model is made up of Use Case Diagrams and accompanying Use Case Descriptions.

a. Use Case Diagrams
These are static diagrams used for modelling and analysis of functional requirements of a system i.e. they specify only what a system is supposed to do.
They describe sequences of actions a system performs that yield an observable result of value to a particular actor and model actions of the system at its external interface.
Use case diagrams represent the high level view of the system.

Notation

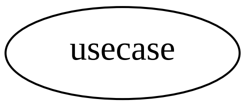
- Actor



An actor is an external entity that interacts with the system. An actor can be a:
  - **User**
  - **External system**
  - **Physical environment**

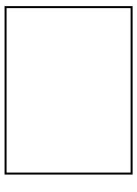
An actor has a unique name and an optional description.

- Use Case



A use case is a complete sequence of related actions initiated by an actor. The symbol for a use case is an oval with a label that describes the action or event.
Use cases can also interact and participate in relationships with other use cases. There are two types of use case relationships
  - Extends: a use case adds new behavior or actions to another use case.
  - Include: one use case references another use case.

- System Boundary

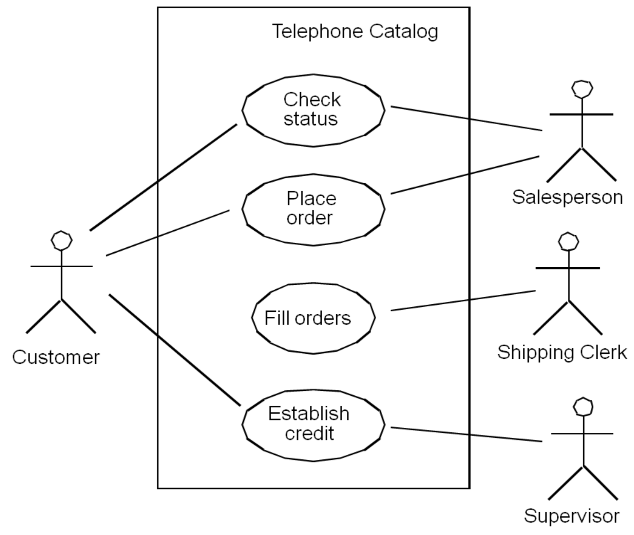


Represents the boundary between the physical system and the actors who interact with the physical system.

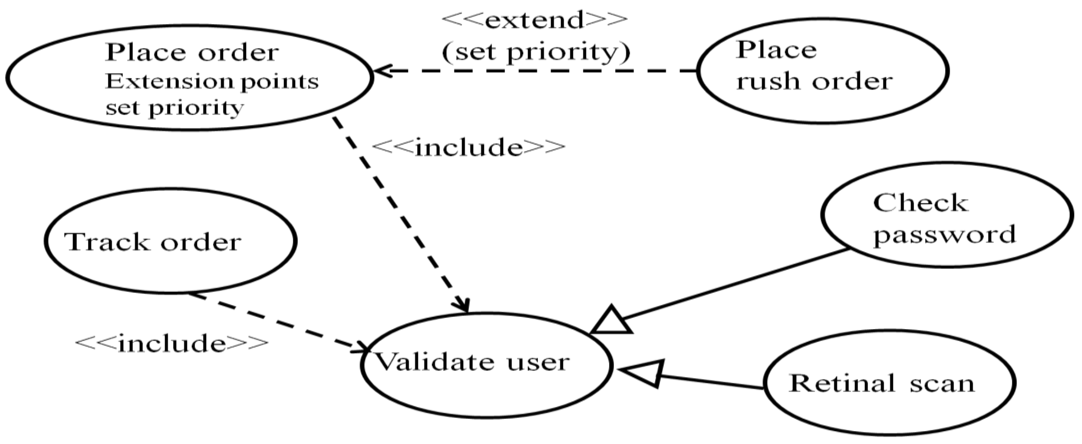
Use Case Relationships

Construct	Description	Syntax
association	The participation of an actor in a use case. i.e., instance of an actor and instances of a use case communicate with each other.	———
generalization	A taxonomic relationship between a more general use case and a more specific use case.	———▷
extend	A relationship from an *extension* use case to a *base* use case, specifying how the behavior for the extension use case can be inserted into the behavior defined for the base use case.	<<extends>> - - - - - ->
include	An relationship from a *base* use case to an *inclusion* use case, specifying how the behavior for the inclusion use case is inserted into the behavior defined for the base use case.	<<includes> - - - - - ->

Use Case Example



Example of Use Case Relationship



Purposes of Use Case diagrams

- To provide a graphic over view of system functionality (scope).
- To provide a bridge between developers and client/user. The notation is sufficiently simple to enable a non-user to understand and comment.
- To provide a starting point for developing detailed requirements.
- To provide an important basis for developing design.
- To provide integration test cases.

Use Case Modelling Tips

- Make sure that each use case describes a significant chunk of system usage that is understandable by both domain experts and programmers; imagine scenarios of use and make sure there is a use case about them
- When defining use cases in text, use nouns and verbs accurately and consistently to help derive objects and messages for interaction diagrams
- Factor out common usages that are required by multiple use cases
  - If the usage is required use <<include>>
  - If the base use case is complete and the usage may be optional, consider use <<extend>>
- A use case diagram should
  - contain only use cases at the same level of abstraction
  - include only actors who are required

Steps in Use Case Modelling

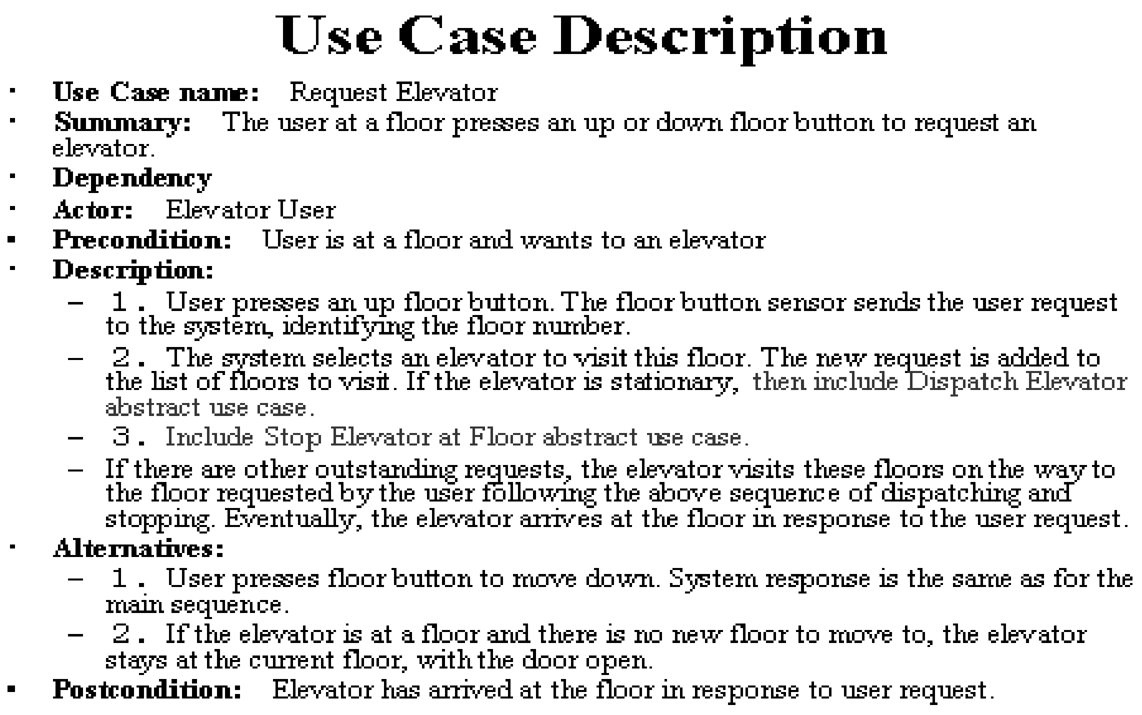
- Establish the context (**actors**).
- Consider **behavior** expected by an actor.
- Name the common behaviors as **use cases**.
- Create **use case descriptions**.
- Factor common behavior into **new** use cases
- Model use cases, actors and their relationships in a **use case diagram**.

b. Use Case description

A use case description is a formal documentation of each use case as represented on the use case diagram. It provides a specification of the use case diagram and includes further details for design.

Format
Sample fields in a use case description include:

- Name of use case
- Brief description
- Flows of Events
- Preconditions
- Postconditions
- Interaction, activity and state diagrams
- Relationships
- Use-Case diagrams
- Special requirements
- Other diagrams (UML diagrams associated with the use-case)



Use Case Description

- **Use Case name:** Request Elevator
- **Summary:** The user at a floor presses an up or down floor button to request an elevator.
- **Dependency**
- **Actor:** Elevator User
- **Precondition:** User is at a floor and wants to an elevator
- **Description:**
  - 1. User presses an up floor button. The floor button sensor sends the user request to the system, identifying the floor number.
  - 2. The system selects an elevator to visit this floor. The new request is added to the list of floors to visit. If the elevator is stationary, then include Dispatch Elevator abstract use case.
  - 3. Include Stop Elevator at Floor abstract use case.
  - If there are other outstanding requests, the elevator visits these floors on the way to the floor requested by the user following the above sequence of dispatching and stopping. Eventually, the elevator arrives at the floor in response to the user request.
- **Alternatives:**
  - 1. User presses floor button to move down. System response is the same as for the main sequence.
  - 2. If the elevator is at a floor and there is no new floor to move to, the elevator stays at the current floor, with the door open.
- **Postcondition:** Elevator has arrived at the floor in response to user request.

Use Case Scenario
A scenario is a description of a use case as a story - a story of one user's interaction with the system. By using scenarios, you are able to discover the objects that must collaborate to produce the results needed.
A scenario can be described in terms of:
—The actors involved, and
—The steps to be taken in order to achieve the functionality described in the use-case title.

The steps involved in a scenario usually take the form of the normal flow of events, followed by alternate flows and/or exception flows.

Scenario Example
For an ATM banking system, a use case would be "Validate User". The steps involved in authenticating a user are described in scenarios and there will be a number of different scenarios for "Validate User" describing different situations that can arise.

Main (primary) flow of events:
1. System prompts user for PIN number
2. User enters PIN number via keypad.
3. User commits PIN by pressing the Enter key.
4. System then checks PIN to see if it is valid.
5. If the PIN is valid, system acknowledges the entry.
6. End of Use Case

Exception Flow of events
1. System prompts user for PIN number
2. User enters PIN number via keypad.
3. User presses Cancel button to cancel transaction
4. End of Use case. No changes made to user account.

Further Reading
http://www.agilemodeling.com/style/classDiagram.htm
http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/
http://www.sparxsystems.com/resources/uml2_tutorial/
http://www.uml-diagrams.org/class-diagrams.html