OBJECT ORIENTED CONCEPTS

OBJECT

- **Objects** are the physical and conceptual things that exist in the universe.
- In programming terms, an object is a self-contained component that contains properties and methods needed to make a certain type of data useful.
- Object is an instance of a class.

CLASS

- A ***class*** is a blueprint or template or set of instructions to build a specific type of object.
- In programming terms, a class is a way to bind, data and functions into a single unit.
- A class is an abstraction that describes important properties of object.

CLASS vs OBJECT

Parameters	CLASS	OBJECT
Definition	Class is mechanism of binding data members and associated methods in a single unit.	Instance of class or variable of class.
Existence	It is logical existence	It is physical existence
Memory Allocation	Memory space is not allocated, when it is created.	Memory space is allocated, when it is created.
Declaration/definition	Definition is created once.	It is created many time as you require.

WHAT IS OBJECT ORIENTED?

- Software is organised as a collection of discrete objects that incorporate both data structure and behaviour.
- It includes identity, classification, polymorphism and inheritance.

IDENTITY

- ✓ Identity means that data is organized into discrete distinguishable entities called objects.
- ✓ Objects can be physical or conceptual.
- ✓ In real world object simply exist whereas in programming language each object has a unique handle (unique identifier) by which it can be uniquely referenced.
- ✓ The handle can be implemented by address, array index or unique value of an attribute.

CLASSIFICATION

- ✓ It means that objects with same data structure (attribute) and behaviour (operations) are grouped into a class.
- ✓ A class is an abstraction that describes important properties and ignores the rest.

POLYMORPHISM

- ✓ It means that the same operation (action or transformation that the object performs) may behave differently on different classes.
- ✓ Specific implementation of an operation by a certain class is called a method.

INHERITANCE

- ✓ It is the sharing of attributes and operations among classes based on a hierarchical relationship.
- ✓ Subclasses can be formed from broadly defined class.
- ✓ Each subclass incorporates or inherit all the properties of its super class and add its own unique properties.

OBJECT ORIENTED METHODOLOGY

- ➢ **Object Oriented Methodology** is a methodology for object oriented development and a graphical notation for representing objects oriented concepts. This methodology is termed as OMT. The methodology has the following stages:
  1. Analysis
  2. System design
  3. Object design
  4. Implementation

ANALYSIS

- ✓ An analysis model is a concise, precise abstraction of what the desired system must do, not how it will be done. It should not contain any implementation details. The objects in the model should be application domain concepts and not the computer implementation concepts.

SYSTEM DESIGN

- ✓ The designer makes high level decisions about the overall architecture. In system design, the target system is organized as various subsystems based on both the analysis structure and the proposed architecture.

OBJECT DESIGN

- ✓ The designer builds a design model based on the analysis model but containing implementation details. The focus of object design is the data structures and algorithms needed to implement each cycle.
- ✓ The Objects identified in the system design phase are designed. Here the implementation of these objects is decided in the form of data structures required and the interrelationships between the objects.

IMPLEMENTATION

- ✓ The object classes and relationships developed during object design are finally translated into a particular object oriented programming language, database, or hardware implementation.
- ✓ During implementation, it is important to follow good software engineering practice so that the system can remain the traceability, flexibility and extensibility.

ADVANTAGES OF OBJECT ORIENTED METHODOLOGY

- The systems designed using OOM are closer to the real world as the real world functioning of the system is directly mapped into the system designed using OOM. Because of this, it becomes easier to produce and understand designs.
- The objects in the system are immune to requirement changes because of data hiding and encapsulation features of object-orientation.
- OOM designs encourage more reusability. The classes once defined can easily be used by other applications. This is achieved by defining classes and putting them into a library of classes where all the classes are maintained for future use. Whenever a new class is needed the programmer first looks into the library of classes and if it is available, it can be used as it is or with some modification. This reduces the development cost & time and increases quality.
- Another way of reusability is provided by the inheritance feature of the object-orientation. The concept of inheritance allows the programmer to use the existing classes in new applications i.e. by making small additions to the existing classes can quickly create new classes.
- As the programmer has to spend less time and effort so he can utilize saved time (due to the reusability feature of the OOM) in concentrating on other aspects of the system.

OBJECT ORIENTED CONCEPTS

❖ ABSTRACTION

- ✓ Abstraction is "To represent the essential feature without representing the back ground details."
- ✓ Abstraction makes to focus on what the object does instead of how it does it.
- ✓ Abstraction provides you a generalized view of your classes or object by providing relevant information.
- ✓ Abstraction is the process of hiding the working style of an object, and showing the information of an object in understandable manner.

❖ ENCAPSULATION

- ✓ Wrapping up data member and method together into a single unit (i.e. Class) is called Encapsulation.
- ✓ Encapsulation is like enclosing in a capsule. That is enclosing the related operations and data related to an object into that object.

- ✓ Encapsulation means hiding the internal details of an object, i.e. how an object does something.
- ✓ Encapsulation prevents clients from seeing its inside view, where the behaviour of the abstraction is implemented.
- ✓ Encapsulation is a technique used to protect the information in an object from the other object.

❖ POLYMORPHISM

- ✓ Polymorphism means **one name many forms**. It is the property of an object to behave differently in different instances.
- ✓ One function behaves different forms.
- ✓ In other words, "Many forms of a single object is called Polymorphism."

❖ INHERITANCE

- ✓ Inheritance is the capability of one class to acquire properties and characteristics from another class.
- ✓ The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.
- ✓ Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.
- ✓ All members of a class except Private, are inherited
- ✓ **Purpose of Inheritance**

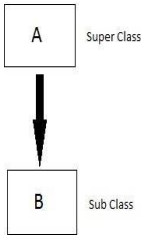
1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

- ✓ **The various types of Inheritance are listed below:**

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance

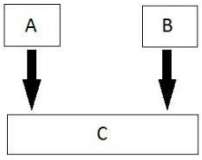
Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance.



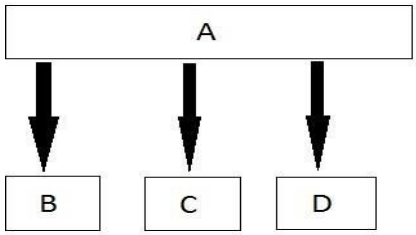
Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



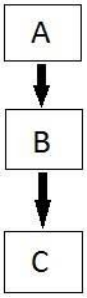
Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



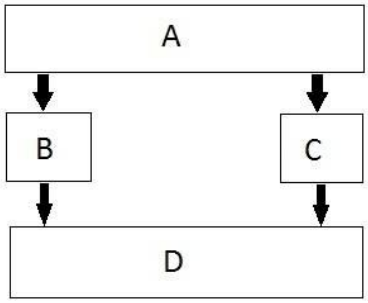
Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.



DIFFERENCE BETWEEN ABSTRACTION AND ENCAPSULATION

Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. **Abstraction**- Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	4. **Encapsulation**- Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

2. MODELING & OBJECT MODEL

MODELING

- A model is an abstraction of something for the purpose of understanding it before building it.
- A model is also considered as a miniature form of a system.
- A model as a simplified representation of reality. A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form.
- The word model has two dimensions:
  i) A view of a system ii) A stage of development

PURPOSE OF MODELING

- Testing a physical entity before building it.
- Provides easier communication with the customers / stakeholders.
- Can be used as a better visualization aid.
- Helps to reduce complexity during the actual implementation.

OMT METHODOLOGY

- OMT (Object Modelling Techniques) is an object-oriented software development methodology given by James Rumbaugh et.al. This methodology describes a method for analysis, design and implementation of a system using object-oriented technique.
- The OMT consists of three related but different viewpoints each capturing important aspects of the system i.e. the static, dynamic and functional behaviours of the system. These are described by object model, dynamic model and functional model of the OMT.
- The **object model** describes the static, structural and data aspects of a system.
- The **dynamic model** describes the temporal, behavioural and control aspects of a system.
- The **functional model** describes the transformational and functional aspects of a system.

MODELS IN OMT / OBJECT ORIENTED MODELS

Object model

- Describes basic structure of objects and their relationships.
- Contains object diagram.
- Object diagram is a graph whose nodes (vertices) are object classes (classes) and whose arcs (edges) are relationships among classes.

Dynamic model

- Describes the aspects of a system that change over time.
- It specifies and implement control aspects of a system.
- Contains state diagram.
- State diagram is a graph whose nodes (vertices) are states and whose arcs (edges) are transitions.

Functional model

- Describes the data value transformation within a system.
- It represents how the input data is converted into the required output data.
- Contains data flow diagrams (DFD).

- DFD is a graph whose nodes are process and whose arcs are data-flows.

OBJECT vs DYNAMIC vs FUNCTIONAL MODELS

MODEL	REPRESENTS	DIAGRAMS	GRAPH REPRESENTATION	
			VERTICES	EDGES
OBJECT	Static or structural features of a system.	Class diagram	Class	Relationship
DYNAMIC	Aspects of a system that change over time.	State diagram	State	Transition
FUNCTIONAL	Data transformation in a system.	Data Flow Diagram	Process	Data / Dataflow

OBJECT MODELING

- The object model describes the structure of the objects in the system - their identity, their relationships to other objects, their attributes, and their operations.
- The object model is represented graphically with an object diagram. The object diagram contains classes interconnected by association lines.
- Each class represents a set of individual objects. The association lines establish relationships among classes. Each association line represents a set of links from the object of one class to the object of another class.

OBJECT AND CLASS

- A class describes a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined.
- A class defines the basic attributes and the operations of the objects of that type. Defining a class does not define any object, but it only creates a template. For objects to be actually created, instances of the class are to be created as per the requirement of the case.

REPRESENTATION OF CLASS IN OMT

- In OMT, classes are represented by a rectangular box which may be divided into three parts as shown in Figure below. The top part contains the name of the class, middle part contains a list of attributes and bottom part contains a list of operations.

Class_Name
Attribute-name1:data-type1=default-val1 Attribute-name2:data-type2=default-val2
Operation-name1(arguments1):result-type1 Operation-name2(arguments2):result-type2

- Class name represents the name of the class. The class name should start with a character followed by combination of letters or digits. The class name should begin with upper-case letter. No white spaces are allowed in class names. Underscore (_) can be used.

- An attribute is a data value held by objects in a class. Each attribute has a value for each object instance. This value should be a pure data value, not an object. Attributes are listed in the second part of the class box. Attributes may or may not be shown; it depends on the level of detail desired. Each attribute name may be followed by the optional details such as type and default value. An object model should generally distinguish independent base attributes from dependent derived attributes. A derived attribute is that which is derived from other attributes. For example, age is a derived attribute, as it can be derived from date-of-birth and current-date attributes.
- An operation is a function or transformation that may be applied to or by objects in a class. Operations are listed in the third part of the class box. Operations may or may not be shown; it depends on the level of detail desired. Each operation may be followed by optional details such as argument list and result type. The name and type of each argument may be given. An empty argument list in parentheses shows explicitly that there are no arguments.
- Examples for class representations

Class Book:

Book
title: string author:string publisher:string
open() close() read()

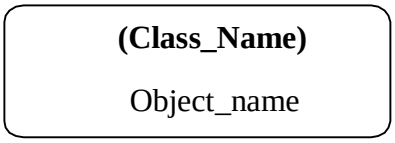
Class Student

Student
Name: string Branch:string Address:string Mark2:integer Mark1:integer

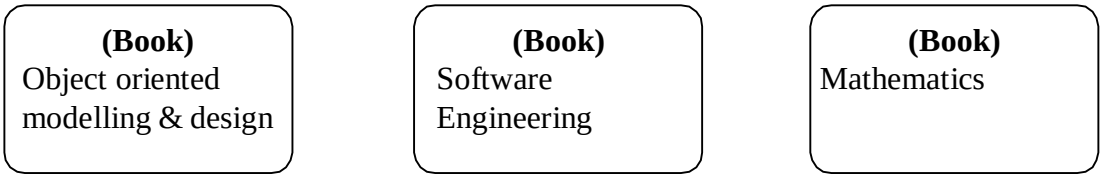
Calc_marks() Calc_per() View_details()

OBJECT

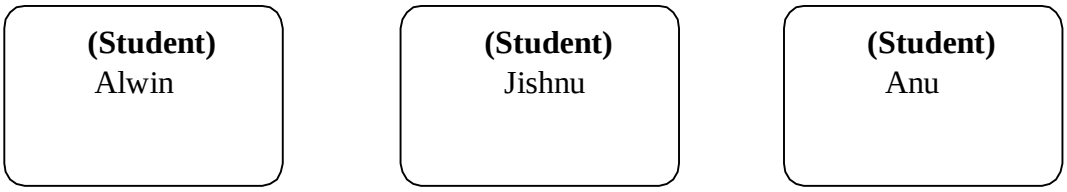
- An object is an instance of an object class.
- An object has the following four main characteristics:
  - Unique identification
  - Set of attributes
  - Set of states
  - Set of operations (behaviour)
- Every object has a unique name by which it is identified in the system.
- Rounded box represents an object instance in OMT.
- Object instance is a particular object from an object class.
- The box may/may not be divided in particular regions. Object instances can be used in instance diagrams, which are useful for documenting test cases and discussing examples.



Examples of objects of class Book:

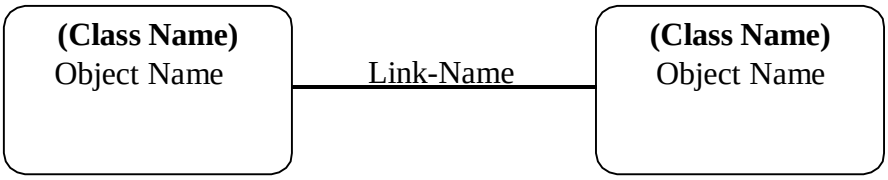


Examples of objects of class Student:

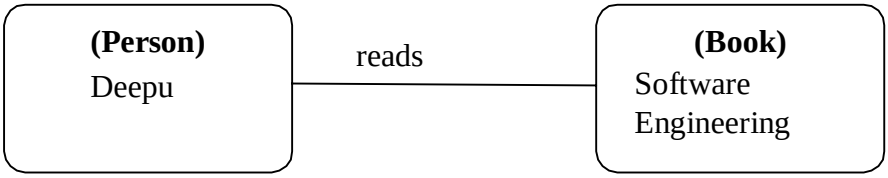


LINKS

- A link is a physical or conceptual connection between object instances. In OMT, link is represented by a line labelled with its name as shown below.



For example:



ASSOCIATIONS

- An association describes a group of links with common structure and common semantics between two or more classes. Association is represented by a line labelled with the association name as shown below.

- Association names are optional. If the association is given a name, it should be written above the line.
- In case of a binary association, the name reads in a particular direction (i.e. left to right), but the binary association can be traversed in either direction.
- For example, a student studies a subject or a subject is studied by a student.
- All the links in an association connect objects from the same classes. Associations are bidirectional in nature.

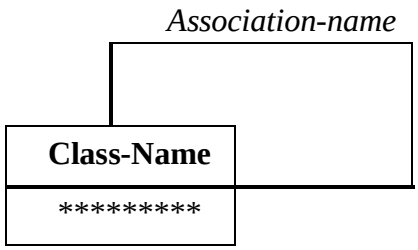
MULTIPLICITY

- It specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects.
- There are special line terminators to indicate certain common multiplicity values.
  - A solid ball is the symbol for "many", meaning zero, one or more.
  - A hollow ball indicates "optional", meaning zero or one.
- The multiplicity is indicated with special symbols at the ends of association lines.

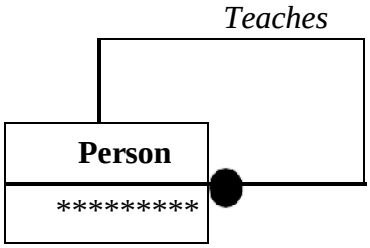
- In the most general case, multiplicity can be specified with a number or set of intervals. If no multiplicity symbol is specified that means a one-to-one association.
- The rules of multiplicity are summarized below:
  - Line without any ball indicates one-to-one association.
  - Hollow ball indicates zero or one.
  - Solid ball indicates zero, one or more.
  - Numbers written on solid ball such as 1,2,6 indicates 1 or 2 or 6.
  - Numbers written on solid ball such as 1+ indicates 1 or more, 2+ indicates 2 or more

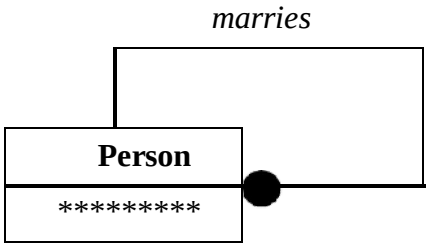
TYPES OF ASSOCIATIONS

- An association can be unary, binary, ternary or n-ary.
- **Unary association** is between the same class as shown below

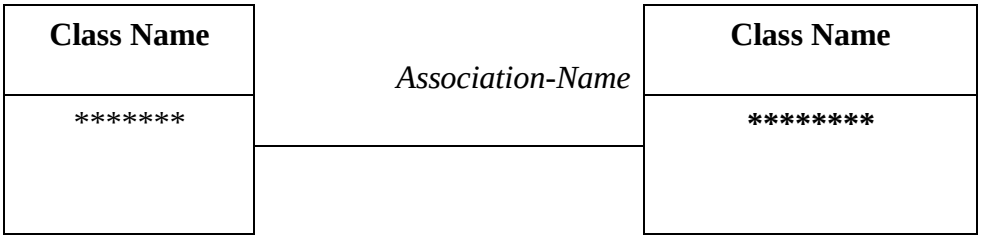


- Example of unary association is Person teaches Person as shown below. Other examples of unary association can be Person marries a Person, Person manages Person etc.

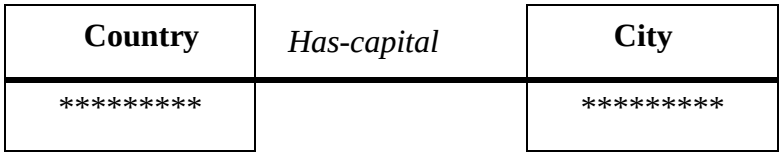




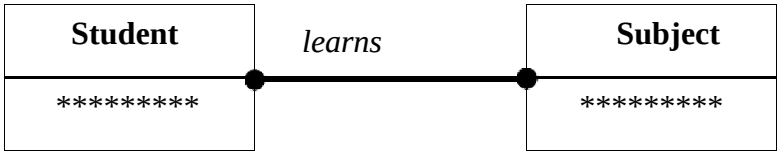
- A **binary association** is an association between two classes as shown below



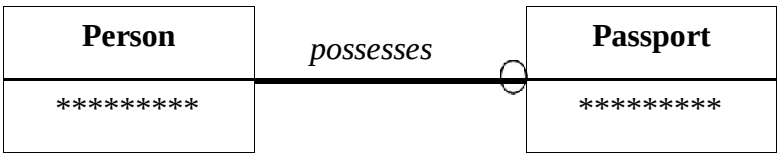
- Example of a binary association is "Country has capital city". One country has only one capital city. So multiplicity of this association is one-to-one as shown below.



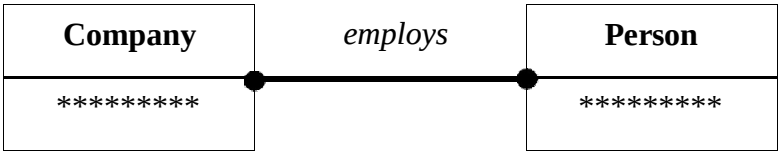
Another example of a binary association is “Student learns subject”. One student can learn many subjects or one subject can be learnt by many students. So multiplicity of this association is many-to-many as shown below.



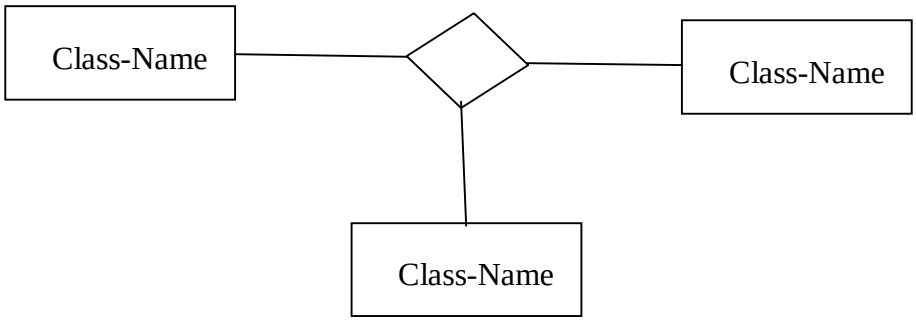
- Another example of binary association is “Person possesses a Passport”. Either a person can have one passport or no passport but one passport can be with one person. So multiplicity of this association is one-to-optional as shown below.



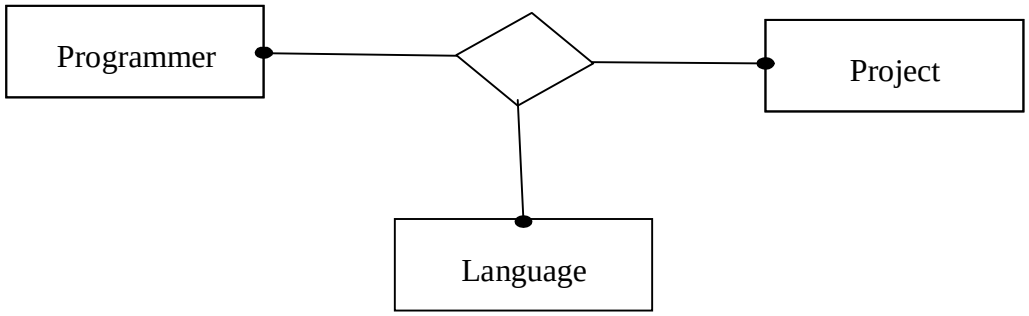
- Another example of binary association is “Company employs Person”. One company can employ zero, one or more persons but one person can be employed in one company only (assume). So multiplicity of this association is one-to-many as shown below.



- **Ternary association** is an association among three classes. On the same line, n-ary association is an association among n classes.
- The OMT symbol for ternary and n-ary associations is a diamond with lines connecting to related classes as shown below. A name for the association is optional and is written next to the diamond. An n-ary associations cannot be subdivided into binary associations without losing information.



Example of a ternary association. Programmers develop Projects in (programming) Languages. One programmer can be engaged in zero, one or more projects and can know zero, one or languages. Similarly, one project can be developed by one or more programmersand in one or more languages. So this association along with its multiplicity is shown below.

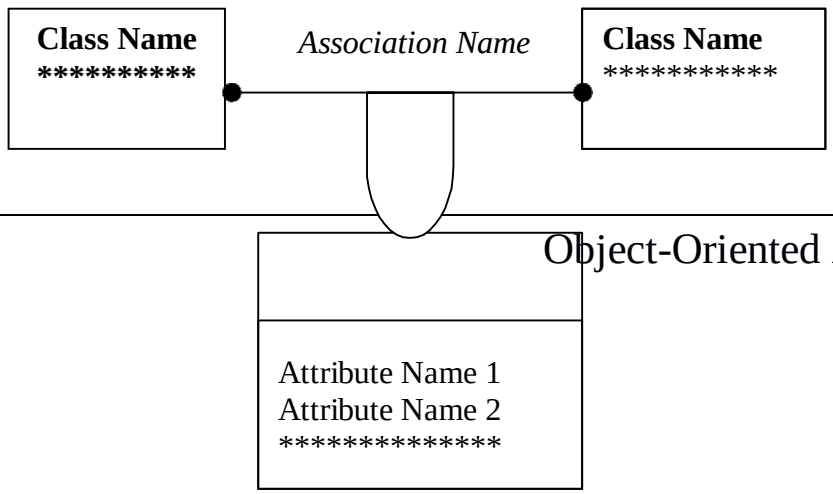


ADVANCED OBJECT MODELLING CONCEPTS

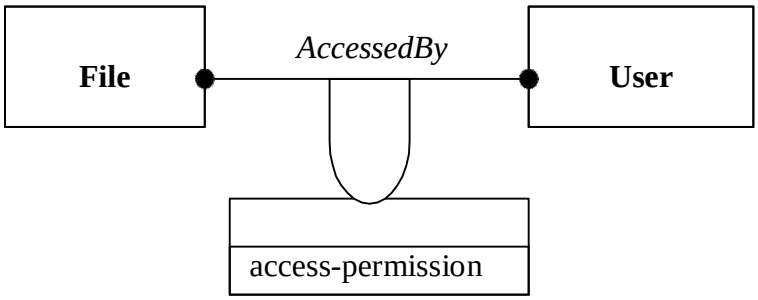
LINK ATTRIBUTES

- Sometimes, an attribute(s) cannot be associated with either of the two classes associated by the association. In such cases, the attribute(s) is associated with the association and is called as link attribute. It is a property of the links in an association.

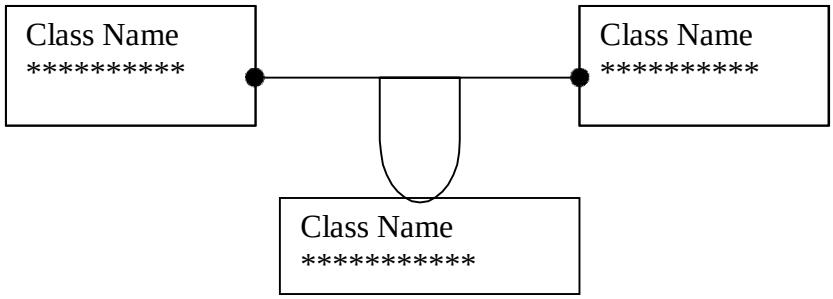
- The OMT notation for a link attribute is a box attached to the association by a loop as shown below.



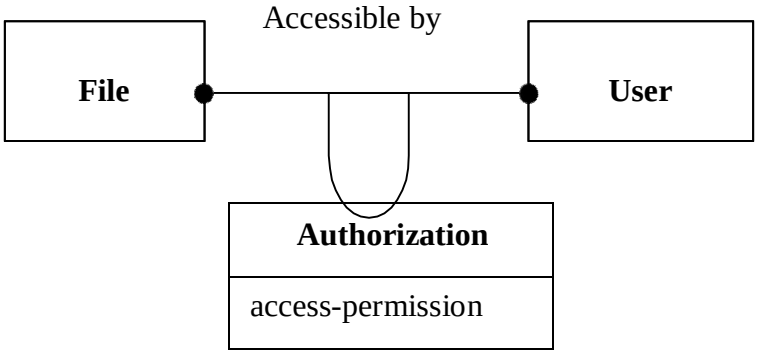
- For example: A File is accessed by a User. So, the classes File and User are related by association, the association named "AccessedBy". Many users can access one file and one user can access many files. So, multiplicity of the association is many-to-many. Now, the attribute "access-permission" cannot be associated with either File class or with User class. This attribute can be associated with the link as shown below. Hence, access-permission is link attribute.



- It is also possible to model an association as a class such class is called as link class as shown in below figure. Each link becomes one instance of the class. The notation for this kind of association is the same as for a link attribute and has a name and (optional) operations added to it.

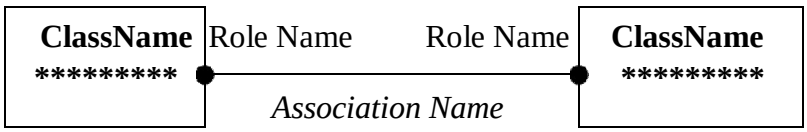


- Now, consider the example shown in below figure, where whole class is associated with the link. In this example, the class Authorization is a link class. It has one attribute "access-permission" and two methods grantPermission() and changePermission().



ROLE NAMES

- A role is one end of an association. A binary association can have two roles, each of which may have a role name. A role name is a name that uniquely identifies one end of an association.
- Roles provide a way of viewing a binary association as a traversal from one object to a set of associated objects. Each role on a binary association identifies an object or set of objects associated with an object at the other end.

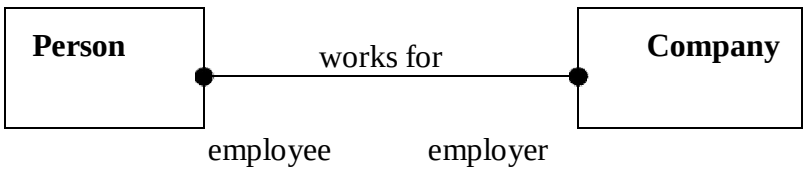


- The use of role names is optional, but is often easier and less confusing to assign role names instead of, or in addition to, association names. Role names are necessary for associations between two objects of the same class. They are also useful to distinguish between two associations between the same pair of classes.
- Follow these two guidelines for role names:

  i) All role names on the far end of associations attached to a class must be unique.

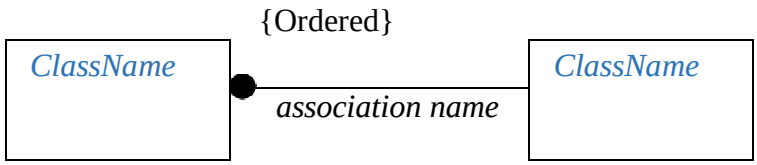
  ii) No role name should be the same as an attribute name of the source class. It is also possible to use role names for n-ary associations.

- The role name is a derived attribute whose value is a set of related objects. Use of role names provides a way of traversing associations from an object at one end, without explicitly mentioning the association.
- For example, consider the association ‘a person works for a company’, in this employee and employer are role names for the classes Person and Company respectively as shown in figure.

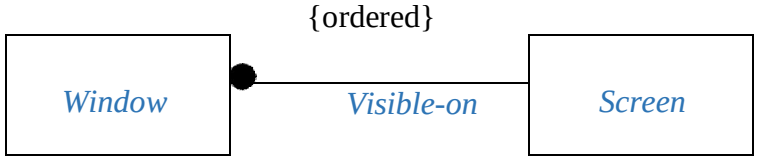


ORDERING

- Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes the objects on the many side of an association have order. Writing {ordered} next to the multiplicity dot as shown in figure below,indicates an ordered set of objects of an association.

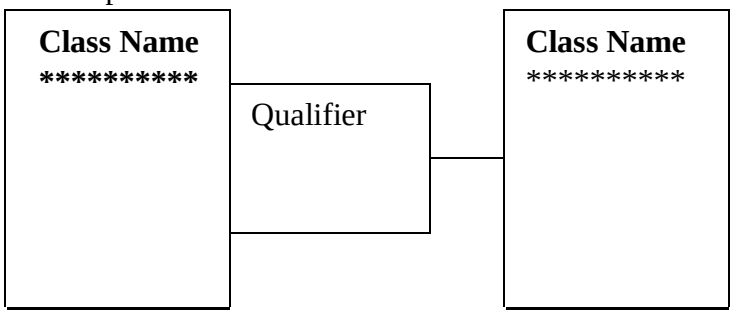


- Consider the example of association between Window class and Screen class. A screen can contain a number of windows. Windows are explicitly ordered. Only topmost window is visible on the screen at any time. Figure 2.8 shows this example.

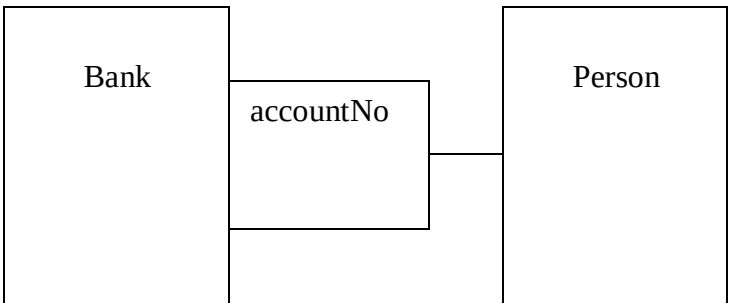


QUALIFICATION

- A qualifier is an association attribute. A qualified association relates two object classes and a qualifier. The qualifier is a special attribute that reduces the effective multiplicity of an association. One-to-many and many-to-many associations may be qualified.
- Qualifier is represented as shown below:



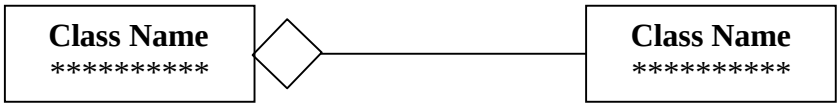
- The qualifier is drawn as a small box on the end of the association line near the class it qualifies. The qualifier rectangle is part of the association, not of class. The qualifier distinguishes among the set of objects at the "many" end of an association. A qualified association can also be considered a form of ternary association. The advantage of the qualification is that it improves semantic accuracy and increases the visibility of navigation paths.
- For example, a person object may be associated to a Bank object as shown in figure below. An attribute of this association is the accoutNo. The accountNo is the qualifier of this association.


Bank
accountNo
Person

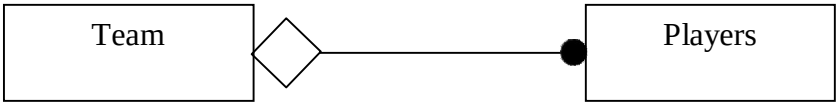
AGGREGATION

- Aggregation is another relationship between classes. It is a tightly coupled form of association with some extra semantics. It is the "part-whole" or "a-part-of" relationship in which objects representing the component of something are associated with an object representing the entire assembly.

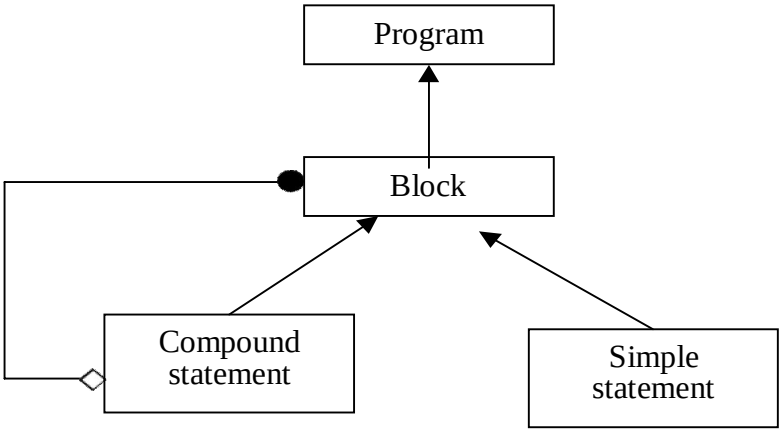
- Aggregations are drawn like associations, except a small hollow diamond indicating the assembly end of the relationship as shown in figure below. The class opposite to the diamond side is part of the class on the diamond side.



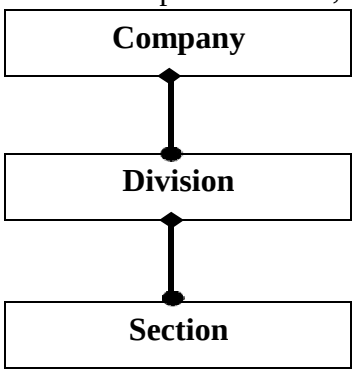
- For example, a team is aggregation of players. This can be modelled as shown below



- Aggregation can be fixed, variable or recursive.
  - In a fixed aggregation number and subtypes are fixed i.e. predefined.
  - In a variable aggregation number of parts may vary but number of levels is finite.
  - A recursive aggregate contains, directly or indirectly, an instance of the same aggregate. The number of levels is unlimited. For example, as shown in figure below, a computer program is an aggregation of blocks, with optionally recursive compound statements. The recursion terminates with simple statement. Blocks can be nested to arbitrary depth.



- One more example of aggregation is shown below. A company is composed of zero, one or more divisions. A division is composed of zero, one or more sections.



- Another example: A pen consists of refill, A refill consist of nib.

INHERITANCE

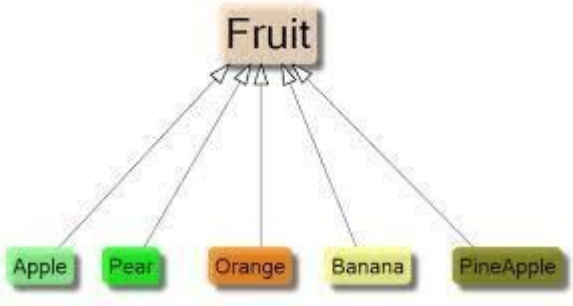
- Inheritance is a way to form new classes using classes that have already been defined. Inheritance is intended to help reuse existing code with little or no modification. The new classes, known as derived classes (or child classes or sub classes), inherit attributes and behaviour of the pre-existing classes, which are referred to as base classes (or parent classes or super classes) as shown below The inheritance relationship of sub- and super classes gives rise to a hierarchy.
- Inheritance is referred as generalization relationship.

GENERALIZATION

- The Generalization association ("is a") is the relationship between the base class that is named as “superclass” or “parent” and the specific class that is named as “subclass” or “child”.

Inheritance is a “is-a” relationship between two classes. For example, Student is a Person; Chair is Furniture; Parrot is a Bird etc. in all these examples, first class (i.e. Student, Chair, Parrot) inherits properties from the second class (i.e. Person, Furniture, Bird).

- The Generalization association is also known as Inheritance. The subclass is a particular case of the superclass and inherits all attributes and operations of superclass, but can have your own additional attributes and operations.
- In UML is used also the multiple inheritance when the subclass inherits properties and behaviours of more than one superclass.
- On the UML Diagram the Generalization association represents as the line with empty triangle that connects superclass and subclass as shown below.



- There are several reasons to use inheritance as enumerated below:

- Inheritance for specialization
- Inheritance for generalization
- Inheritance for extension
- Inheritance for restriction
- Inheritance for overriding

GROUPING CONSTRUCTS

- There are two grouping constructs: module and sheet.

Module is logical construct for grouping classes, associations and generalizations. An object model consists of one or more modules. The module name is usually listed at the top of each sheet.

A **sheet** is a single printed page. Sheet is the mechanism for breaking a large object model into a series of pages. Each module is contained in one or more sheets. Sheet numbers or sheet names inside circle contiguous to a class box indicate other sheets that refer to a class.

3. DYNAMIC & FUNCTIONAL MODELLING

DYNAMIC MODELING

- Dynamic model describes those aspects of the system that changes with the time.
- It is used to specify and implement control aspects of the system. It depicts states, transitions, events and actions.
- The dynamic model includes event trace diagrams describing scenarios.
- An event is an external stimulus from one object to another, which occurs at a particular point in time. An event is a one-way transmission of information from one object to another.
- A scenario is a sequence of events that occurs during one particular execution of a system. Each basic execution of the system should be represented as a scenario.
- The dynamic model is represented graphically by state diagrams.
- A state corresponds to the interval between two events received by an object and describes the "value" of the object for that time period.
- A state is an abstraction of an object's attribute values and links, where sets of values are grouped together into a state according to properties that affect the general behaviour of the object.
- Each state diagram shows the state and event sequences permitted in a system for one object class.

SCENARIO

- A scenario is a sequence of events that occurs during one particular execution of a system.
- Each basic execution of the system should be represented as a scenario.
- The scope of scenario may vary. It may include all events in the system or it may include only those events generated by certain objects. A scenario can be written as a list of text statements.

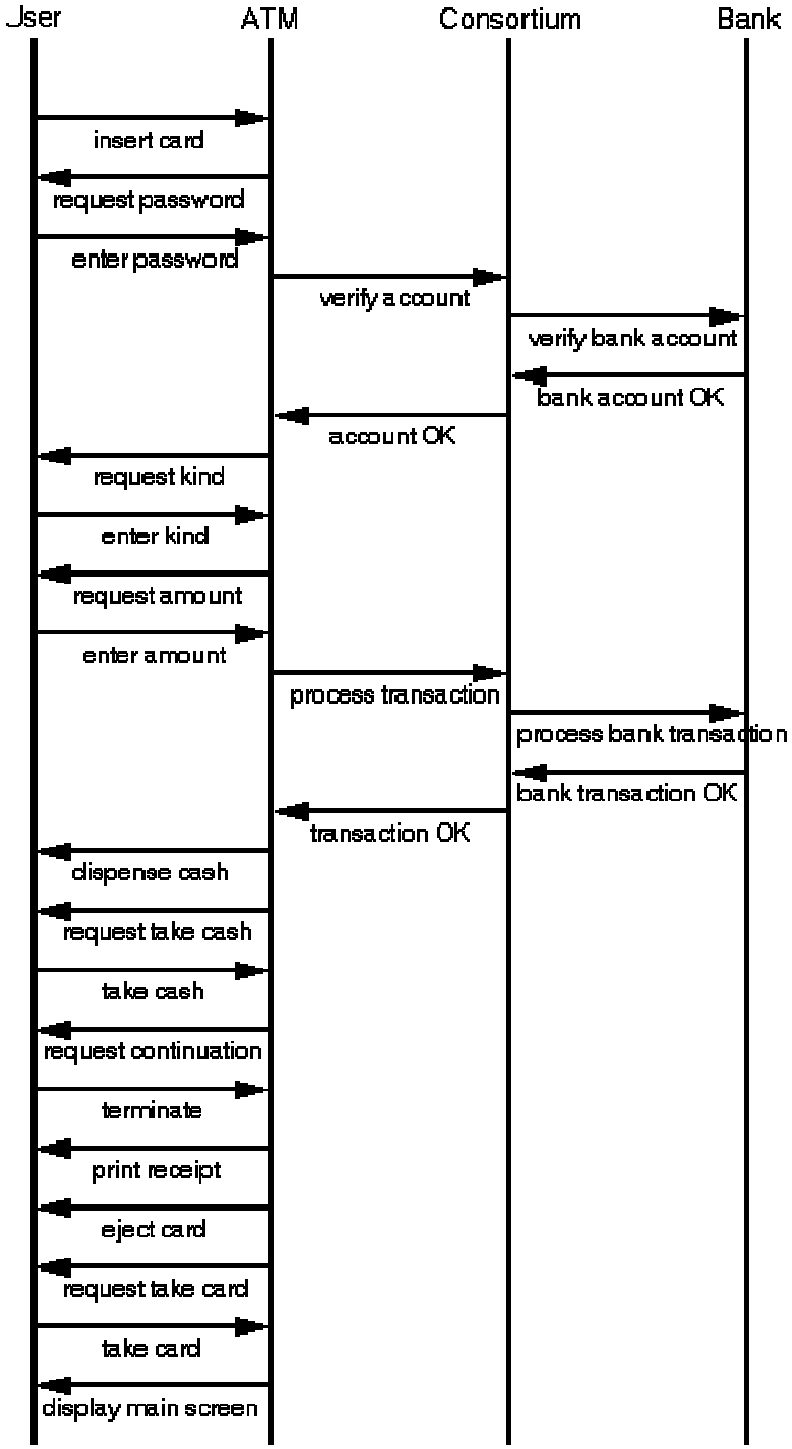
Example: A scenario to use ATM for withdrawing money. Each event transmits information from one object to another. For example, the event "the ATM asks the user to insert a card" transmits a signal from the ATM to the User. The next event is "the user inserts a cash card". The next event is "the ATM accepts the card and reads its serial no." and so on.

Normal ATM scenario	
1.	The ATM asks the user to insert a card The user inserts a cash card
2.	The ATM accepts the card and reads its serial number.
3.	The ATM requests the password The user enters 1234
4.	The ATM verifies the serial number and password with the consortium The consortium checks it with bank ABC and notifies the ATM of acceptance
5.	The ATM requests amount The user enters 15000
6.	The ATM processes the request and dispenses the required amount of money.

EVENT TRACE DIAGRAM

- The limitation of scenario is that it is not clear from scenario, how many objects are involved and which object generates an event and which object receives an event.
- To overcome this limitation, an event-trace diagram is introduced. In the event-trace diagram, the sequence of events and the objects exchanging events both can be shown.
- The diagram shows each object as a vertical line and each event as a horizontal arrow from the sender object to the receiver object. Time increases from top to bottom. Spacing between horizontal arrows carries no information. Figure 3.2 below shows the event-trace diagram for interaction with the ATM.

Example: In this diagram, four objects – User, ATM, Consortium and Bank - are involved, which are shown with four vertical lines. User generates an event "insert card" which is shown as horizontal arrow from user to ATM. That means source of the event is User object and destination of the event is ATM. In response to this event, the ATM generates the "request password" event to the User. Spacing between these two arrows is insignificant but the event "insert card" occurs before the event "request password" and so on.



STATE MACHINE

- A state machine is a behaviour which specifies the sequence of states an object visits during its lifetime in response to events, together with its responses to those events.
- **State:** A state is a condition during the life of an object during which it satisfies some condition, performs some activity, or waits for some external event.
- A state corresponds to the interval between two events received by an object and describes the "value" of the object for that time period.
- A state is an abstraction of an object's attribute values and links, where sets of values are grouped together into a state according to properties that affect the general behaviour of the object.
- A sub-state is a state that is nested in another state. A state that has sub-states is called a composite state. A state that has no sub-states is called a simple state. Sub-states may be nested to any level.
- **Event:** An event is the specification of a significant occurrence. For a state machine, an event is the occurrence of a stimulus that can trigger a state transition.
- In other words, an event is something that happens at a point in time. An event does not have duration.
- An individual stimulus from one object to another is an event. Press a key on the key board, train departs from station are examples of events.
- **Transition:** A transition is a relationship between two states indicating that an object in the first state will, when a specified set of events and conditions are satisfied, perform certain actions and enter the second state. Transition can be self-transition. It is a transition whose source and target states are the same.
- **Action:** An action is an executable, atomic (with reference to the state machine) computation. Actions may include operations, the creation or destruction of other objects, or the sending of signals to other objects (events).
- An action is an instantaneous operation. An action represents an operation whose duration is insignificant compared to the resolution of the state diagram. For instance, disconnect phone line might be an action in response to an on-hook event for the phone line. An action is associated with an event.
- **Activity:** Activity is an operation that takes time to complete. An activity is associated with a state. Activity includes continuous operations such as displaying a picture on a television screen as well as sequential operations that terminate by themselves after an interval of time such as closing a valve or performing a computation.
- A state may control a continuous activity such as ringing a telephone bell that persists until an event terminates it causing the transition of the state. Activity starts on entry to the state and stops on exit. A state may also control a sequential activity such as a robot moving a part that progresses until it completes or until it is interrupted by an event that terminates prematurely.

STATE DIAGRAM

- ✓ State diagrams are used to describe the behaviour of a system. State diagrams describe all of the possible states of an object as events occur.

- ✓ Each diagram usually represents objects of a single class and tracks the different states of its objects through the system.

- ✓ It relates events and states. A change of state caused by an event is called a transition. Transition is drawn as an arrow from the receiving state to the target state.

- ✓ A state diagram is graph whose nodes are states and whose directed arcs are transitions labelled by event names. State diagram specifies the state sequence caused by an event sequence.

When to use state diagrams ?

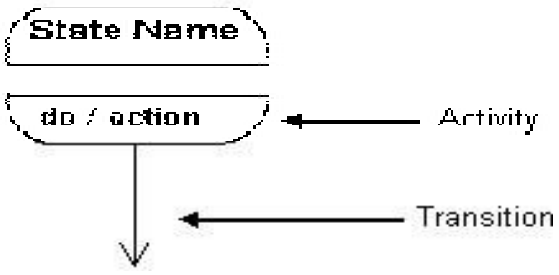
- ✓ Use state diagrams to demonstrate the behaviour of an object through many use cases of the system.

- ✓ Only use state diagrams for classes where it is necessary to understand the behaviour of the object through the entire system.

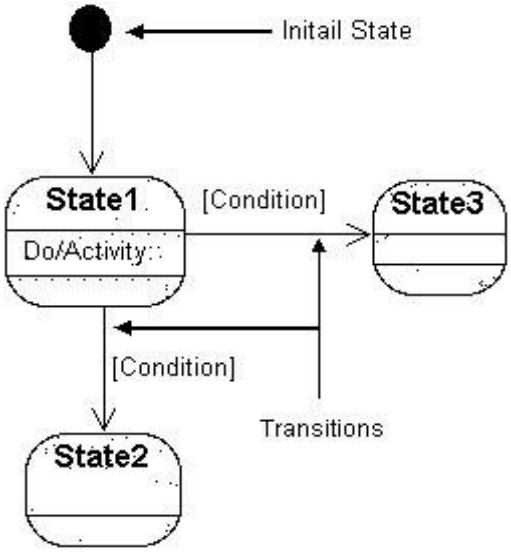
- ✓ Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case. State diagrams are combined with other diagrams such as interaction diagrams and activity diagrams.

How to draw state diagrams ?

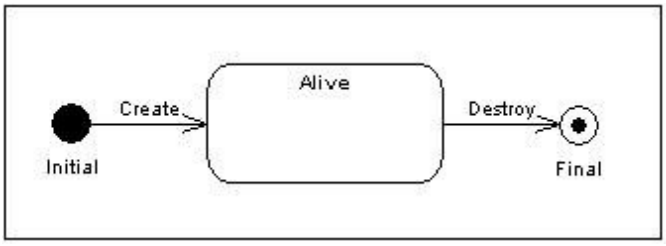
- ✓ State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state as shown in figure below.



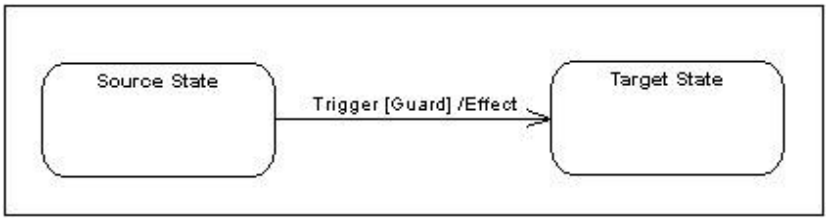
- ✓ **Initial and Final States:** All state diagrams being with an initial state of the object as shown below. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.



- ✓ The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name as shown in figure below



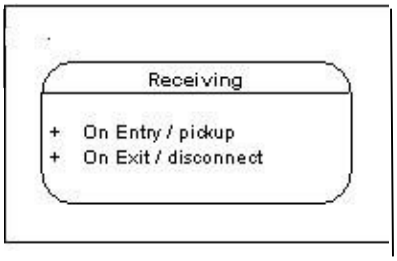
- ✓ **Transitions:** Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as shown in figure below.



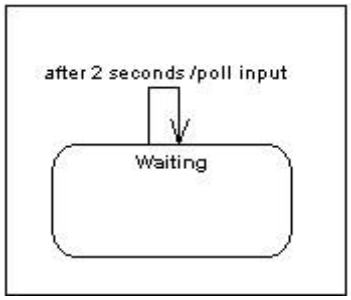
- ✓ "Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the

object that owns the state machine as a result of the transition.

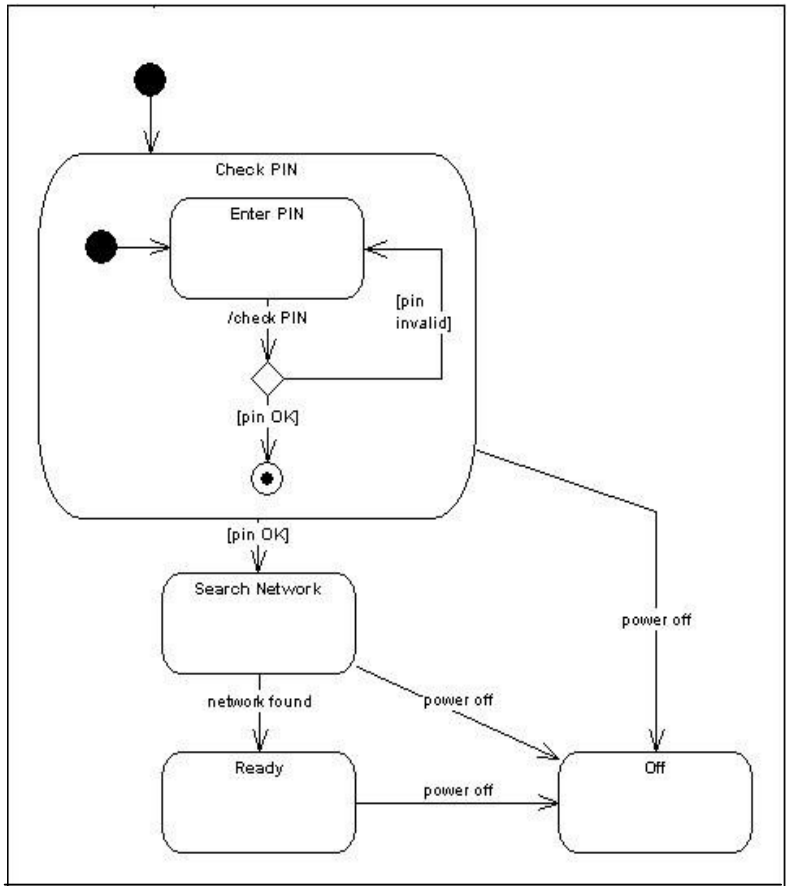
- ✓ **State Actions:** In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transition. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action. It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.



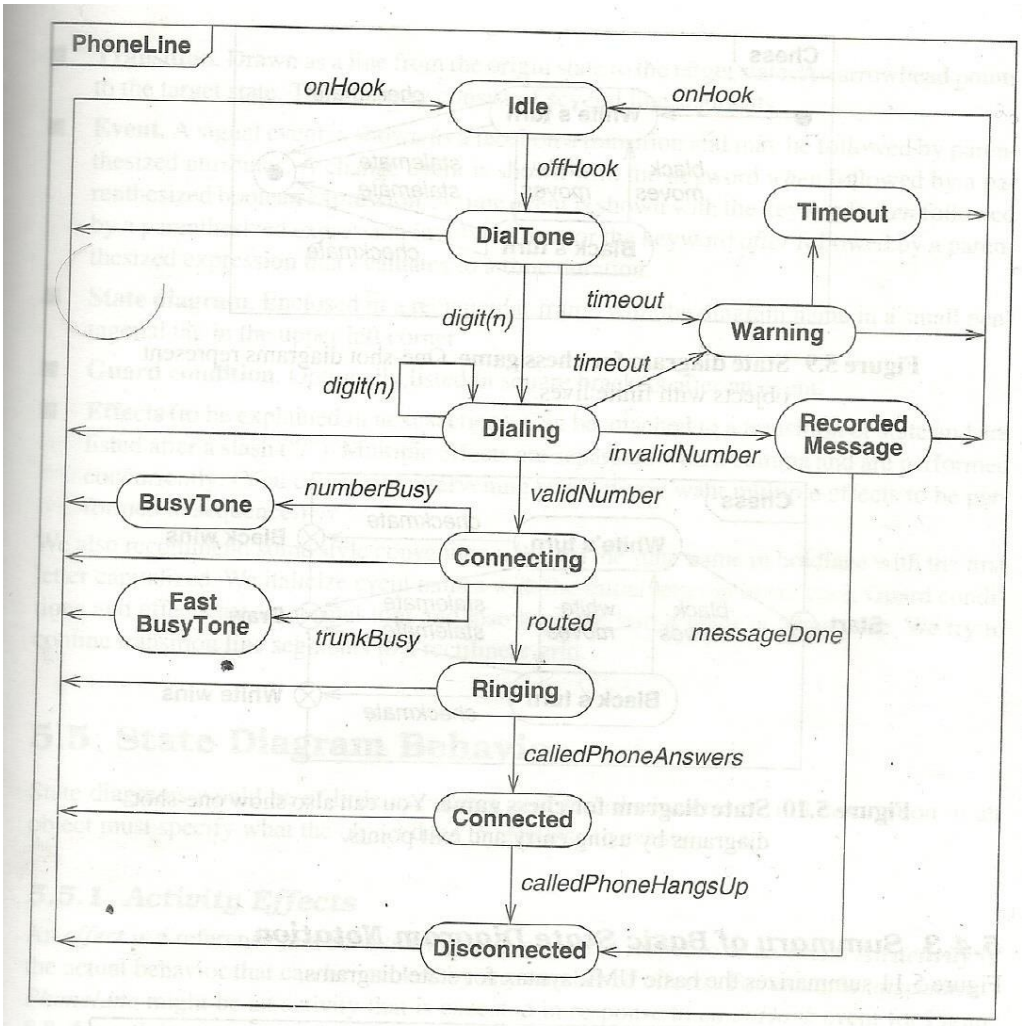
- ✓ Self-Transitions: A state can have a transition that returns to itself, as shown in the Figure below. This is the most useful when an effect is associated with the transition.



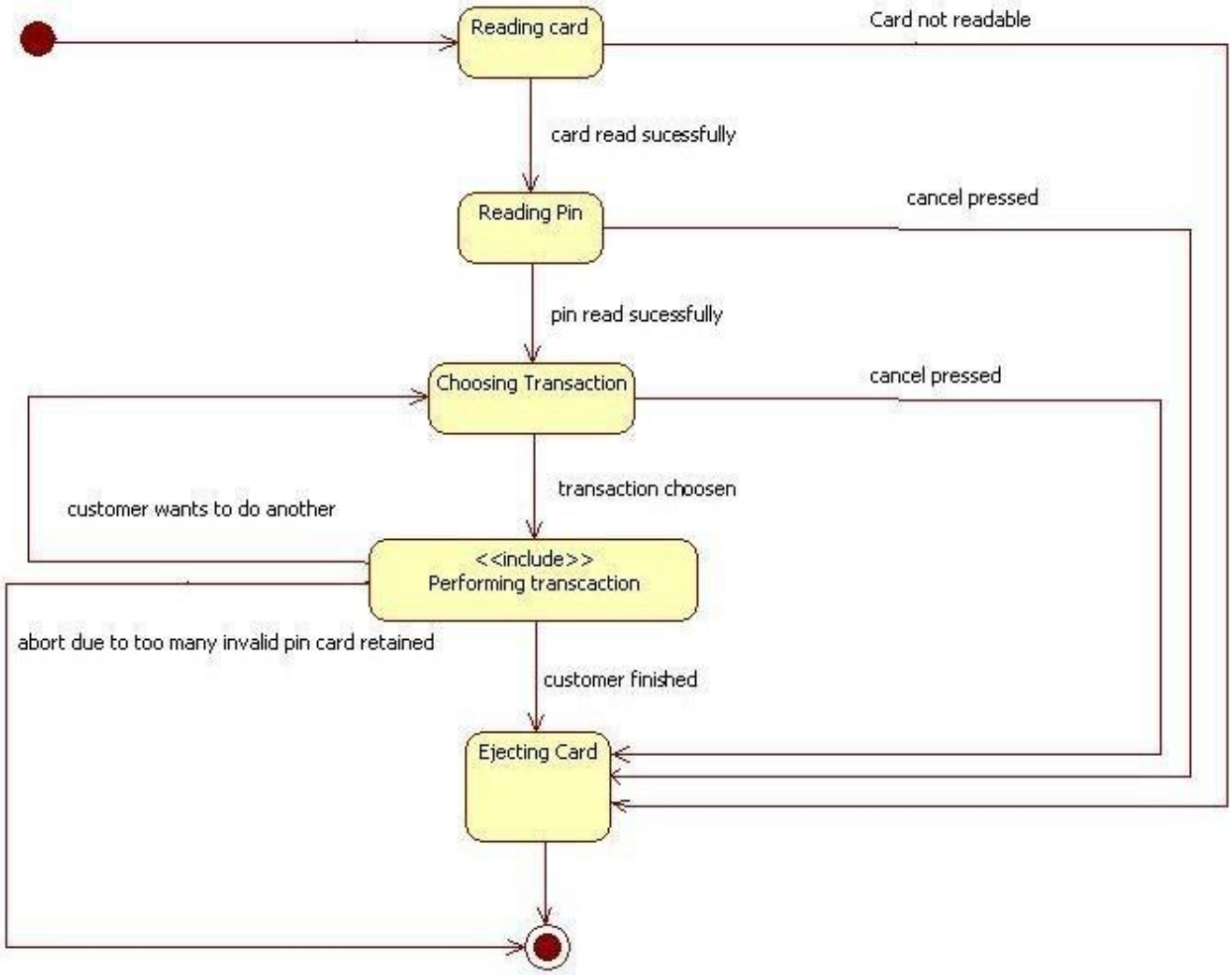
- ✓ Compound States: A sub-state is a state that is nested in another state. A state that has sub-states is called a composite state. A state that has no sub-states is called a simple state. Sub-states may be nested to any level. The notation in the above version indicates that the details of the Check PIN submachine are shown in a separate diagram.


Check PIN
Enter PIN
/check PIN
[pin invalid]
[pin OK]
[pin OK]
Search Network
power off
network found
power off
Ready
power off
Off

STATE DIAGRAM FOR MAKING A PHONE CALL



STATE DIAGRAM FOR ATM TRANSACTION



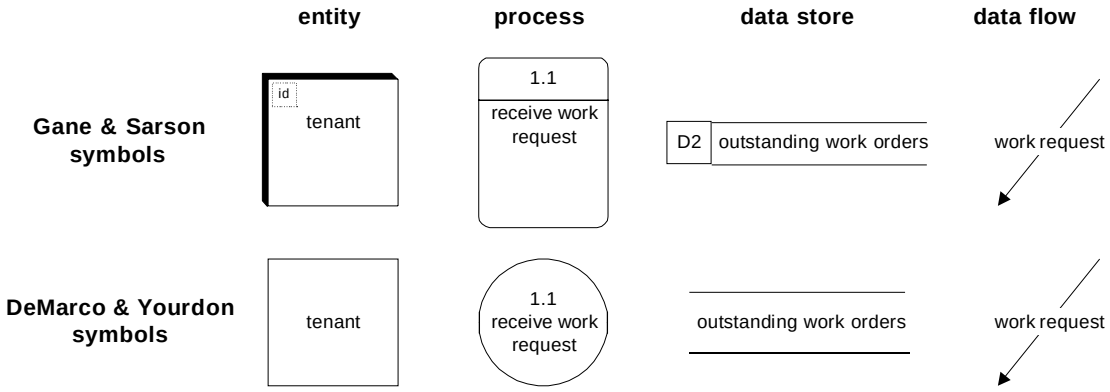
FUNCTIONAL MODELLING

- The functional model describes computations and specifies those aspects of the system concerned with transformations of values - functions, mappings, constraints, and functional dependencies. The functional model captures what the system does, without regard to how or when it is done.
- The functional model is represented graphically with multiple data flow diagrams, which show the flow of values from external inputs, through operations and internal data stores, to external outputs.
- Data flow diagrams show the dependencies between values and the computation of output values from input values and functions. Functions are invoked as actions in the dynamic model and are shown as operations on objects in the object model.

DATA FLOW DIAGRAMS (DFD)

- The DFD is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on that data, and the output generated by the system.
- DFD is used to represent the data transformations. That is it shows how input data is transformed into suitable output data.
- DFD is also termed as Bubble chart though a process is represented using bubble.
- Advantages of DFD are:
    i. DFDs are simple to understand and use
    ii. It uses a very few primitive symbols to represent the functions performed by a system and the data flow among these functions
- There are two different notations used in DFDs.
    i. Gane and Sarson notation
    ii. DeMarco and Yourdon notation

You may use either of these two conventions, just don't mix them.



Entities:

- Those physical entities external to the software system such as people, departments, other companies, other systems or even a logical entity like bank account.
- Entities are called **sources** if they are external to the system and provide data to the system, and **sinks** if they are external to the system and receive information from the system

Processes:

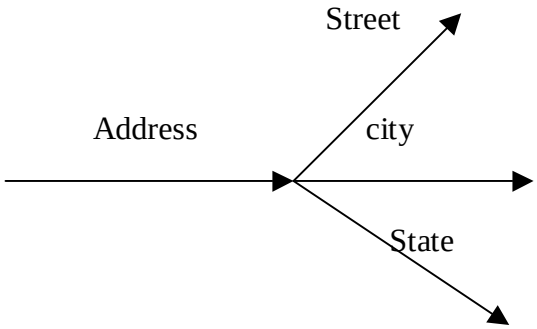
- A function / process is represented using a circle.
- A process must have at least one input and at least one output
- A process should be numbered.
- The name of the process should be specified in the bubble.

Data stores:

- Data store represents a logical file or a data base or any media which stores data. It can be online or "hard copy" .
- Data stores are labelled with a noun (e.g. the label "customer" indicates that information about customers is kept in that data store)
- Data is stored whenever there are more than one process that needs it and these processes don't always run one after the other (if the data is ever needed in the future it must be stored)

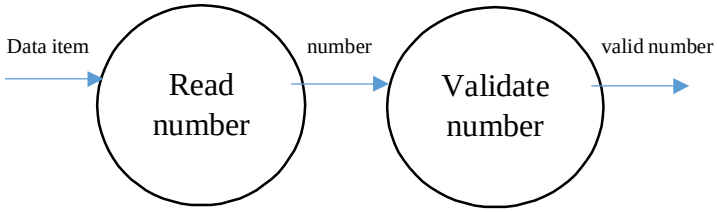
Data flows:

- It must originate from and/or lead to a process (this means that entities and data stores cannot communicate with anything except processes –remember that it takes a process to make the data flow)
- It can go from process to process, but that does imply that no data is stored at that point
- It can have one arrowhead indicating the direction in which the data is flowing
- It can have 2 arrowheads when a process is altering (updating) existing records in a data store
- Elementary data cannot be decomposed into its meaningful constituents. For example, roll no, pin code, and quantity. Aggregate data can be decomposed into its meaningful constituents. For example, name can be decomposed into first name, middle name and last name. Sometimes an aggregate value is split into its constituents, each of which goes to a different process. A fork in the path as shown below is used to do this. Reverse can also be done. That is elementary data coming from different sources can be aggregated. This is done by reversing the arrows in the diagram below.

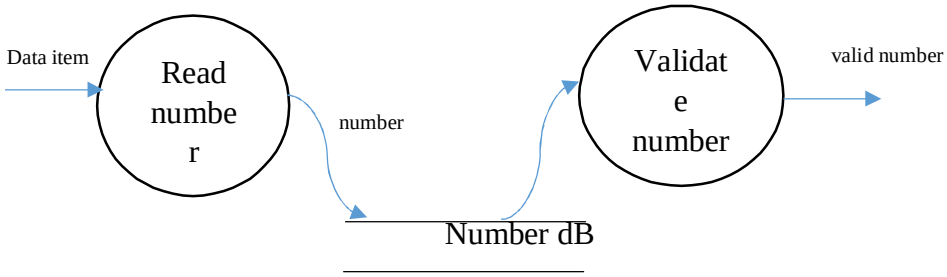


- In De Marco and Yourdon notation an additional symbol is mentioned for representing hardcopy output.

- A data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and lists all data items that appear in a DFD model.
- **Synchronous and Asynchronous operations:**
  - If two processes are directly connected by a dataflow arrow, then they are synchronous. This means that they operate at the same speed.



  - If two processes are connected through a data store, then the speed of operation is independent. They are asynchronous.



❖ **Short comings of DFD / Disadvantages:**

i. DFDs leave ample scope to be imprecise

ii. Control aspects are not defined by a DFD

iii. Though decomposition is possible, same problem has several alternative DFD representations.

CONTEXT DIAGRAM / LEVEL 0 DFD

Context diagram is the highest level of data flow representation of a system. It represents the entire system as a single bubble. The bubble in context diagram is annotated with the name of the software system being developed.

✓ Context diagram establishes the context in which the system operates. That is who the users are, what data do they input to the system and what data do they received from the system.

✓ All entities should be represented in the context diagram. No entities are represented in any other levels of DFD.

DRAWING DFDs

- ✓ Except for the context DFD, each **DFD** represents the breakdown of one process.

  For example, in the hierarchy on the next page, the level 1 DFD that represents the breakdown of process 1.2 will contain processes 1.2.1, 1.2.2 and 1.2.3. But it will not contain 1.2, nor any other process.
- ✓ **Context level** diagrams show all external entities. They do not show any data stores. The context diagram always has only one process labelled 0.
- ✓ When you draw a **level 0** diagram, follow these rules:
  - include all entities in the context diagram
  - show any data store that are shared by the processes in the level 0 diagram
- ✓ When you draw a **level 1 or 2 etc**. diagram, follow these rules:
    - o include all entities and data stores that are directly connected by data flow to the one process you are breaking down
    - o show all other data stores that are shared by the processes in this breakdown (these data stores are "internal" to this diagram and will not appear in higher level diagrams, but will appear in lower level diagrams)

  That last statement is often confusing. Here is another explanation using the hierarchy on If a data store is used only by processes 3.2.1 and 3.2.3, then it will appear only in the level 2 diagram that includes processes 3.2.1, 3.2.2 and 3.2.3. It will not appear in the diagram that shows processes 3.1 and 3.2 because it is internal to process 3.2.
- ✓ A DFD that contains processes that are not further broken down is called a **primitive** DFD.

GUIDELINES FOR DRAWING DFD

- Naming conventions:
  - o Processes: strong verbs
  - o Data flows: nouns
  - o data stores: nouns
  - o external entities: nouns
- No more than 7 - 9 processes in each DFD.
- Dataflow must begin, end, or both begin & end with a process.
- Dataflow must not be split.
- A process is not an analogy of a decision in a systems or programming flowchart. Hence, a dataflow should not be a control signal. Control signals are modelled separately as control flows.
- Loops are not allowed.
- A dataflow cannot be an input signal. If such a signal is necessary, then it must be a part of the description of the process, and such process must be so labelled. Input signals as well as their effect on the behaviour of the system are incorporated in the behavioural model (say, state transition graphs) of the information system.
- Decisions and iterative controls are part of process description rather than dataflow.
- If an external entity appears more than once on the same DFD, then a diagonal line is added to the north-west corner of the rectangle (representing such entity).
- Updates to data store are represented in the textbook as double-ended arrows. This is not, however, a universal convention. I would rather you did not use this convention since it can

be confusing. Writing to a data store implies that you have read such data store (you cannot

write without reading). Therefore, data store updates should be denoted by a single-ended arrow from the updating process to the updated data store.

- Data flows that carry a whole record between a data store and a process is not labelled in the textbook since there is no ambiguity. This is also not a universal convention. I would rather you labelled such data flows explicitly.
- **Conservation Principles:**
  Data stores & Data flows: Data stores cannot create (or destroy) any data. What comes out of a data store therefore must first have got into a data store through a process.

  Processes: Processes cannot create data out of thin air. Processes can only manipulate data they have received from data flows. Data outflows from processes therefore must be derivable from the data inflows into such processes.
- **Levelling Conventions:**

  - Numbering: The system under study in the context diagram is given number `0'. The processes in the top level DFD are labelled consecutively by natural numbers beginning with 1. (Also you can number the processes in LEVEL 1 as 0.1, 0.2, 0.3, 0.4… 0.n). When a process is exploded in a lower level DFD, the processes in such lower level DFD are consecutively numbered following the label of such parent process ending with a period or full-stop (for example 1.2, 1.2.3, etc. or you can use 0.1.1, 0.1.2…etc).

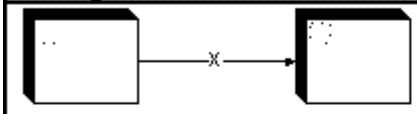
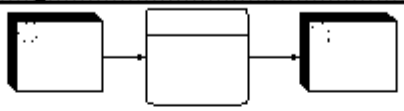
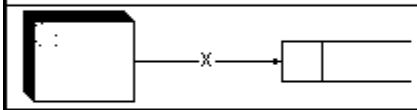
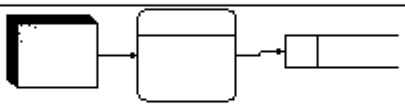
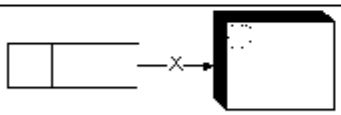
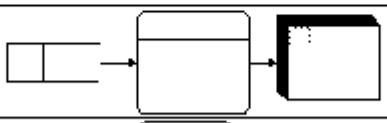
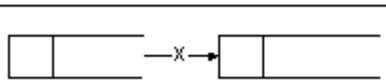
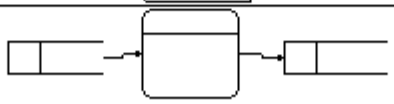
    Remember: Don't mix up the numbering conventions. If you label the level 1 processes as 0.1, 0.2, 0.3…0.n then you have to number the level 2 processes as 0.1.1, 0.1.2, 0.1.3 etc.

    Else if you label the level 1 processes as 1,2,3…n, then you have to number the level 2 processes as 1.1, 1.2, 1.3 etc.

  - Balancing: The set of DFDs pertaining to a system must be balanced in the sense that corresponding to each dataflow beginning or ending at a process there should be an identical dataflow in the exploded DFD.
  - Data stores: Data stores may be local to a specific level in the set of DFDs. A data store is used only if it is referenced by more than one process.
  - External entities: Lower level DFDs cannot introduce new external entities. The context diagram must therefore show all external entities with which the system under study interacts. In order not to clutter higher level DFDs, detailed interactions of processes with external entities are often shown in lower level DFDs but not in the higher level ones.

COMMONLY MADE MISTAKES IN DFDS

Table 2: Common data flow diagramming mistakes

Wrong	Right	Description
		A source or a sink cannot provide data to another source or sink without some processing occurring.
		Data cannot move directly from a source to a data store without being processed.
		Data cannot move directly from a data store to a sink without being processed.
		Data cannot move directly from one data store to another without being processed.

Source: Adapted from Figure 9.9, p. 360 in Whitten, J. L.; Bentley, L. D.; Barlow, V. M. (1994). Systems Analysis and Design Methods (Third Edition). Burr Ridge, IL: Irwin.

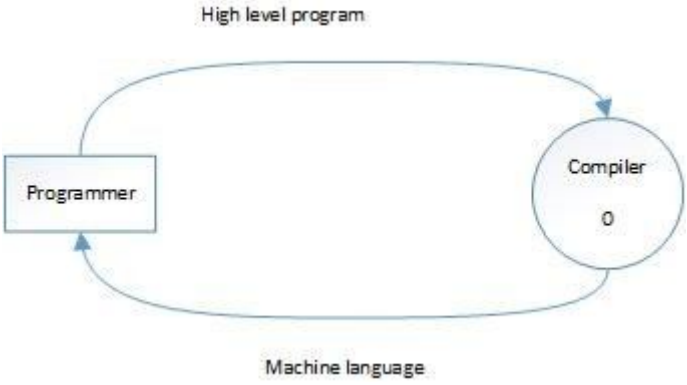
Diagramming mistakes: Black holes, grey holes, and miracles

A second class of DFD mistakes arise when the outputs from one processing step do not match its inputs. It is not hard to list situations in which this might occur:

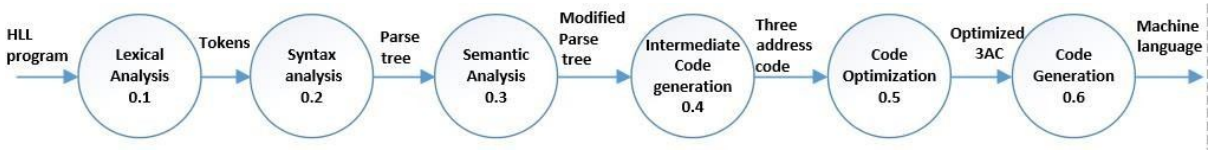
- A processing step may have input flows but no output flows. This situation is sometimes called a *black hole* .
- A processing step may have output flows but now input flows. This situation is sometimes called a *miracle*.
- A processing step may have outputs that are greater than the sum of its inputs - e.g., its inputs could not produce the output shown. This situation is sometimes referred to as a *grey hole*.

DFD OF COMPILER

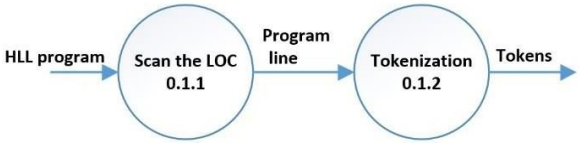
LEVEL 0 DFD: CONTEXT DIAGRAM OF COMPILER



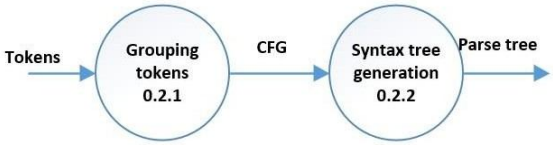
LEVEL 1 DFD: COMPILER



LEVEL 2.1 DFD: LEXICAL ANALYSIS



LEVEL 2.2 DFD: SYNTAX ANALYSIS



4. SYSTEM DESIGN

- System design is the first stage in which the basic approach to solving the problem is selected.
- During system design, the overall structure and style are decided.
- The system architecture is the overall organization of the system into components called subsystems.

STEPS IN SYSTEM DESIGN

1. Breaking a system into subsystems.
2. Identifying concurrency
3. Allocating subsystems to processors and tasks.
4. Management of data stores
5. Handling global resources
6. Choosing software control implementation
7. Handling boundary conditions
8. Setting trade-off priorities
9. Selecting the suitable architectural frame works.

Organizing a system into Subsystems

- Divide the system into a smaller number of components. Each major component of a system is called a subsystem.
- Each subsystem encompasses aspects of the system that share some common property – similar functionality, the same physical location or execution on the same kind of hardware.
- Each subsystem has a well -defined interface to the rest of the system.
- Each subsystem can be again decomposed into modules.
- The decomposition of systems into subsystems may be organized as a sequence of horizontal layers or vertical partitions.
    - **Layers:** A layered system is an ordered set of virtual worlds, each built in terms of the ones below it and providing the implementation basis for the ones above it.
    - Layered architecture comes in closed or open.
    - In a closed architecture, each layer is built only in terms of the immediate lower layer. This reduces dependencies between layers and allows changes to be made most easily. In an open architecture, a layer can use the features of any lower layer to any depth. This reduces the need to redefine operations at each level.
    - In a layered architecture model, classes within each subsystem layer provide services to the layer above it. Ideally, this knowledge is one-way: each layer knows

about the layer below it, but the converse is not true. An example of a layered

system architecture is the ISO Reference Model for Open Systems Interconnection (OSI) as shown in Figure below.

- Example of Subsystem Layers is (ISO Reference Model for OSI)

Application
Presentation
Session
Transport
Network
Data Link
Physical

Partitions: Partitions vertically divide a system into several independent or weakly subsystems, each providing one kind of service.

- One difference between layers and partitions is that layers vary in their level of abstraction, but partitions merely divide a system into pieces, all of which have a similar level of abstraction.
- Layers can be partitioned and partitions can be layered. Most large systems require a mixture of layers and partitions.

❖ Relationships between subsystems are: there are two types of relationships between subsystems: a) Client-Server and b) Peer-to-peer

a) Client–Server relationship: In client-server relationship client calls on the server for performing certain task and server replies back with the result. The client–server characteristic describes the relationship of cooperating programs in an application. The server component provides a function or service to one or many clients, which initiate requests for such services.

b) Peer-to-peer relationship: In peer-to-peer relationship, each subsystem may call on the others. The communication in this case can be much complex because individual subsystems may

not be aware about each other. Designing such type of systems may lead to subtle design errors.

Identifying Concurrency

- Concurrency (or distribution) is an important issue during design process as it may affect the design of classes and their interfaces. Concurrency can be very important for improving the efficiency of a system.
- In the analysis model, as the real world and in hardware, all objects are concurrent. In an implementation, not all software objects are concurrent, because one processor may support many objects.
- One important goal of the system design is to identify the objects that must be active concurrently and the objects that have mutually exclusive activity
- For concurrent applications, such as distributed and real-time applications, the following activities are performed:

  - **Identify inherent concurrency**: - Two objects are inherently concurrent if they can receive events at the same time without interacting. Inherently concurrent subsystems need not be implemented as separate hardware units.
  - **Define concurrent Tasks:** Many objects in a system are dependent on each other. By examining the state diagrams of individual objects and exchange of events among them, many objects can be folded in to a single thread of model. A thread of control is a path through a set of state diagrams on which only a single object at a time is active.
  - **Make decisions about subsystem structure and interfaces.** Develop the overall software architecture. Structure the application into subsystems.
  - **Make decisions about how to structure the distributed application into distributed subsystems, in which subsystems are designed as configurable components**. Design the distributed software architecture by decomposing the system into distributed subsystems and defining the message communication interfaces between the subsystems.
  - **Make decisions about the characteristics of objects, in particular, whether they are active or passive.** For each subsystem, structure the system into concurrent tasks (active objects). During task structuring, tasks are structured using the task structuring criteria, and task interfaces are defined. Make decisions about the characteristics of messages, in particular, whether they are asynchronous or synchronous (with or without reply).
  - **Make decisions about class interfaces.** For each subsystem, design the information hiding classes (passive classes). Design the operations of each class and the parameters of each operation. Use inheritance to develop class hierarchies.

- Develop the detailed software design, addressing detailed issues concerning task synchronization and communication, and the internal design of concurrent tasks.
- **For real-time applications, analyse the performance of the design**. Apply real-time scheduling to determine if the concurrent real-time design will meet performance goals. If not, investigate alternative software architectures.

Allocation of Subsystems

- The designer must allocate each concurrent subsystem to a hardware unit, either a general-purpose processor or specialized functional unit. The system designer must do the following:

  - Estimate performance needs and the resources needed to satisfy them.
  - Choose hardware or software implementation for subsystems.
  - Allocate software subsystems to processors to satisfy performance needs and minimize inter processor communication
  - Determine the connectivity of the physical units that implement the subsystems.
  - Consider the connection between nodes and communication protocols to be used.
  - Consider the need for redundant processing.
  - Identify any interface implied by deployment.
  - Determining physical Connectivity by considering connection topology, repeated units, communications etc.

Management of Data Storage

- System designer must decide from among several alternatives for data storage that can be used separately or in combination of data structures, files and databases. This involves identifying the complexity of the data, the size of the data, the type of access to data (single user or multiple user), access times and portability.
- Different kinds of data stores provide trade-offs among cost, access time, capacity and reliability.
- Files provide cheap, simple and permanent storage and are easy to work with. However, file on one system may not be useful when transported to another system because of varying file implementations over different hardware types. Files may be used in random access mode or sequential access mode. Sequential file format is mostly a standard format and is easy to handle. Whereas, the commands and storage formats for random access files and index files vary in their formats. The kind of data that belongs to files can be characterized as follows:

- Data with high volumes and low information density (such as archival files or historical data)
- Modest quantities of data with simple structure.
- Data that are accessed sequentially.
- Data that can be fully read into the memory.

❖ Another alternative to store data is to use database management systems (DBMSs). There are various kinds of DBMSs like relational, object oriented, network and hierarchical etc. Databases make applications easier to port to different hardware and operating system platforms. One disadvantage of DBMSs is their complex interface. The kind of data that belongs to a database can be characterized as follows:

- Data to be stored exists in large quantity.
- Data that is to be kept in store for a very longer period of time.
- Data that must be secured against unauthorized and malicious access.
- Data that must be accessed by multiple application programs.
- Data that require updates at fine levels of detail by multiple users

Handling Global Resources

❖ The system designer must identify global resources and determine mechanisms for controlling access to them. There are several kinds of global resources:

- **Physical system**: Example includes processors, tape drives and communication channels.
- **Space**: Example includes keyboard, buttons on a mouse, display screen
- **Logical names**: Example includes object IDs, filenames, and class names.
- **Access to shared data:** Example includes Databases

❖ Physical resource such as processors, tape drives etc. can control their own access by establishing a protocol for obtaining access.

❖ For a logical resource like Object ID or a database, there arises a need to provide access in a shared environment without any conflicts. One strategy to avoid conflict may be to employ a guardian object which controls access to all other resources.

Choosing a Software Control Strategy

❖ It is best to choose a single control style for the whole system. There are two kinds of control flows in a software system: External control and internal control.

- External control concerns the flow of externally visible events among the objects in the system. There are three kinds of external events: procedural-driven sequential, event-driven sequential and concurrent.

  Procedural-driven Control: In a procedure-driven system, the control lies within the program code. Procedures request external input and then wait for it, when input arrives, control resumes within the procedure that made the call.

  Event-driven Control: In the sequential model, the control resides within a dispatcher or monitor that the language, subsystem or operating system provides. In event-driven, the developers attach application procedures to events and the dispatcher calls the procedures when the corresponding events occur. Usually event driven systems are used for external control in preference to procedure driven systems, because the mapping from events to program constructs is simpler and more powerful. Event driven systems are more modular and can handle error conditions better than procedure-driven systems.

  Concurrent system Control: Here control resides concurrently in several independent objects, each as a separate task. A task can wait for input, but other tasks continue execution. The operating system keeps track of the raised events while a task is being performed so that events are not lost. Scheduling conflicts among tasks are also resolved by the operating system.

- **Internal control** refers to the flow of control within a process. It exists only in the implementation and therefore is neither inherently concurrent nor sequential.

Handling boundary Conditions

- Although most of the system design concerns steady-state behaviour system designer must consider boundary conditions as well and address issues like initialization, termination and failure (the unplanned termination of the system).

  Initialization: It refers to initialization of constant data, parameters, global variables, tasks, guardian objects, and classes as per their hierarchy. Initialization of a system containing concurrent tasks must be done in a manner so that tasks can be started without prolonged delays. There is quite possibility that one object has been initialized at an early stage and the other object on which it is dependent is not initialized even after considerable time.

  Termination: Termination requires that objects must release the reserved resources.

In case of concurrent system, a task must intimate other tasks about its termination.

Failure: Failure is the unplanned termination of the system, which can occur due to system fault or due to user errors or due to exhaustion of system resources, or from external breakdown or bugs from external system. The good design must not affect remaining environment in case of any failure and must provide mechanism for recording details of system activities and error logs.

Setting trade-off Priorities

- The system designer must set priorities that will be used to guide trade-offs for the rest of the design. For example system can be made faster using extra memory.
- Design trade-offs involve not only the software itself but also the process of developing it. System designer must determine the relative importance of the various criteria as a guide to making design trade-offs. Design trade-offs affect entire character of the system.
- Setting trade-offs priorities is at best vague. Priorities are generally specified as a statement of design philosophy.

Choose Common Architectural Styles

- Several prototypical architectural styles are common in existing systems. Each of these is well suited to a certain kind of system.
- Architectural styles include Pipe and Filter, Repository architecture, Layered systems, Client –Server, Peer- Peer, Publisher – Subscriber etc.

5. OBJECT DESIGN

- Object design phase determines the full definitions of the classes and associations used in the implementation, interfaces and algorithms of the methods used to implement operations.
- Adds internal objects for implementation and optimizes data structures and algorithms.
- Object oriented design is primarily a process of refinement or adding details.

STEPS IN OBJECT DESIGN

1. Combine the three models to obtain operations on classes.
2. Design algorithms to implement operations
3. Optimize access paths to data / Design optimization
4. Implementation of control for external interactions
5. Adjust class structure to increase inheritance
6. Design of associations
7. Determine object representation
8. Packing classes and associations into modules.

Combine the three models to obtain operations on classes

- The object, dynamic, functional models obtained after analysis is combined together to define operations.
- The designer convert the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model.
- Process of mapping the logical structure of the analysis model into a physical organization of a program.
- Algorithm implementing an operation depends on the state of the object.
- The network of processes within DFD represents the body of an operation.

Design algorithms to implement operations

- Each operation specified in the functional model must be formulated as an algorithm.
- Algorithm shows how an operation is done.
- Algorithm may be subdivided into calls on simpler operations.
- Algorithm designer must
  - ✓ Choose algorithms that minimize the cost of implementing operations.
  - ✓ Select data structures appropriate to the algorithms.
  - ✓ Define new internal classes and operations as necessary
  - ✓ Assign responsibility for operations to appropriate classes.

Optimize access paths to data / Design optimization

- Basic design model uses analysis model which captures the logical information about the system, while design model must add details to support efficient information access.
- Inefficient but semantically correct analysis model can be optimized to make the implementation more efficient.
- During design optimization, designer must
  - ✓ Add redundant associations to minimize access cost and maximize convenience.
  - ✓ Rearrange the computation for greater efficiency.
  - ✓ Save derived attributes to avoid re-computation of complicated expression

Implementation of control for external interactions

- Designer must refine the strategy for implementing the state – event models present in the dynamic model.
- Basic approaches to implement dynamic model are
  - ✓ Using a location within the program to hold state.
  - ✓ Direct implementation of a state – machine mechanism.
  - ✓ Control as concurrent tasks.

Adjust class structure to increase inheritance

- The designer should rearrange and adjust classes and operations to increase inheritance
- The designer should abstract common behaviour out of group of classes
- The designer should use delegation to share behaviour when inheritance is semantically invalid.

Design of associations

- Associations are the "glue" of object model, providing access paths between objects.
- Designer must formulate a strategy for implementing the associations in the object model.
- Either choose a global strategy for implementing all associations uniformly or select a particular technique for each association based on the way it is used in the application.
- Associations may be either one-way associations or two –way associations.

Determine object representation

- Designer must choose when to use primitive types in representing objects and when to combine groups of related objects.
- Classes can be defined in terms of other classes, but eventually everything must be implemented in terms of built-in primitive datatypes, such as integers, strings and enumerated types.
- Designer must often choose whether to combine groups of related objects.

Packing classes and associations into modules.

- Object oriented languages have various degrees of packaging.
- In large project, careful partitioning of an implementation into packages is important to permit different persons to cooperatively work on a program.
- Packaging involves the following issues:
  - ✓ Hiding internal information from outside view.
  - ✓ Coherence of entities.
  - ✓ Constructing physical modules

6. UNIFIED MODELLING LANGUAGE (UML)

- UML is a standard language for specifying, visualizing, constructing, and documenting the artefacts of software systems.
- UML was created by Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.
- UML is a modelling language used to model software and non- software systems.
- Although UML is used for non- software systems the emphasis is on modelling object oriented software applications. Most of the UML diagrams discussed so far are used to model different aspects like static, dynamic etc.

BUILDING BLOCKS OF UML

➢ The building blocks of UML can be defined as:

- Things
- Relationships
- Diagrams

THINGS

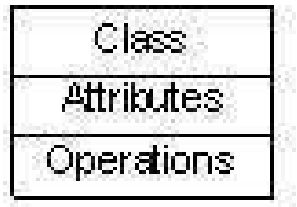
Things are the most important building blocks of UML. Things can be:

- Structural
- Behavioral
- Grouping
- Annotational

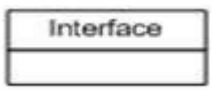
❖ **Structural things:**

The **Structural things** define the static part of the model. They represent physical and conceptual elements. Following are the brief descriptions of the structural things.

Class: Class represents set of objects having similar responsibilities.



Interface: Interface defines a set of operations which specify the responsibility of a class.

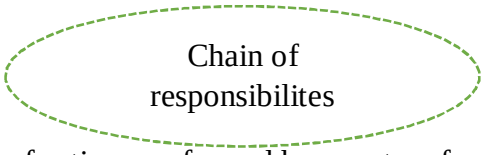


or

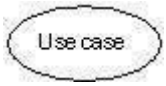
IInterface

Collaboration: Collaboration defines interaction between elements.

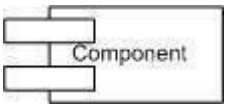
or



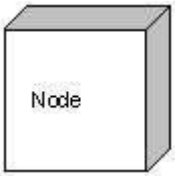
Use case: Use case represents a set of actions performed by a system for a specific goal.



Component: Component describes any physical or replaceable part of a system.



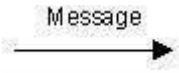
Node: A node can be defined as a physical element that exists at run time which possess a limited memory and processing capability



❖ **Behavioral things:**

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:

Interaction: Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



State machine: State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

- **Grouping things:**

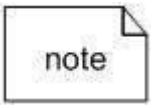
  Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available:

  Package: Package is the only one grouping thing available for gathering structural and behavioural things.

- **Annotational things:**

  Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** is the only one Annotational thing available.

  Note:
  A note is used to render comments, constraints etc of an UML element.

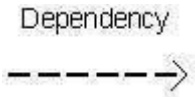


RELATIONSHIPS

Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

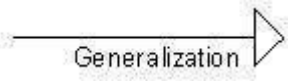
There are four kinds of relationships available.

Dependency: Dependency is a relationship between two things in which change in one element also affects the other one.

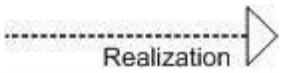


Association: Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.

Generalization: Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.



Realization: Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.


Realization

UML DIAGRAMS

- ❖ UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.
- ❖ The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it a complete one.
- ❖ Various diagrams in UML are listed below:

1. Class diagram – A class diagram shows a set of classes, interfaces and collaborations and their relationships.
2. Object diagram – An object diagram shows a set of objects and their relationships.
3. Use case diagram – A use-case diagram shows a set of use cases and actors and their relationships
4. Sequence diagram – A sequence diagram is an interaction diagram that emphasizes the time – ordering of messages.
5. Collaboration diagram – A collaboration diagram is an interaction diagram that emphasizes the structural organisation of the objects that send and receive messages.
6. Activity diagram – An activity diagram is a special kind of state-chart diagram that shows the flow from activity to activity within a system.
7. State-chart diagram – A state-chart diagram shows a state machine, consisting of states, transitions, events and activities.
8. Deployment diagram – A deployment diagram shows the configuration of run time processing nodes and the components that live on them.
9. Component diagram – A component diagram shows the organisation and dependencies among a set of components.

COMMON MECHANISMS IN UML

- ✓ The UML is made simpler by the presence of four common mechanisms that apply throughout the language: specifications, adornments, common divisions, and extensibility mechanisms.

Specifications

- ✓ By using a specification, we can easily specify something in a bit more detail so that the role and meaning of the term being specified is presented to us in a more clear and concise manner.
- ✓ For example, we can give a class a rich specification by defining a full set of attributes, operations, full signatures, and behaviours. The developer will then have a clearer notion of what the capabilities and limitations of that class are.
- ✓ Specifications can be included in the class, or specified separately.

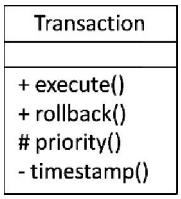
Common Divisions

- ✓ Common divisions are used in order to distinguish between two things that might appear to be quite similar, or closely related to one another. There exist two main common divisions: abstraction vs. manifestation and interface vs. implementation.

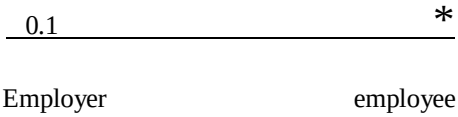
- ✓ The distinction between a class and an object is an example of common division in UML, where the class is an abstraction and the object is a clear manifestation of that class. Most UML building blocks have this kind of class/object distinction, e.g. use case, use case instance etc.
- ✓ In the second common division – interface vs. implementation – we say that an interface declares some kind of contract, or agreement, whereas an implementation represents one concrete realisation of that contract. The implementation is then responsible for carrying out the interface.

Adornments

- ✓ Adornments are textual or graphical items, which can be added to the basic notation of a UML building block in order to visualise some details from that element's specification.
- ✓ For example an abstract class consist of a public, private and protected operations, then it can be represented as follows: + denotes public, # denotes protected and – denotes private.



- ✓ For example, let us consider association, which in its most simple notation consists of one single line. Now, this can be adorned with some additional details, such as the role and the multiplicity of each end as shown below.



Extensibility Mechanisms

- ➢ The extensibility mechanisms allow you to customize and extend the UML by adding new building blocks, creating new properties, and specifying new semantics in order to make the language suitable for your specific problem domain. There are three common extensibility mechanisms that are defined by the UML: stereotypes, tagged values, and constraints.

Stereotypes

- ✓ Stereotypes allow you to extend the vocabulary of the UML so that you can create new model elements, derived from existing ones, but that have specific properties that are suitable for your problem domain.
- ✓ They are used for classifying or marking the UML building blocks in order to introduce new building blocks that speak the language of your domain and that look like primitive, or basic, model elements.
- ✓ For example, when modelling a network you might need to have symbols for representing routers and hubs. By using stereotyped nodes you can make these things appear as primitive building blocks.

- ✓ As another example, let us consider exception classes in Java or C++, which you might sometimes have to model. Ideally you would only want to allow them to be thrown and caught, nothing else. Now, by marking them with a suitable stereotype you can make these classes into first class citizens in your model; in other words, you make them appear as basic building blocks.
- ✓ Stereotypes also allow you to introduce new graphical symbols for providing visual cues to the models that speak the vocabulary of your specific domain (see fig 4).
- ✓ Graphically, a stereotype is rendered as a name enclosed by guillemots and placed above the name of another element (see fig 3). Alternatively, you can render the stereotyped element by using a new icon associated with that stereotype (see fig 4).

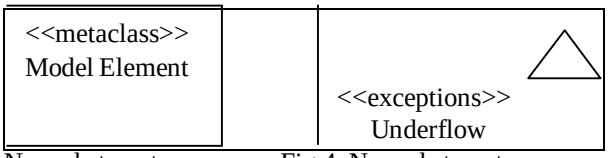


Fig 3. Named stereotype

Fig 4. Named stereotype with icon

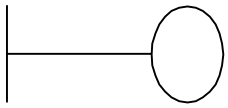


Fig 5. Stereotyped element as icon

Tagged Values

- ✓ Tagged values are properties for specifying keyword-value pairs of model elements, where the keywords are attributes. They allow you to extend the properties of a UML building block so that you create new information in the specification of that element.
- ✓ Tagged values can be defined for existing model elements, or for individual stereotypes, so that everything with that stereotype has that tagged value.
- ✓ It is important to mention that a tagged value is not equal to a class attribute. Instead, you can regard a tagged value as being a metadata, since its value applies to the element itself and not to its instances.
- ✓ One of the most common uses of a tagged value is to specify properties that are relevant to code generation or configuration management. So, for example, you can make use of a tagged value in order to specify the programming language to which you map a particular class, or you can use it to denote the author and the version of a component.
- ✓ As another example of where tagged values can be useful, consider the release team of a project, which is responsible for assembling, testing, and deploying releases. In such a case it might be feasible to keep track of the version number and test results for each main subsystem, and so one way of adding this information to the models is to use tagged values.
- ✓ Graphically, a tagged value is rendered as a string enclosed by brackets, which is placed below the name of another model element. The string consists of a name (the tag), a separator (the symbol =), and a value (of the tag) (see figure below).

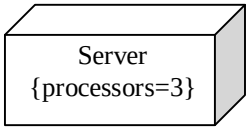


Fig: Tagged Value

Constraints

- ✓ Constraints are properties for specifying semantics and/or conditions that must be held true at all times for the elements of a model. They allow you to extend the semantics of a UML building block by adding new rules, or modifying existing ones.
- ✓ For example, when modelling hard real time systems it could be useful to adorn the models with some additional information, such as time budgets and deadlines. By making use of constraints these timing requirements can easily be captured.
- ✓ Graphically, a constraint is rendered as a string enclosed by brackets, which is placed near the associated element(s), or connected to the element(s) by dependency relationships. This notation can also be used to adorn a model element's basic notation, in order to visualise parts of an element's specification that have no graphical cue.

7. CLASS DIAGRAMS

- The class diagram is a static diagram. It represents the static view of an application.
- Class diagram is not only used for visualizing, describing and documenting different aspects of a system but also for constructing executable code of the software application.
- The class diagram describes the attributes and operations of a class and also the constraints imposed on the system.
- The class diagrams are widely used in the modelling of object oriented systems because they are the only UML diagrams which can be mapped directly with object oriented languages.
- The class diagram shows a collection of classes, interfaces, associations, collaborations and constraints. It is also known as a *structural diagram*.

PURPOSE

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

APPLICATION OF CLASS DIAGRAMS

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

STEPS FOR DRAWING CLASS DIAGRAMS

- Identify the various objects in the system.
- Group the objects with similar properties and semantics into a class.
- Identify the responsibilities, operations and attributes associated with each class.
- Identify the relationship between classes.
- Adjust class structures to improve inheritance.
- Model the class diagram suitably by showing classes and their interrelationships.

CLASS DIAGRAM CONCEPTS

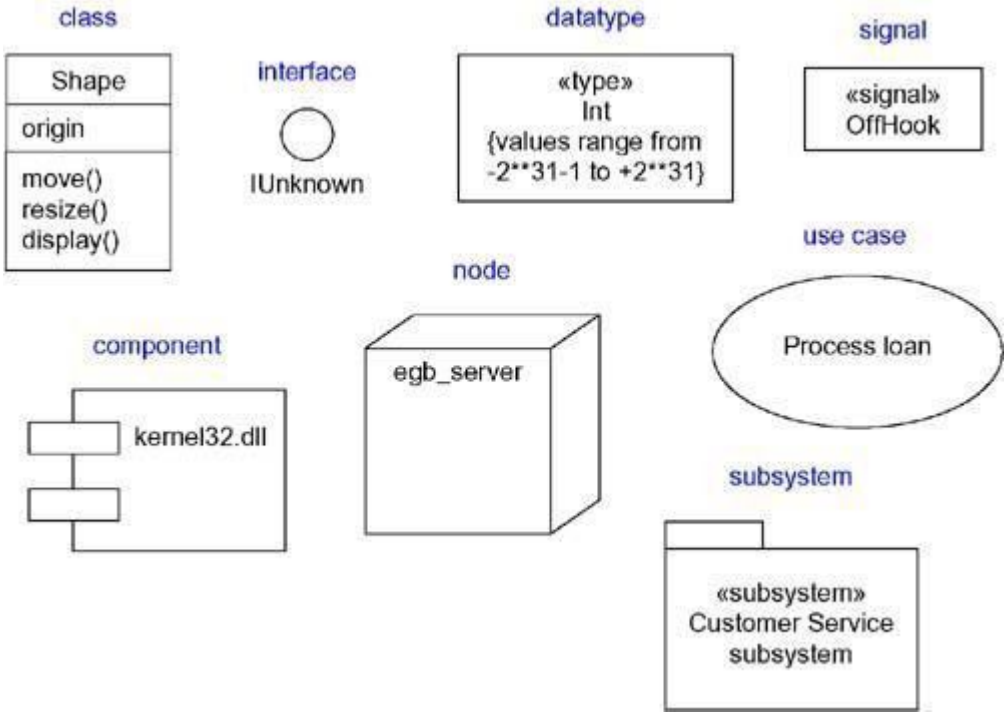
- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified.
- For each class minimum number of properties should be specified. Because unnecessary properties will make the diagram complicated.
- Use notes whenever required to describe some aspect of the diagram. Because at the end of the drawing it should be understandable to the developer/coder.

ADVANCED CLASSES

- A **classifier** is a mechanism that has structural features (in the form of attributes), as well as behavioural features (in the form of operations).
- Classifiers include classes, interfaces, datatypes, signals, components, nodes, use cases, and subsystems. Those modelling elements that can have instances are called classifiers.
- Every instance of a given classifier shares the same features. The most important kind of classifier in UML is class.

• Interface	A collection of operations that are used to specify a service of a class or a component
• Datatype	A type whose values have no identity, including primitive built-in types (such as numbers and strings), as well as enumeration types (such as Boolean)
• Signal	The specification of an asynchronous stimulus communicated between instances
• Component	A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces
• Node	A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
• Use case	A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
• Subsystem	A grouping of elements of which some constitute a specification of the behavior offered by the other contained elements

- Classifiers represented graphically are shown below.

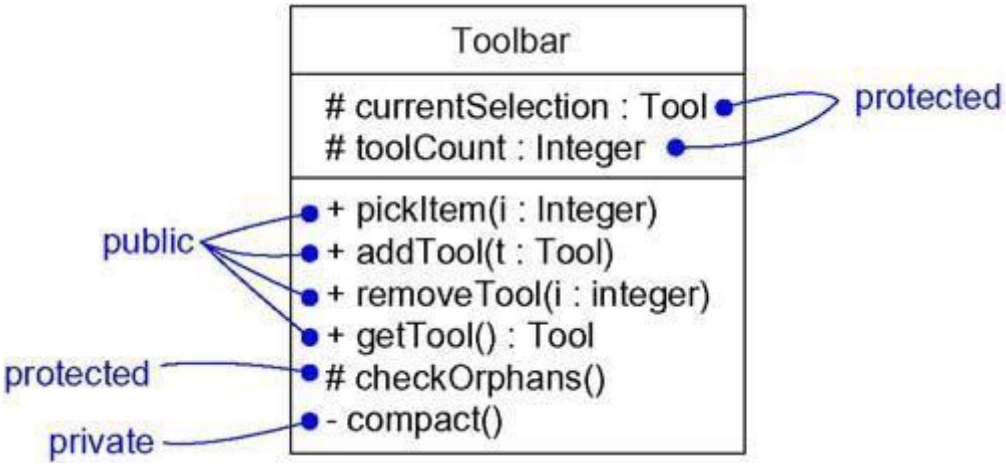


- **Visibility of a Classifier**

Visibility indicates whether the attributes and operations of a classifier can be used by any other classifiers. There are three levels of visibility in UML public, protected and private.

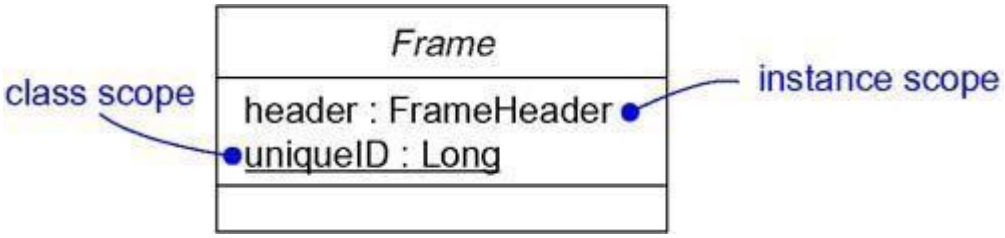
1. public	Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +
2. protected	Any descendant of the classifier can use the feature; specified by prepending the symbol #
3. private	Only the classifier itself can use the feature; specified by prepending the symbol

Figure below indicates the various visibilities of class Toolbar

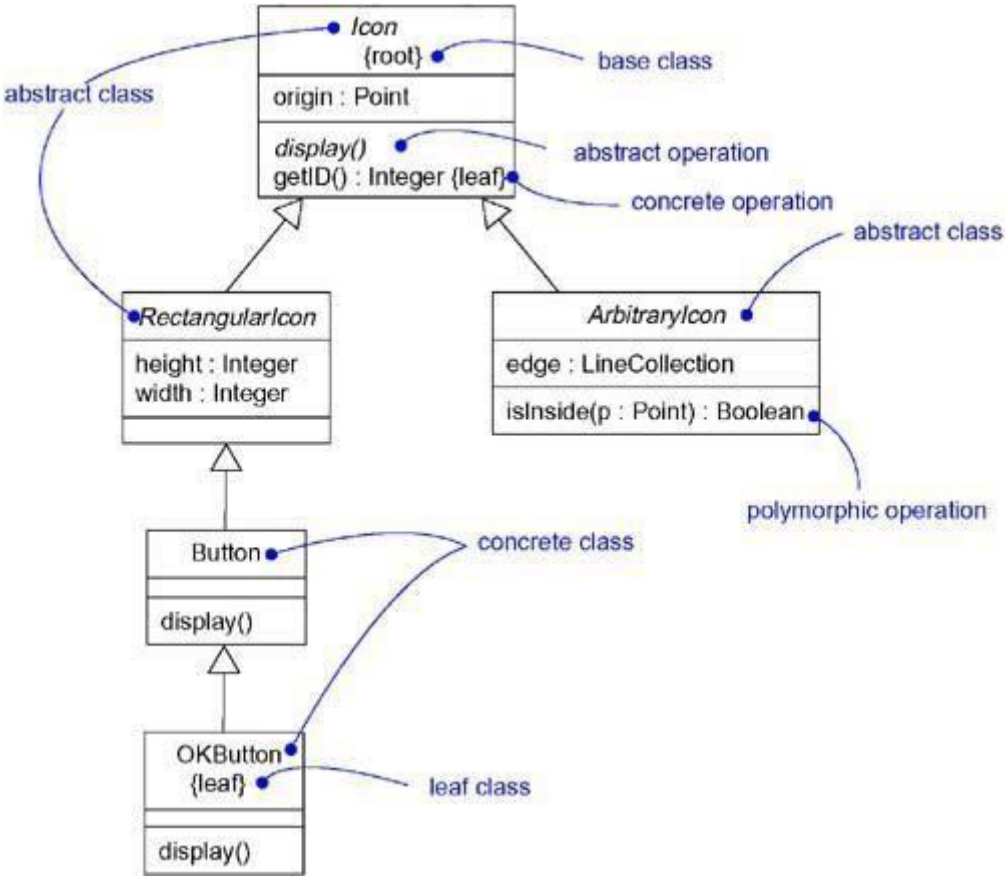


- **Scope(Owner Scope) of a Classifier**

The owner scope of a feature(attribute/operations) specifies whether the feature appears in each instance of the classifier or whether there is just a single instance of the feature for all instances of the classifier. *Two kinds of owner scope* – classifier scope and instance scope. *An instance scope* is an owner scope in which each instance of the classifier holds its own value for the feature whereas *classifier scope*is the one which have just one value of the feature for all instances of the classifier. classifier scope is rendered(shown) by underlining the feature's name. No adornment means that the feature is instance scoped. Figure:3 shows the scope of a classifier.

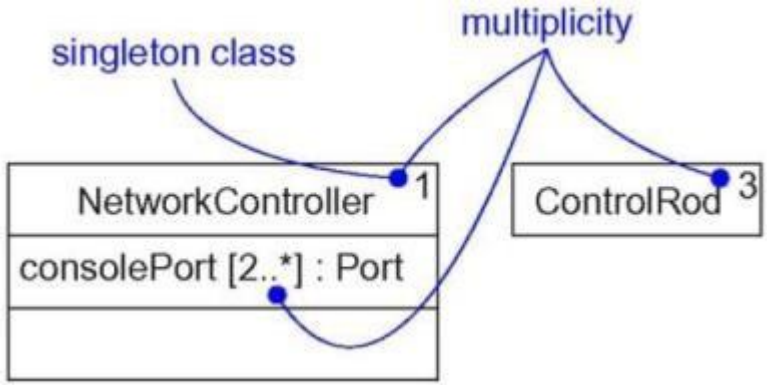


- **Abstract, Root, Leaf and Polymorphic Elements**
  - Abstract classes are those that do not have any direct instances and is specified in UML by writing its name in italics. A leaf class is a class that have no children and is specified in UML by writing the property leaf below the class's name. A root class is a class that has no parents and is specified in UML by writing the property root below the class's name.
  - An operation is polymorphic if it is specified with the same signature at different places in the hierarchy of classes. Which operation to invoke is done polymorphically, that is a match is determined at run time according to the type of the object. These are indicated in figure



For example, *display* and *isInside* are both polymorphic operations. Furthermore, the operation Icon::display() is abstract, meaning that it is incomplete and requires a child to supply an implementation of the operation. In the UML, you specify an abstract operation by writing its name in italics, just as you do for a class. By contrast, Icon::getID() is a leaf operation as indicated by the property leaf. This means that the operation is not polymorphic and may not be overridden.

- **Multiplicity**
  The number of instances a class may have is called its multiplicity. Multiplicity applies to attributes, as well. A class having single instance is called as a singleton class. It is indicated in Figure below.



- **Attributes**
  A class's structural features are indicated by its attributes.
  The ***syntax of an attribute*** in UML is
  [visibility] name [multiplicity] [: type] [= initial-value] [{property-string}]
  Some legal attribute declarations are given in Table below:

• origin	Name only
• + origin	Visibility and name
• origin : Point	Name and type
• head : *Item	Name and complex type
• name [0..1] : String	Name, multiplicity, and type
• origin : Point = (0,0)	Name, type, and initial value
• id : Integer {frozen}	Name and property

- Three defined ***properties that can be used with attribute*** values are given in Table below

1. changeable	There are no restrictions on modifying the attribute's value.
2. addOnly	For attributes with a multiplicity greater than one, additional values may be added, but once created, a value may not be removed or altered.
3. frozen	The attribute's value may not be changed after the object is initialized.

  Where default property is 'changeable'.

- **Operations**
  A class's behavioral features are indicated by its operations.
  The UML ***distinguishes between operation and method***. An operation specifies a service that can be requested from any object of the class to affect behavior; a method is an implementation of an operation.
- The ***syntax of an operation*** in the UML is

  [visibility] name [(parameter-list)][: return-type] [{property-string}]

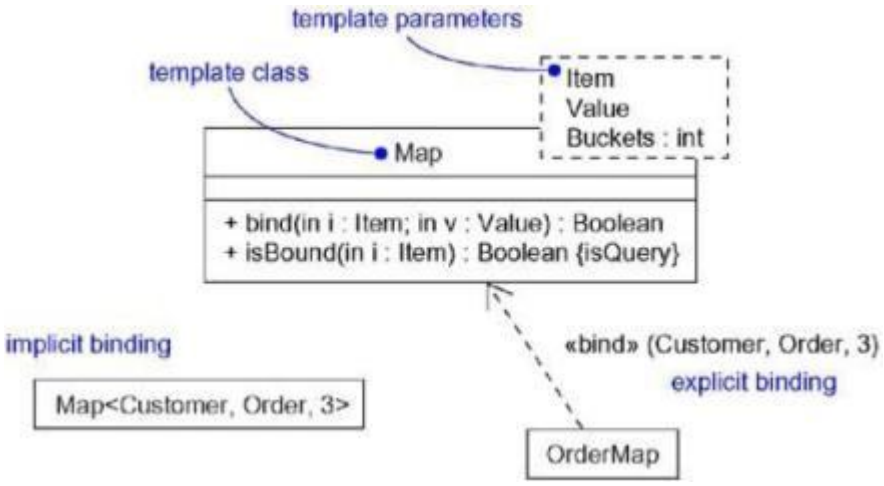
Template Classes

A template is a parameterized element. A template includes slots for classes, objects, and values, and these slots serve as the template's parameters. Every templates should be instantiated first. Instantiation involves binding these formal template parameters to actual ones. For a template class, the result is a concrete class that can be used just like any ordinary class.

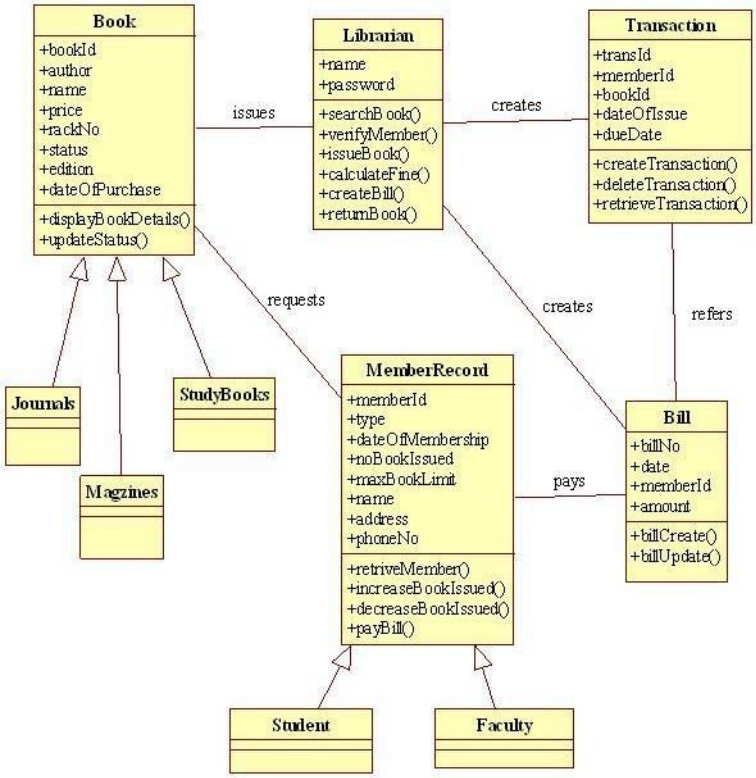
The ***instantiation of a template class can be modelled in two ways***.

First done implicitly, by declaring a class whose name provides the binding.

Second, explicitly by using a dependency stereotyped as bind, which specifies that the source instantiates the target template using the actual parameters. A template class is indicated in Figure below



SAMPLE CLASS DIAGRAM FOR LIBRARY MANAGEMENT SYSTEM



8. OBJECT DIAGRAMS & PACKAGES

- An object diagram shows a set of objects and their relationships at a point in time.
- An object diagram consists of the objects that collaborate, but without any of the messages passed among them.
- An object diagram is essentially an instance of a class diagram or the static part of an interaction diagram.
- Used to model the static design view or static process view of a system
- Object diagrams helps in modelling static data structures.
- Object diagrams commonly contain – Objects & Links.
- Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.
- Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.
- Object diagrams are used to render a set of objects and their relationships as an instance.
- Object diagram is similar to class diagram. The difference is that a class diagram represents an abstract model consisting of classes and their relationships. But an object diagram represents an instance at a particular moment which is concrete in nature.

PURPOSE

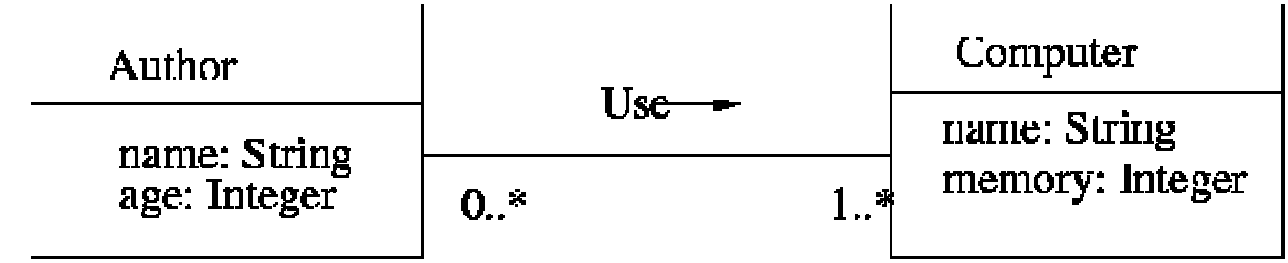
- Forward and reverse engineering.
- Object relationships of a system
- Static view of an interaction.
- Understand object behaviour and their relationship from practical perspective
- Making the prototype of a system.
- Modelling complex data structures.
- Understanding the system from practical perspective.

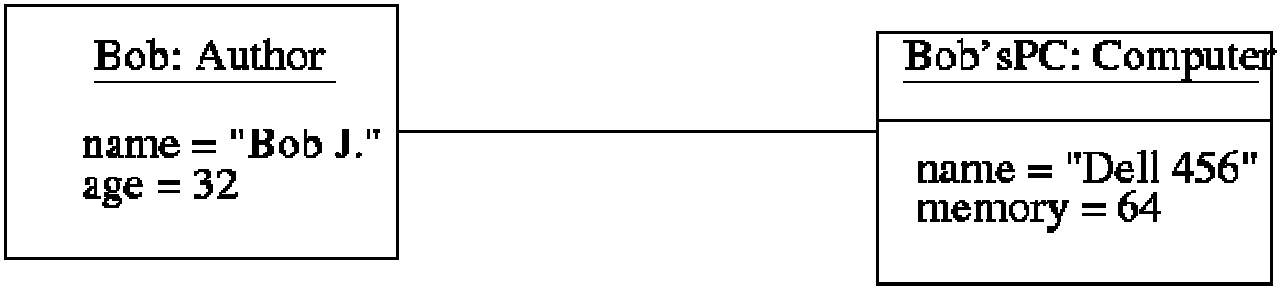
MODELLING OBJECT STRUCTURES

To model an object structure,

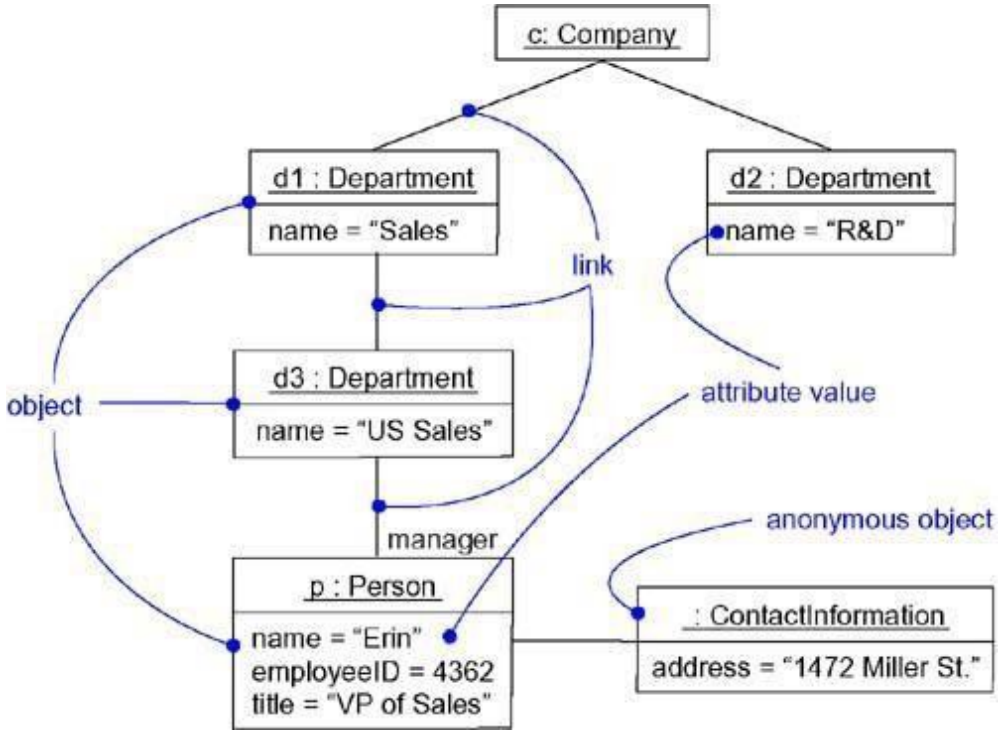
- Identify the mechanism you'd like to model. A mechanism represents some function or behaviour of the part of the system you are modelling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things, as well.
- Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.
- Expose the state and attribute values of each such object, as necessary, to understand the scenario.
- Similarly, expose the links among these objects, representing instances of associations among them.

OBJECT DIAGRAM EXAMPLE



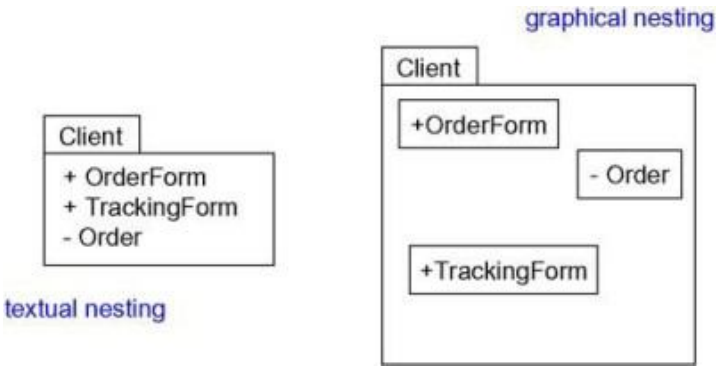


OBJECT DIAGRAM EXAMPLE - Company



PACKAGES

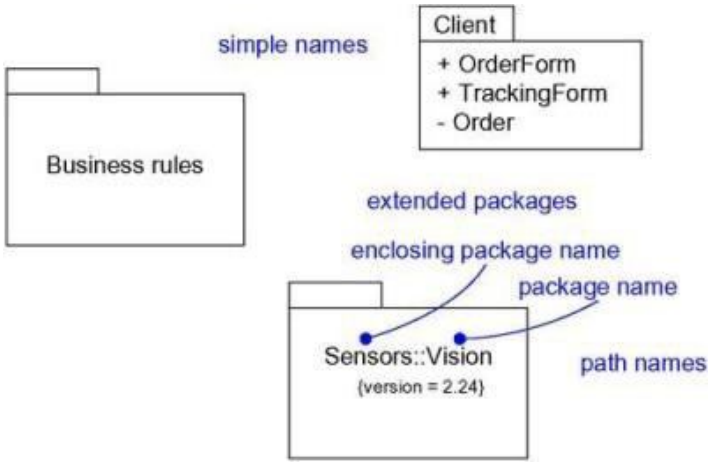
- A mechanism for organizing elements into groups
- Package name must be unique within its enclosing package
- Package must have a name that distinguishes it from other packages
- Two naming mechanism simple name, path name (prefixed by the name of the package in which that package lives).
- Package may contain classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages
- Package forms a namespace(every element in a package can be identified uniquely eg:P1:: Queue and P2:: Queue are different)
- Contents of a package can be shown textually or graphically as shown below.



- Visibility : +, -, #

1. Public elements (denoted by +) are visible outside the packages also.
2. Protected elements (denoted by #) are visible only to packages that inherit from another package
3. Private elements (denoted by -) are not visible outside the package.

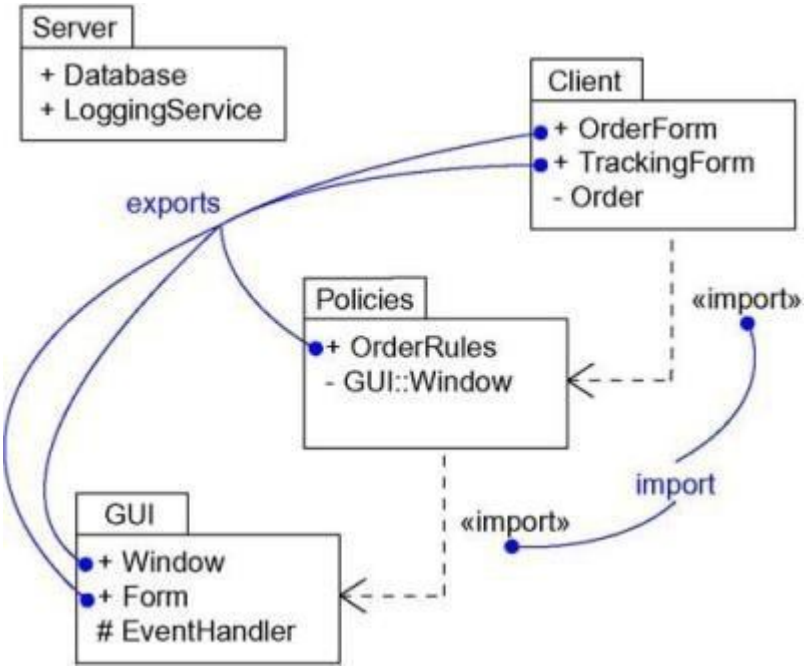
- Fully qualified name example Client::OrderForm.
- Two stereotypes import and access– both specify that the source package has access to the contents of the target. Import adds contents to source, so chances of name clashes, access doesn't add contents. These are non transitive



❖ **Importing and Exporting**

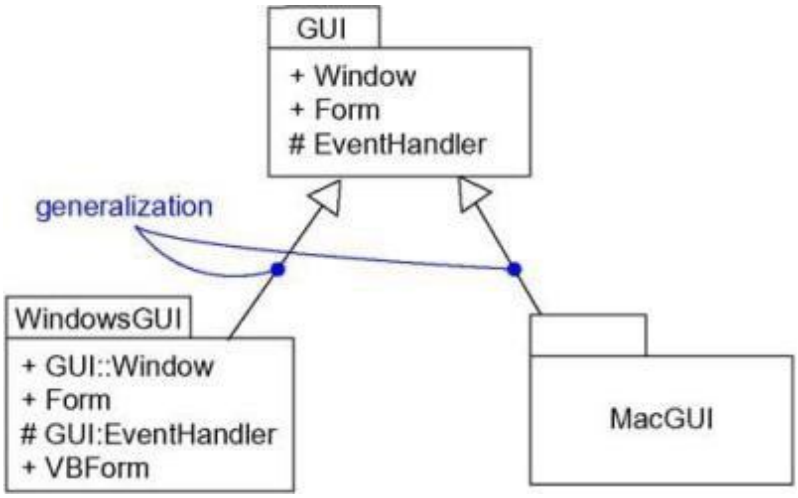
Importing means accessing the elements of the source by the target. It can be done by using the stereotype <<import>>. It's a one way process. It's non transitive. if A's package imports B's package, A can now see B, although B cannot see A. Importing grants a one-way permission for the elements in one package to access the elements in another package.

Exports are the public parts of a package. It's also non-transitive Figure below represents this.



❖ **Generalization**

In Generalization the specialized packages inherit the public and protected elements of the more general package. Figure below illustrates this.



❖ UML defines five standard stereotypes that apply to packages.
1. facade – Specifies a package that is only a view on some other package
2.framework – Specifies a package consisting mainly of patterns
3. stub – Specifies a package that serves as a proxy for the public contents of another package
4. subsystem – Specifies a package representing an independent part of the entire system being modeled
5. system – Specifies a package representing the entire system being modelled.

❖ Distinction between classes and packages:

Classes are abstractions of things found in a problem or solution; packages are mechanisms we use to organize the things in your model. Packages have no identity, classes do have identity through instances.

❖ **Modelling Groups of Elements**

To model groups of elements,

- Scan the modelling elements in a particular architectural view and look for **clumps** defined by elements that are conceptually or semantically close to one another.
- Surround each of these clumps in a package.
- For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.
- Explicitly connect packages that build on others via import dependencies.
- In the case of families of packages, connect specialized packages to their more general part via generalizations

❖ **Modelling Architectural Views**

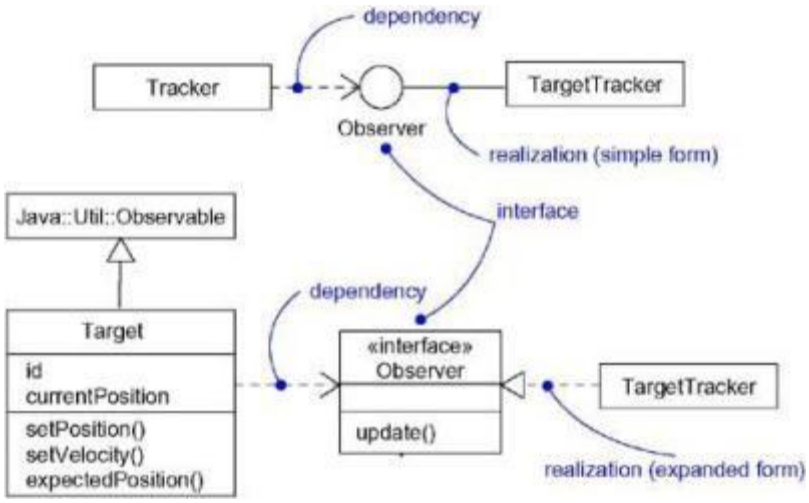
To model architectural views,

- Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, a process view, an implementation view, a deployment view, and a use case view.
- Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.
- As necessary, further group these elements into their own packages.
- There will typically be dependencies across the elements in different views. So, in general, let each view at the top of a system be open to all others at that level.

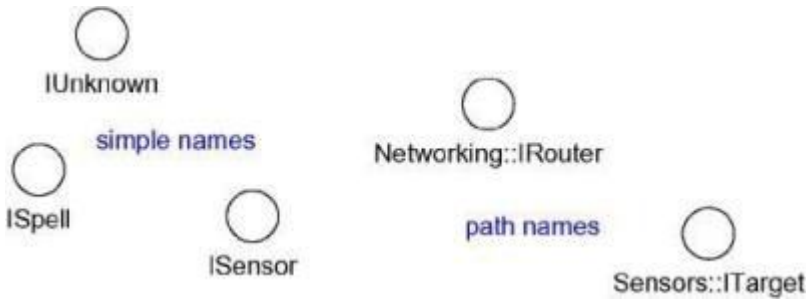
INTERFACES, TYPES & ROLES

❖ INTERFACE

- An interface is a collection of operations that are used to specify a service of a class or a component.
- ***Graphically***, an interface is rendered (represented) as a circle; in its expanded form, an interface may be rendered as a stereotyped class (a class with stereotype interface) as shown in Figure below.



- To distinguish an interface from a class, prepend an 'I' to every interface name. Operations in an interface may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
- Interface don't have attributes. Interfaces span model boundaries and it doesn't have direct instances.
- An interface may participate in generalization, association, dependency and realization relationships.
- Interfaces may also be used to specify a contract for a use case or subsystem.
- An interface name must be unique within its enclosing package. ***Two naming mechanism***; a simple name (only name of the interface), a path name is the interface name prefixed by the name of the package in which that interface lives represented in Figure below.

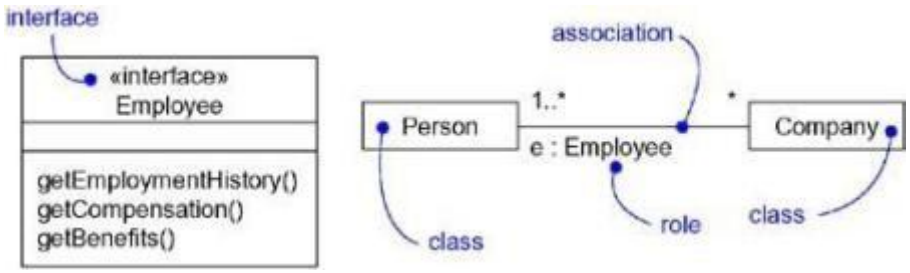


❖ **TYPE**

- A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object of that type.
- To distinguish a type from an interface or a class, prepend a 'T' to every type. Stereotype type is used to formally model the semantics of an abstraction and its conformance to a specific interface.

❖ **ROLE**

- A role names (indicates) a behaviour of an entity participating in a particular context. Or, a role is the face that an abstraction presents to the world.
- For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on.
- When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behaviour depending on the role that it plays at the time.
- For example, an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother. Figure below indicates a role employee played by person and is represented statically there.



9. INTERACTION DIAGRAMS

- Interaction diagram describe some type of interactions among the different elements in the model. So this interaction is a part of dynamic behaviour of the system.
- This interactive behaviour is represented in UML by two diagrams known as *Sequence diagram* and *Collaboration diagram*. The basic purposes of both the diagrams are similar.
- Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.
- Application of interaction diagrams
  - To model flow of control by time sequence.
  - To model flow of control by structural organizations.
  - For forward engineering.
  - For reverse engineering.
- Purpose of Interaction diagrams:
  - To capture dynamic behaviour of a system.
  - To describe the message flow in the system.
  - To describe structural organization of the objects.
  - To describe interaction among objects.

SEQUENCE DIAGRAM

- A sequence diagram shows elements as they interact over time, showing an interaction or interaction instance.
- Sequence diagrams are organized along two axes:
  1. The horizontal axis shows the elements that are involved in the interaction. The elements on the horizontal axis may appear in any order.
  2. The vertical axis represents time proceeding down the page.
- Sequence diagrams are made up of a number of elements, including class roles, specific objects, lifelines, and activations. All of these are described in the following subsections.

Class roles

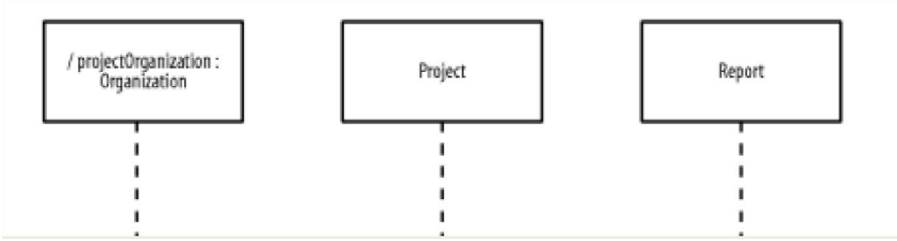
In a sequence diagram, a class role is shown using the notation for a class as defined in previously, but the class name is preceded by a forward slash followed by the name of the role that objects must conform to in order to participate within the role, followed by a colon. Other classes may also be shown as necessary, using the notation for classes, Class roles and other classes are used for specification-level collaborations communicated using sequence diagrams. Following figure shows the projectOrganization class role as well as the Project and Report classes.

Figure A: Class role and two classes

Lifelines

A lifeline, shown as a vertical dashed line from an element, represents the existence of the element over time. Figure B shows lifelines for the class role (projectOrganization) and classes (Project and Report) in Figure A.

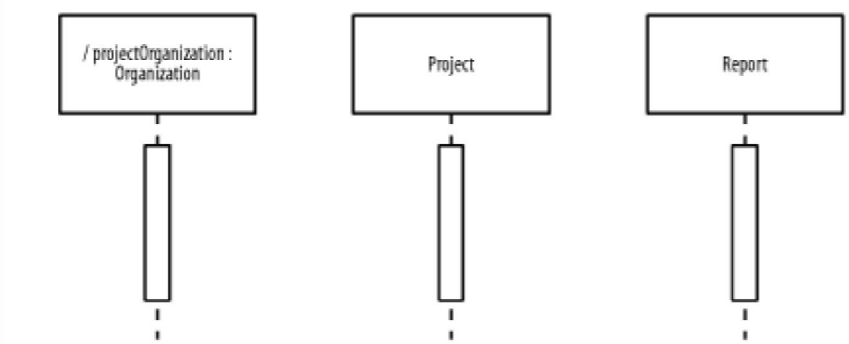
Figure B: Lifelines



Activations

An optional activation, shown as a tall, thin rectangle on a lifeline, represents the period during which an element is performing an operation. The top of the rectangle is aligned with the initiation time, and the bottom is aligned with the completion time. Figure C shows activations for the class roles in Figure A, where all the elements are simultaneously performing operations.

Figure D. Activations



COMMUNICATION / MESSAGING

- In a sequence diagram, a communication, message, or stimulus is shown as a horizontal solid arrow from the lifeline or activation of a sender to the lifeline or activation of a receiver.
- In the UML, communication is described using the following UML syntax:
  [guard] *[iteration]sequence_number : return_variable :=
  operation_name(argument_list)

in which:

guard

Is optional and indicates a condition that must be satisfied for the communication to be sent or occur. The square brackets are removed when no guard is specified.

iteration

Is optional and indicates the number of times the communication is sent or occurs. The asterisk and square brackets are removed when no iteration is specified.

sequence_**number**

Is an optional integer that indicates the order of the communication. The succeeding colon is removed when no sequence number is specified. Because the vertical axis represents time proceeding down the page on a sequence diagram, a sequence number is optional.

return_variable

Is optional and indicates a name for the value returned by the operation. If you choose not to show a return variable, or the operation does not return a value, you should also omit the succeeding colon and equal sign.

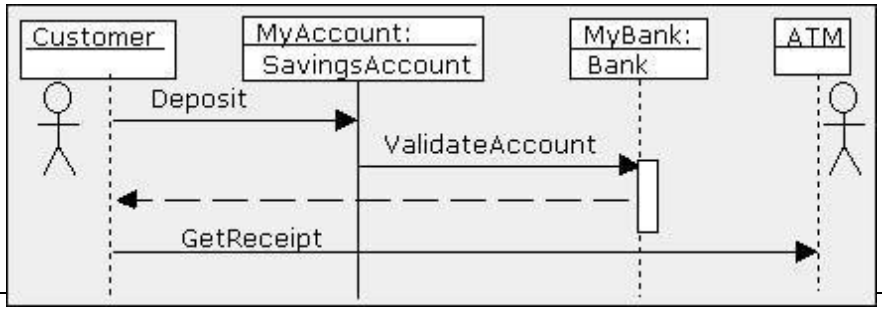
operation_name

Is the name of the operation to be invoked.

argument_list

Is optional and is a comma-separated list that indicates the arguments passed to the operation. Each parameter may be an explicit value or a return variable from a preceding communication. If an operation does not require any arguments, the parentheses are left empty.

- In some ways, a sequence diagram is like a stack trace of object messages. Below is a sample sequence diagram

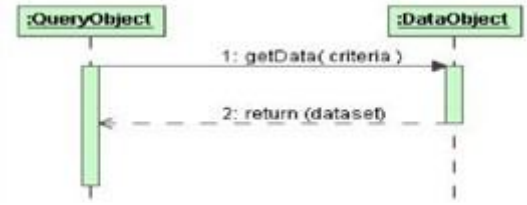


- The Sequence diagram allows the person reading the documentation to ***follow the flow of messages*** from each object. The vertical lines with the boxes on top represent instances of the classes (or objects).
- The label to the left of the colon is the instance and the label to the right of the colon is the class. The horizontal arrows are the messages passed between the instances and are read from top to bottom.
- In the above example a customer (user) depositing money into MyAccount which is an instance of Class SavingsAccount. Then MyAccount object Validates the account by asking the Bank object, MyBank to ValidateAccount. Finally, the Customer Asks the ATM object for a Receipt by calling the ATM's operation GetReceipt.
- The white rectangle indicate the scope of a method or set of methods occurring on the Object My Bank. The dotted line is a return from the method ValidateAccount.

TYPES OF MESSAGING

1. SYNCHRONOUS MESSAGING

- ✓ Assumes that a return is needed.
- ✓ Sender waits for the return before proceeding with any other activity
- ✓ Represented as a full arrow head
- ✓ Return messages are dashed arrows

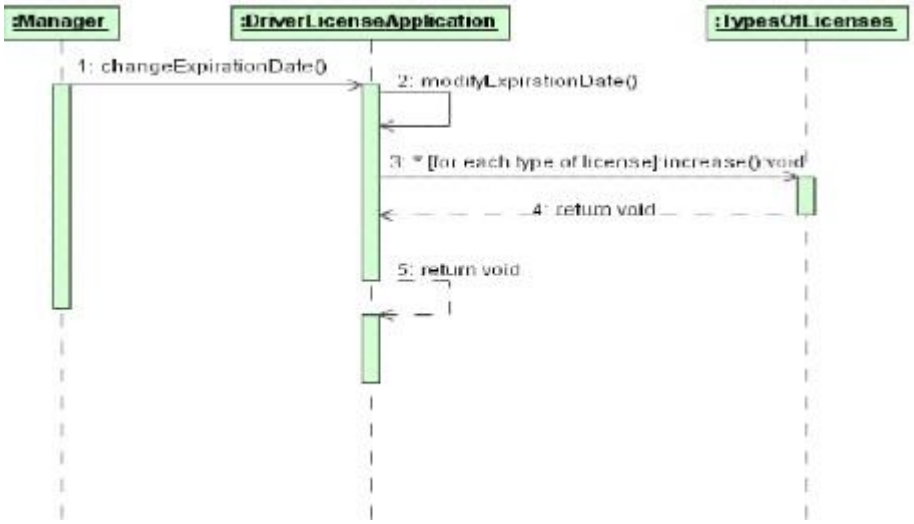


2. ASYNCHRONOUS MESSAGING

- ✓ Does not wait for a return message
- ✓ Exemplified by signals
- ✓ Sender only responsible for getting the message to the receiver
- ✓ Usually modelled using a solid line and a half arrowhead to distinguish it from the full arrowhead of the synchronous message

3. SELF REFERENCE MESSAGE

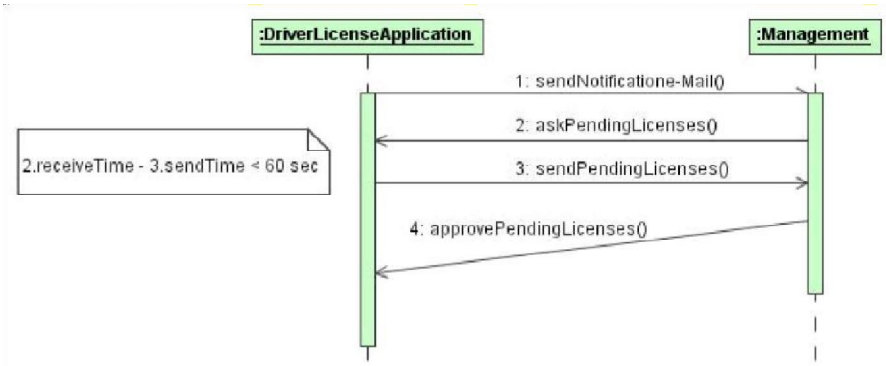
- ✓ A self-reference message is a message where the sender and receiver are one and the same object.
- ✓ In a self-reference message the object refers to itself when it makes the call.
- ✓ message 2 is only the invocation of some procedure that should be executed.



4. TIMED MESSAGES

- ✓ Messages may have user-defined time attributes, such as sentTime or receivedTime
- ✓ User-defined time attributes must be associated with message numbers
- ✓ Instantaneous messages are modeled with horizontal arrows
- ✓ Messages requiring a significant amount of time, it is possible to slant the arrow from the tail down to the head from Object Creation and Destruction .

For Example

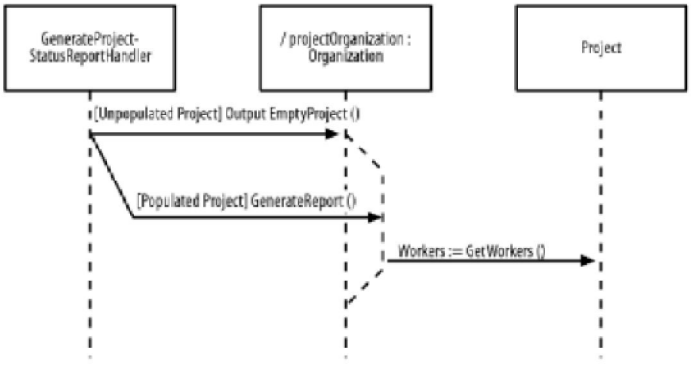


For messages 1, 2 and 3 the time required for their execution is considered equal to zero.

Message 4 requires more time (time > 0) for its execution.

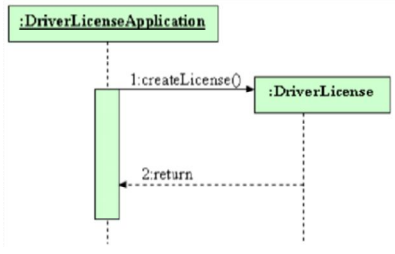
5. CONDITIONAL MESSAGING
   - In a sequence diagram, conditionality (which involves communicating one set of messages or stimuli rather than another set of messages or stimuli) within a generic-form interaction is shown as multiple communications leaving a single point on a lifeline or activation, with the communications having mutually exclusive guard expressions.
   - A lifeline may also split into two or more lifelines to show how a single element would handle multiple incoming communications, and the lifelines would subsequently merge together again.
   - Following figure shows the Generate Project-Status Report interaction and collaboration description where the GenerateProject-StatusReportHandler class requests that the projectOrganization class role indicate that the project is empty if the project is a newly created or unpopulated project, and the GenerateProject-StatusReportHandler class requests that the projectOrganization class role continue generating information for the report element if the project is not a newly created or populated project. In this figure, only the first communication is shown for actually generating the report. If there are no other communications for actually generating the report, the GenerateReport communication may go to the same lifeline as the OutputEmptyProject communication. we use different lifelines in the figure because each lifeline represents a different path of execution.

Sequence diagram conditionality within a generic-form interaction

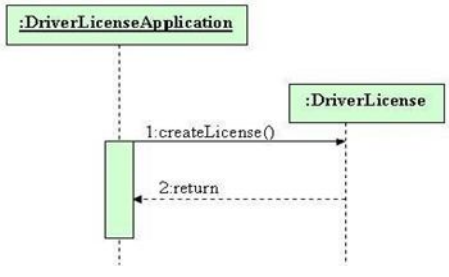


Object Creation: If the object is created during the sequence execution it should appear somewhere below the top of the diagram.
For example.

Alternative A

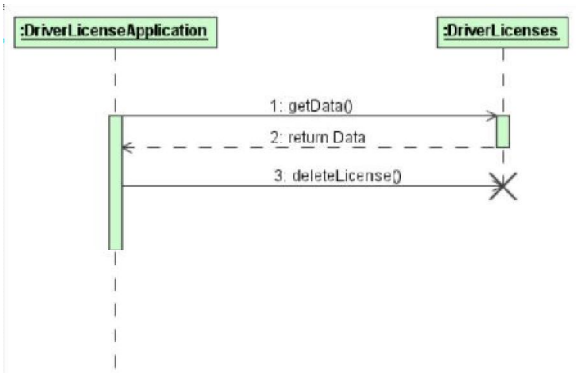


Alternative B



Object Destruction: If the object is deleted during the sequence execution, place an X at the point in the object lifeline when the termination occurs.

For example.



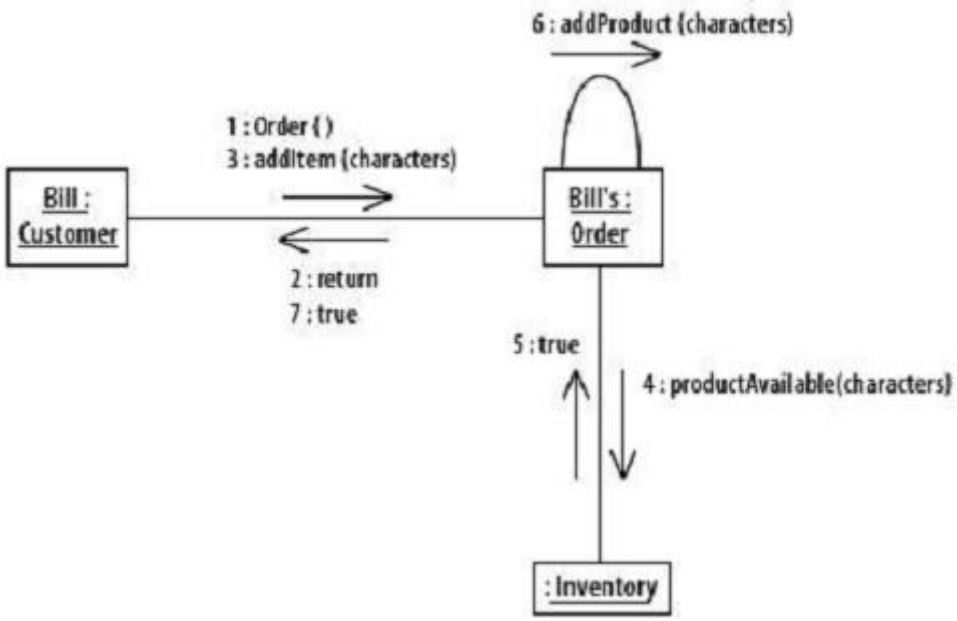
COLLABORATION DIAGRAM

- A collaboration diagram shows elements as they interact over time and how they are related. That is, it shows a collaboration or collaboration instance.
- While sequence diagrams are time-oriented and emphasize the overall flow of an interaction, collaboration diagrams are time- and space oriented and emphasize the overall interaction, the elements involved, and their relationships.
- Sequence diagrams are especially useful for complex interactions, because you read them from top to bottom. Collaboration diagrams are especially useful for visualizing the impact of an interaction on the various elements, because you can place an element on a diagram and immediately see all the other elements with which it interacts.
- A collaboration diagram shows the interactions organized around the structure of a model, using either:

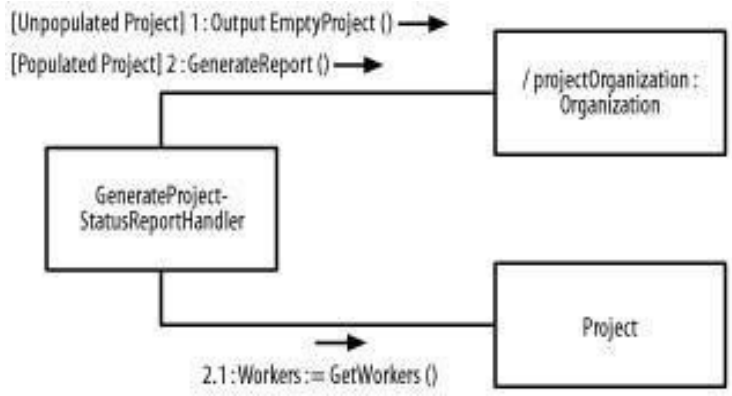
  a) Classifiers (e.g. classes) and associations, or
  b) Instances (e.g. objects) and links.

- Collaboration diagram is an interaction diagram which is similar to the sequence diagram. It reveals both structural and dynamic aspects of a collaboration. It also reveals the need for the associations in the class diagram
- A collaboration diagram shows a graph of either instances linked to each other or classifiers and associations.
- Navigability is shown using arrow heads on the lines representing links.
- An arrow next to a line indicates a stimuli or message flowing in the given direction.
- The order of interaction is given with a number

- Conditions can be represented in collaboration diagram as shown below.

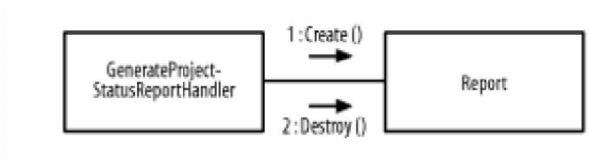


Collaboration diagram conditionality within a generic-form interaction



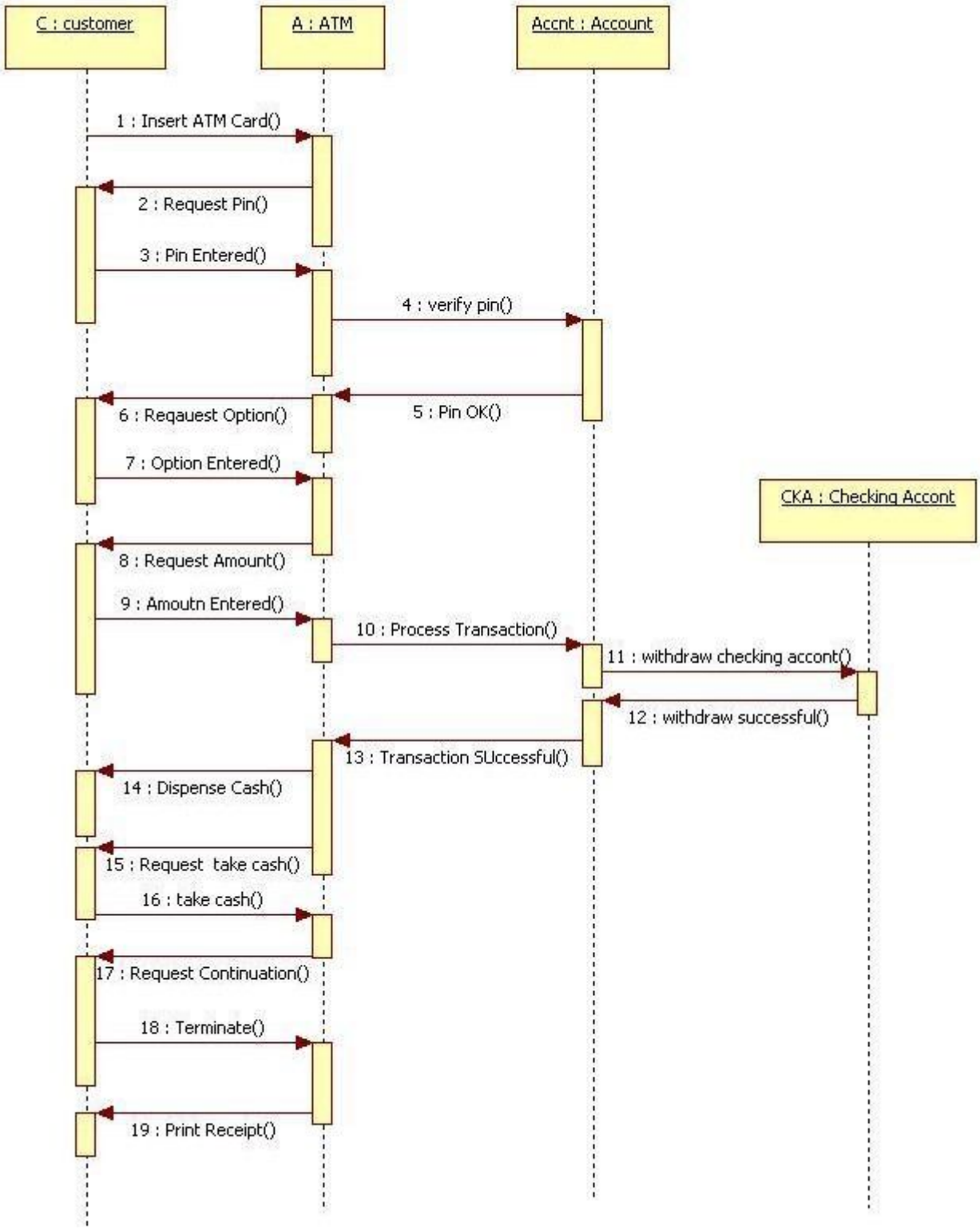
OBJECT CREATION AND DESTRUCTION

- A communication that creates an element and a communication that destroys an element are simply shown like any other communication.
- For example.

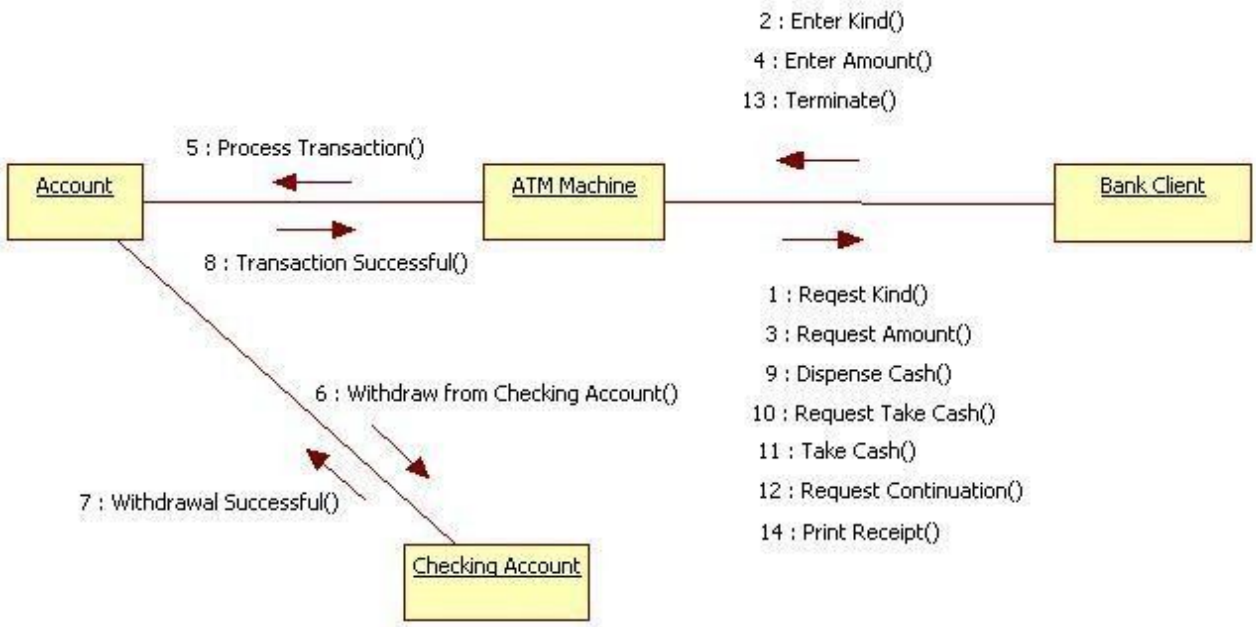


INTERACTION DIAGRAM FOR ATM

SEQUENCE DIAGRAM



COLLABORATION DIAGRAM



10. ACTIVITY DIAGRAM

- Activity diagram is another important diagram in UML to describe dynamic aspects of the system.
- Activity diagram is basically a flow chart to represent the flow form one activity to another activity. The activity can be described as an operation of the system.
- So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent.
- Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

Activity Diagram Symbols

The start symbol represents the beginning of a process or workflow in an activity diagram. It can be used by itself or with a note symbol that explains the starting point.

The activity symbol is the main component of an activity diagram. These shapes indicate the activities that make up a modelled process.

The connector symbol is represented by arrowed lines that show the directional flow, or control flow, of the activity. An incoming arrow starts a step of an activity; once the step is completed, the flow continues with the outgoing arrow.

The join symbol, or synchronization bar, is a thick vertical or horizontal line. It combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a time

A fork is symbolized with multiple arrowed lines from a join. It splits a single activity flow into two concurrent activities.

The decision symbol is a diamond shape; it represents the branching or merging of various flows with the symbol acting as a frame or container.

The note symbol allows the diagram creators or collaborators to communicate additional messages that don't fit within the diagram itself.

The receive signal symbol demonstrates the acceptance of an event. After the event is received, the flow that comes from this action is completed.

The send signal symbol means that a signal is being sent to a receiving activity, as seen above.

The flow final symbol shows the ending point of a process' flow. While a flow final symbol marks the end of a process in a single flow, an end symbol represents the completion of all flows in an activity.

- **The end symbol** represents the completion of a process or workflow.

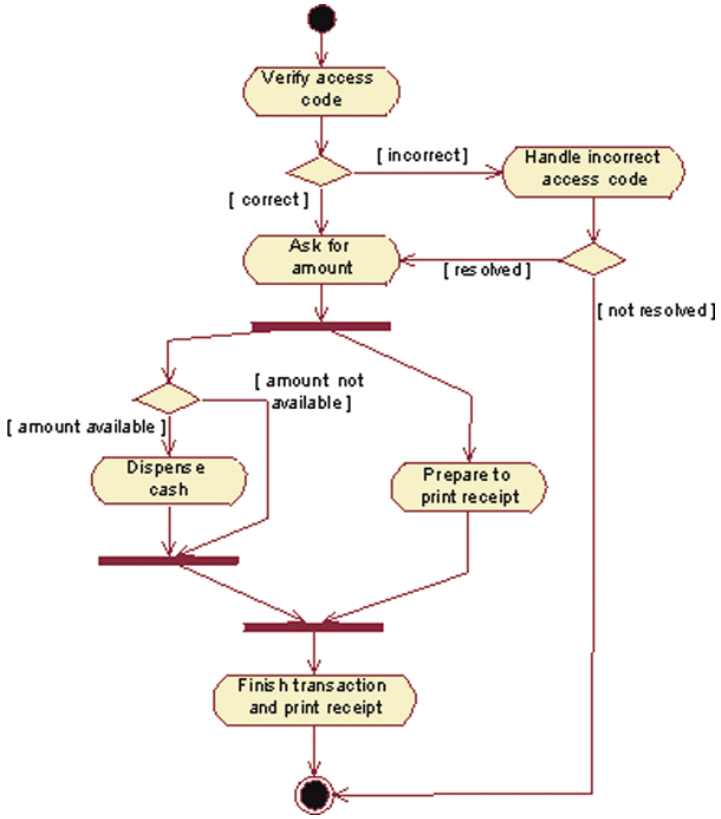
PURPOSE OF ACTIVITY DIAGRAM

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

APPLICATIONS OF ACTIVITY DIAGRAM

- Modelling work flow by using activities.
- Modelling business requirements.
- High level understanding of the system's functionalities.
- Investigate business requirements at a later stage.

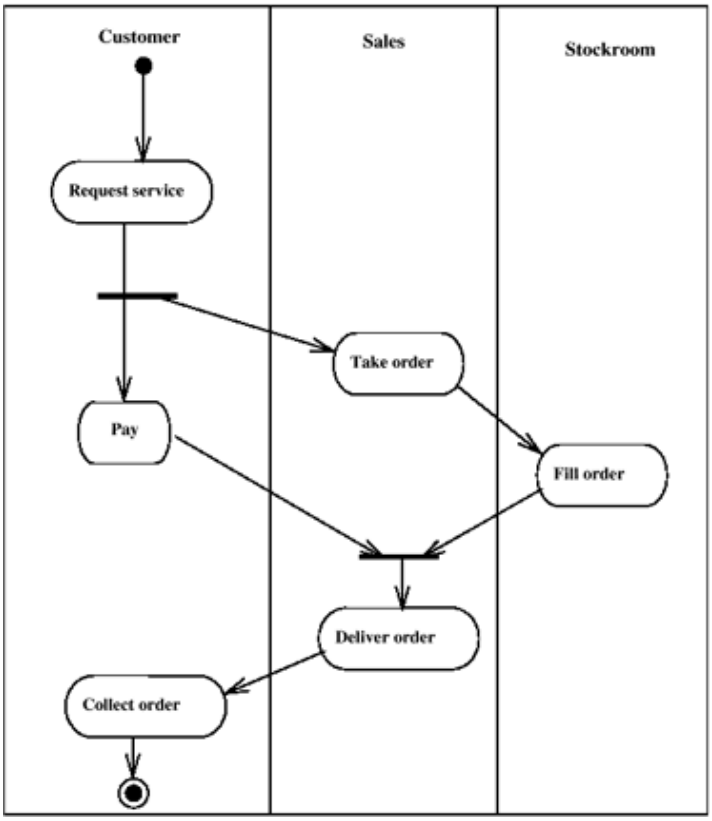
ACTIVITY DIAGRAM FOR ATM WITHDRAWAL TRANSACTION



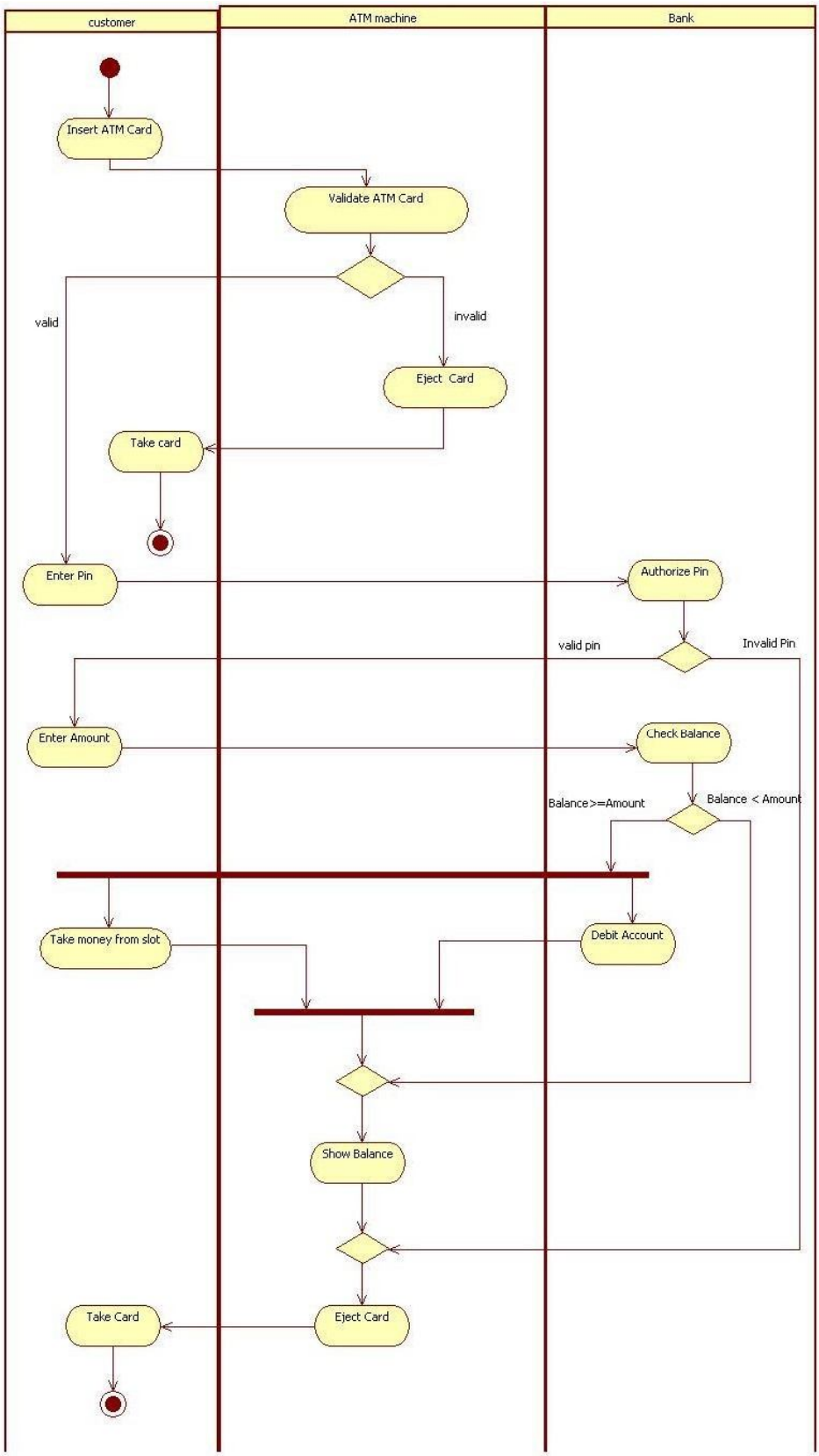
SWIMLANES

- In activity diagrams, it is often useful to model the activity's procedural flow of control between the objects (persons, organizations, or other responsible entities) that actually execute the action. To do this, you can add swimlanes to the activity diagram (swimlanes are named for their resemblance to the straight-line boundaries between two or more competitors at a swim meet).
- To put swimlanes on an activity diagram, use vertical columns. For each object that executes one or more actions, assign a column its name, placed at the top of the column. Then place each action associated with an object in that object's swimlane.
- A process flow or workflow diagram does not have to use swimlanes. However, since swimlanes communicate additional information about who performs the activity or when it takes place, it's typically a preferred best practice to include them.
- Similarly, a swimlane diagram can use only one set of swimlanes (either vertical swimlanes or horizontal swimlanes).
- In the UML standard, the activity diagram flows from top to bottom and veritcal swimlanes are most commonly used.
- One of the more common choices used by creators of swimlane diagrams is to define the roles which perform each activity within horizontal swimlanes and define the process stages in which the activity occurs within vertical swimlanes.

SWIMLANE ACTIVITY DIAGRAM FOR SALES MANAGEMENT SYSTEM



SWIMLANE ACTIVITY DIAGRAM FOR ATM



11. USE CASE DIAGRAM

- 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.
- A use case diagram can identify the different types of users of a system and the different use cases
- Use case diagrams consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application.
- A single use case diagram captures a particular functionality of a system. So to model the entire system numbers of use case diagrams are used.
- A use case diagram does not show the detail of the use cases: it only summarizes some of the relationships between use cases, actors, and systems. In particular, the diagram does not show the order in which steps are performed to achieve the goals of each use case.
- A use case diagram contains four components.
    - The boundary, which defines the system of interest in relation to the world around it.
    - The actors, usually individuals involved with the system defined according to their roles.
    - The use cases, which the specific roles are played by the actors within and around the system.
    - The relationships between and among the actors and the use cases.
- An *actor* is a class of person, organization, device, or external software component that interacts with your system. Example actors are **Customer**, **Restaurant**, **Temperature Sensor**, **Credit card Authorizer.**
- A *use case* represents the actions that are performed by one or more actors in the pursuit of a particular goal. Example use cases are **Order product**, **Update Menu**, **Process Payment**.

Purpose:

- The purpose of use case diagram is to capture the dynamic aspect of a system.
- Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analysed to gather its functionalities use cases are prepared and actors are identified.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

How to draw Use Case Diagram?

1. Identify the following regarding the system:
    - Functionalities to be represented as an use case
    - Actors
    - Relationships among the use cases and actors.

2. The name of a use case is very important. So the name should be chosen in such a way so that it can identify the functionalities performed.
3. Give a suitable name for actors.
4. Show relationships and dependencies clearly in the diagram.
5. Do not try to include all types of relationships. Because the main purpose of the

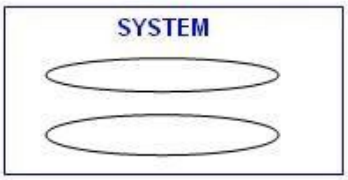
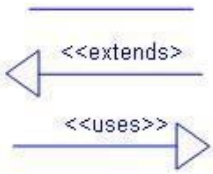
diagram is to identify requirements.

6. Use note whenever required to clarify some important points.

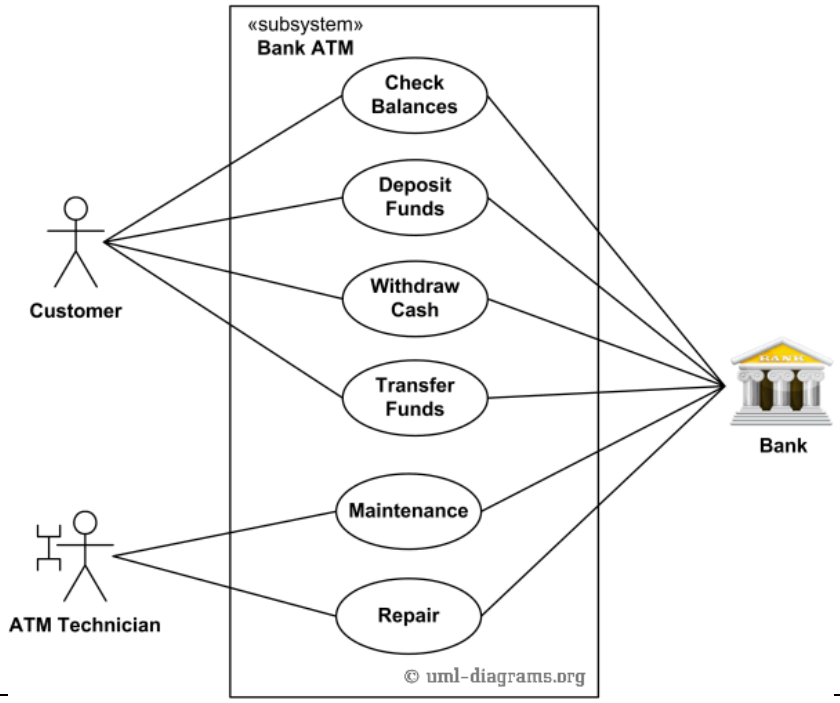
Application areas of Use Case Diagrams

- Requirement analysis and high level design.
- Model the context of a system.
- Reverse engineering.
- Forward engineering.

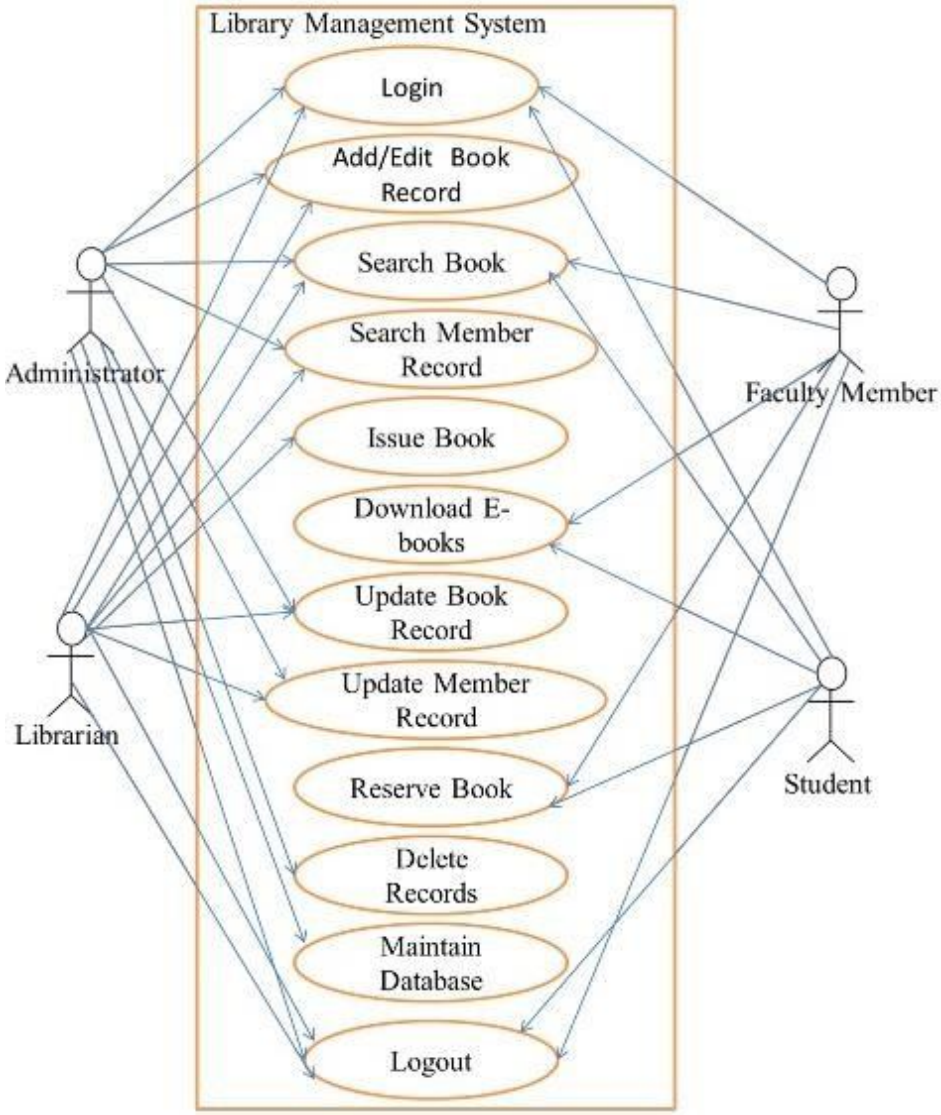
NOTATIONS USED IN USE CASE DIAGRAMS

Symbol	Reference Name
	Actor
	Use Case
	System
	Relationship

USE CASE DIAGRAM FOR ATM



USE CASE DIAGRAM FOR LIBRARY MANAGEMENT SYSTEM (LMS)



12. STATE CHART DIAGRAM

- State chart diagram is simply a presentation of a state machine which shows the flow of control from state to state.
- State chart diagrams are important for constructing executable systems through forward and reverse engineering.
- State chart diagrams are useful in modelling the lifetime of an object
- State chart diagrams commonly contain – Simple states and composite states, Transitions-including events and actions.
- Used for modelling the dynamic aspects of systems.
- Graphically, a state chart diagram is a collection of vertices and arcs.

`PURPOSE

- To model dynamic aspect of a system.
- To model life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model states of an object.

APPLICATIONS

- To model object states of a system.
- To model reactive system. Reactive system consists of reactive objects.
- To identify events responsible for state changes.
- Forward and reverse engineering.

TERMS / NOTATIONS

- *A state* is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- *An even*t in the context of state machines is an occurrence of a stimulus that can trigger a state transition.
- *A transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- *An activity* is ongoing non-atomic execution within a state machine.
- An action is an executable atomic computation that results in a change in state of the model or the return of a value.
- *A reactive or event-driven object* is one whose behaviour is best characterized by its response to events dispatched from outside its context

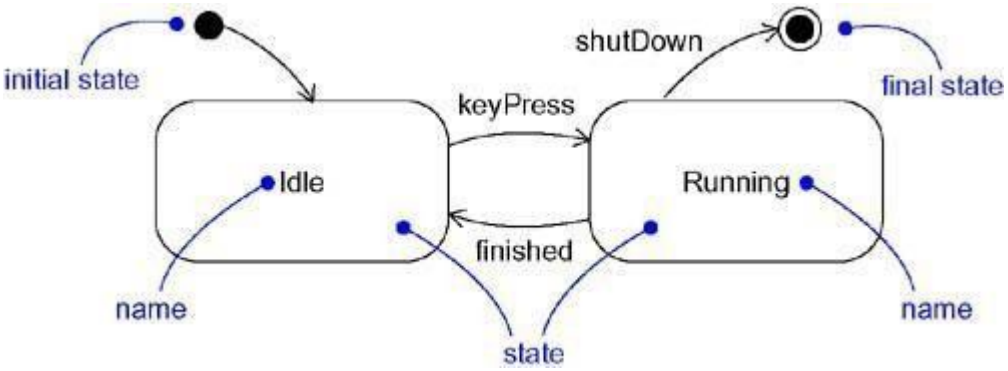
❖ ***States***

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An object remains in a state for a finite amount of time. For example, a Heater in a home

might be in any of four states: Idle, Activating, Active, and ShuttingDown.

- A state name must be unique within its enclosing state

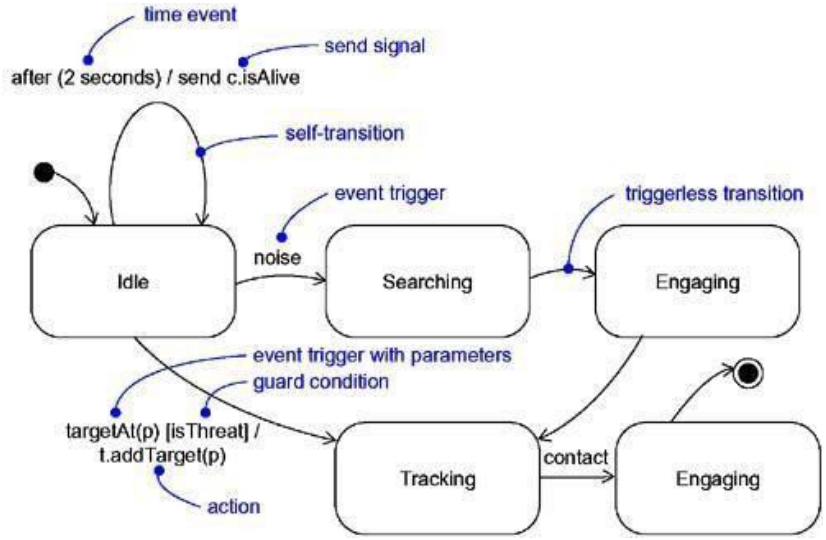
- A state has five parts:
    1. *Name, Entry/exit actions, Internal transitions* – Transitions that are handled without causing a change in state,
    2. *Substates* – nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates,
    3. *Deferred events* – A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state
- Initial state indicates the default starting place for the state machine or sub state and is represented as a filled black circle
- Final state indicates that the execution of the state machine or the enclosing state has been completed and is represented as a filled black circle surrounded by an unfilled circle



❖ ***Transitions***

- A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- Transition fires means change of state occurs. Until transition fires, the object is in the source state; after it fires, it is said to be in the target state.
- A transition has five parts:

    1. *Source state* – The state affected by the transition.
    2. *Event trigger* – a stimulus that can trigger a source state to fire on satisfying guard condition.
    3. *Guard condition* – Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger.
    4. *Action* – An executable atomic computation that may directly act on the object that owns the state machine, and indirectly on other objects that are visible to the object.
    5. *Target state* – The state that is active after the completion of the transition.
- A transition may have multiple sources as well as multiple targets
- A *self-transition* is a transition whose source and target states are the same



❖ ***Event Trigger***

- An event in the context of state machines is an occurrence of a stimulus that can trigger a state transition.
- Events may include signals, calls, the passing of time or a change in state.
- An event – signal or a call – may have parameters whose values are available to the transition, including expressions for the guard condition and action.
- An event trigger may be polymorphic

❖ ***Guard condition***

- A guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event
- A guard condition is evaluated only after the trigger event for its transition occurs
- A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered

❖ ***Action***

- An action is an executable atomic computation i.e, it cannot be interrupted by an event and runs to completion.
- Actions may include operation calls, the creation or destruction of another object, or the sending of a signal to an object
- *An activity* may be interrupted by other events.

❖ **Advanced States and Transitions**

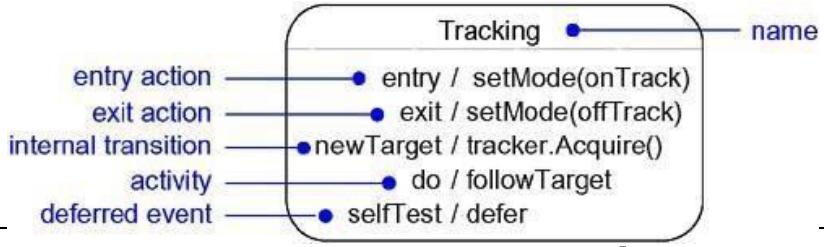


Figure: Advanced States and Transitions

Entry and Exit Actions

- Entry Actions are those actions that are to be done upon entry of a state and are shown by the keyword event 'entry' with an appropriate action.
- Exit Actions are those actions that are to be done upon exit from a state marked by the keyword event 'exit', together with an appropriate action.

Internal Transitions

- Internal Transitions are events that should be handled internally without leaving the state.
- Internal transitions may have events with parameters and guard conditions.

Activities

Activities make use of object's idle time when inside a state. 'do' transition is used to specify the work that's to be done inside a state after the entry action is dispatched.

Deferred Events

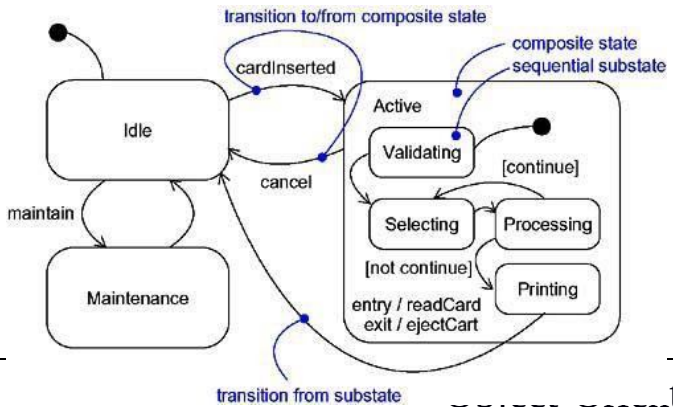
A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred. A deferred event is specified by listing the event with the special action 'defer'.

Sub-states

- A sub-state is a state that's nested inside another one.
- A state that has sub-states is called a composite state.
- A composite state may contain either *concurrent (orthogonal) or sequential (disjoint) sub-states*.
- Sub-states may be nested to any level

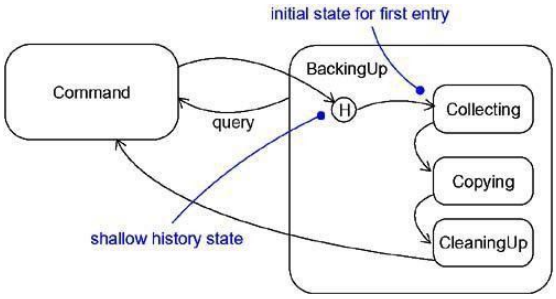
Sequential Sub-states

- Sequential sub-states are those sub-states in which an event common to the composite states can easily be exercised by each states inside it at any time
- Sequential sub-states partition the state space of the composite state into disjoint states
- A nested sequential state machine may have at most one initial state and one final state



History States

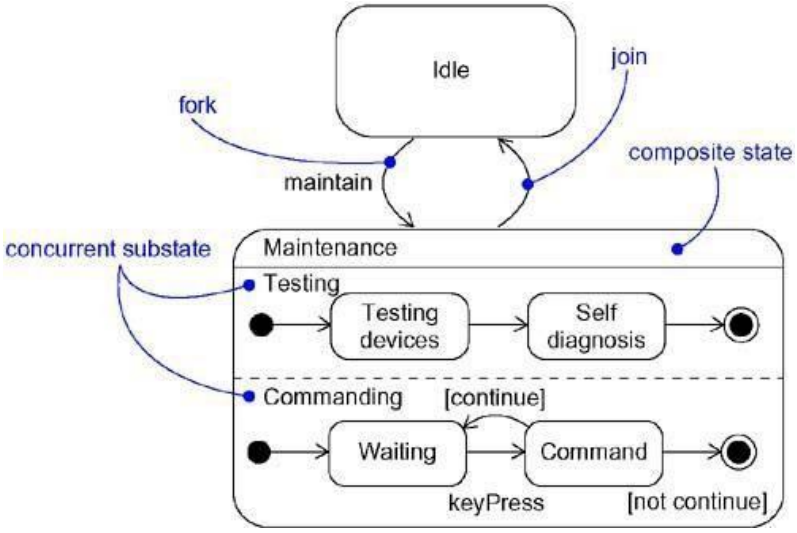
- A history state allows composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state.
- A shallow history state is represented as a small circle containing the symbol H
- The first time entry to a composite state doesn't have any history and the process for collecting history is as shown in the figure below.



- The symbol H designates a ***shallow history***, which remembers only the history of the immediate nested state machine.
- The symbol H* designates ***deep history***, which remembers down to the innermost nested state at any depth.
- When only one level of nesting, shallow and deep history states are semantically equivalent.

Concurrent Substates

- Concurrent substates specify two or more state machines that execute in parallel in the context of the enclosing object
- Execution of these concurrent substates continues in parallel. These substates waits for each other to finish to joins back into one flow
- A nested concurrent state machine does not have an initial, final, or history state



- **Modelling the Lifetime of an Object**
  1. Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
     a) *If the context is a class or a use case*, collect the neighbouring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbours are candidate targets for actions and are candidates for including in guard conditions.
     b) *If the context is the system as a whole*, narrow your focus to one behaviour of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.
  2. Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and post-conditions of the initial and final states, respectively.
  3. Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
  4. Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
  5. Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).
  6. Expand these states as necessary by using sub-states.
  7. Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
  8. Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
  9. Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.
  10. After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

- For example, Figure below shows the state machine for the controller in a home security system, which is responsible for monitoring various sensors around the perimeter of the house

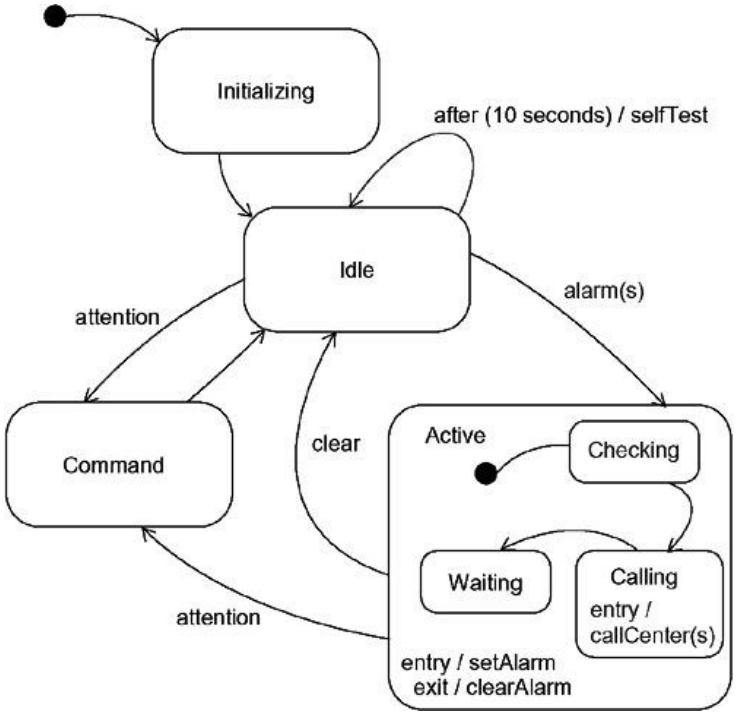


Figure: Modelling the Lifetime of an Object

13. EVENTS & SIGNALS

EVENTS

- ✓ An event is the specification of a significant occurrence that has a location in time and space.
- ✓ Anything that happens is modelled as an event in UML.
- ✓ In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- ✓ Four kinds of events – signals, calls, the passing of time, and a change in state.
- ✓ Figure 1: Events
- ✓ Events may be external or internal and asynchronous or synchronous.

    - o *Asynchronous events* are events that can happen at arbitrary times eg:- signal, the passing of time, and a change of state.
    - o *Synchronous events*, represents the invocation of an operation eg:- Calls
    - o *External events* are those that pass between the system and its actors.
    - o *Internal events* are those that pass among the objects that live inside the system

TYPES OF EVENTS

- ➢ SIGNALS / SIGNAL EVENT
    - ✓ *A signal* is an event that represents the specification of an asynchronous stimulus communicated between instances.
    - ✓ *A* signal event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are an example of internal signal.
    - ✓ *A* signal event is an asynchronous event.
    - ✓ Signal events may have instances, generalization relationships, attributes and operations. Attributes of a signal serve as its parameters.
    - ✓ A signal event may be sent as the action of a state transition in a state machine or the sending of a message in an interaction.
    - ✓ Signals are modelled as stereotyped classes and the relationship between an operation and the events by using a dependency relationship, stereotyped as send.
    - ✓ Figure below shows Signal Events

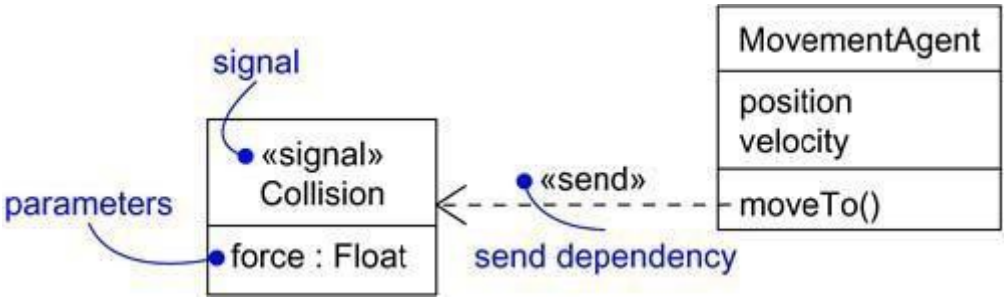
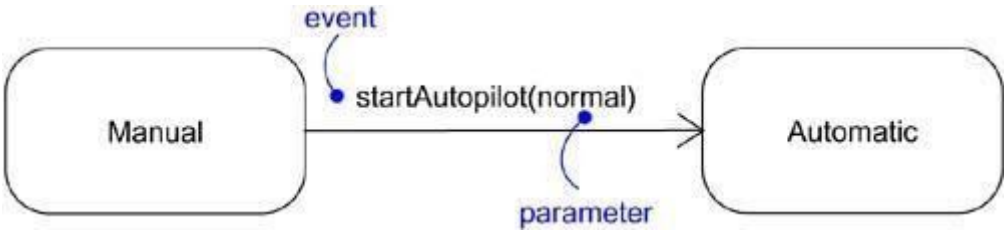


Figure: Signals

- ***CALL EVENT***
  - ✓ A call event represents the dispatch of an operation.
  - ✓ A call event is a synchronous event.
  - ✓ Figure below shows Call Events



- ***TIME AND CHANGE EVENTS***

  - ✓ A *time event* is an event that represents the passage of time..
  - ✓ Modelled by using the keyword 'after' followed by some expression that evaluates to a period of time which can be simple or complex.
  - ✓ A *change event* is an event that represents a change in state or the satisfaction of some condition
  - ✓ Modelled by using the keyword 'when' followed by some Boolean expression

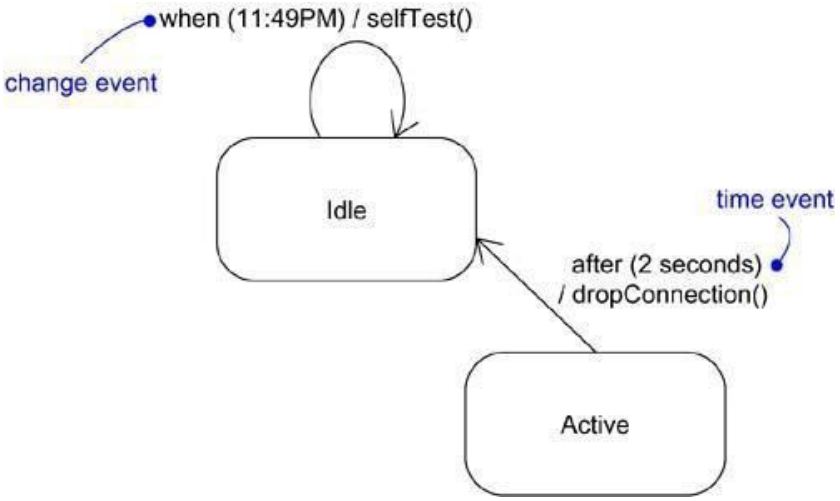


Figure: Time and Change Events

- ***Sending and Receiving Events***

For synchronous events (Sending or Receiving) like call event, the sender and the receiver are in a rendezvous(the sender dispatches the signal and wait for a response from the receiver) for the duration of the operation. when an object calls an operation, the sender dispatches the operation and then waits for the receiver.

For asynchronous events (Sending or Receiving) like signal event, the sender and receiver do not rendezvous ie,the sender dispatches the signal but does not wait for a response from the receiver.

When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver.

- Call events can be modelled as operations on the class of the object.
- Named signals can be modelled by naming them in an extra compartment of the class as in Figure below.

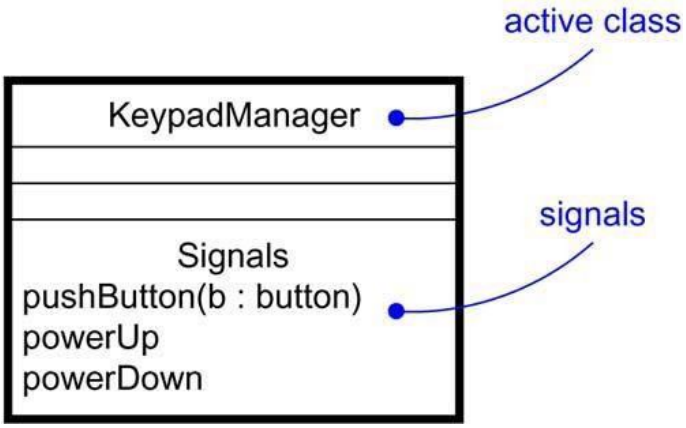
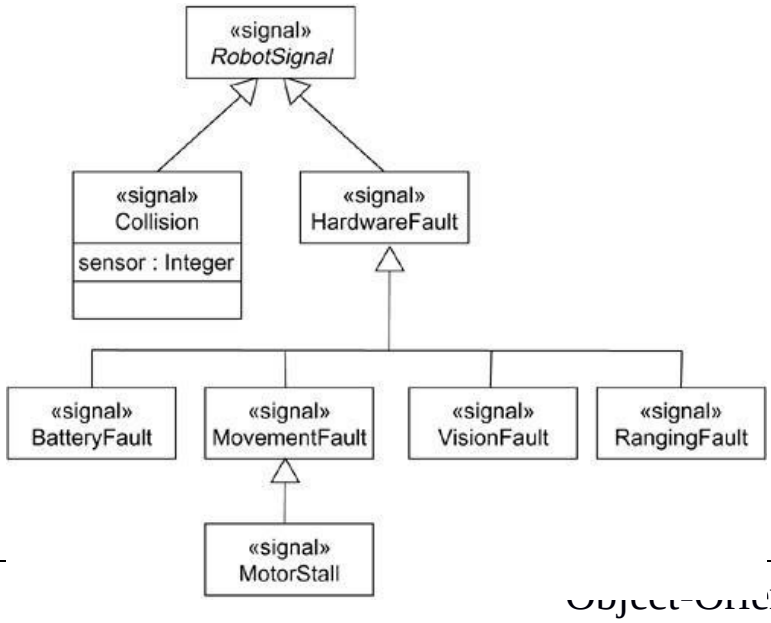


Figure: Signals and Active Classes

Modelling family of signals

- To model a family of signals,
  1. Consider all the different kinds of signals to which a given set of active objects may respond.
  2. Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.
  3. Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.



Modelling Exceptions

✓ To model exceptions,

1. For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.
2. Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.
3. For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing send dependencies from an operation to its exceptions) or you can put this in the operation's specification.

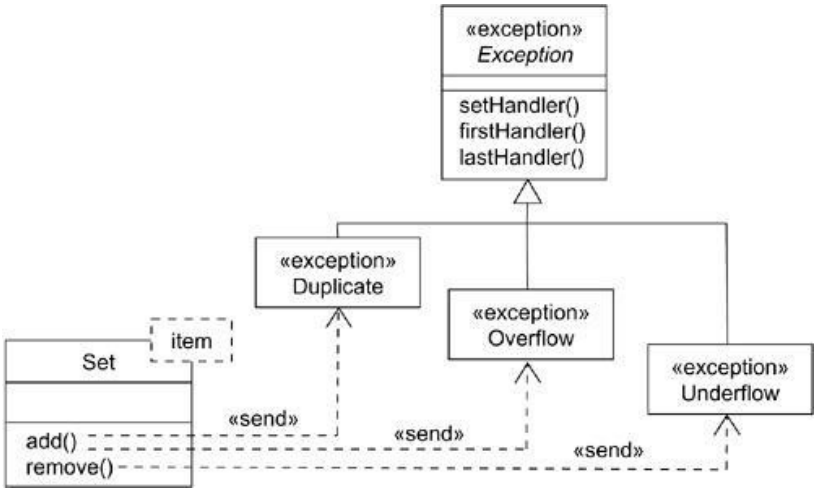
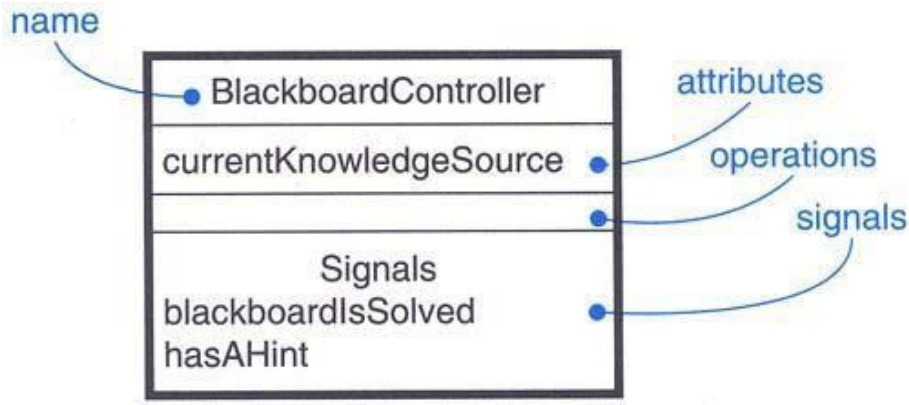


Figure: Modelling Exceptions

14. PROCESS AND THREADS

- A process is a heavyweight flow that can execute concurrently with other processes.
- A thread is a lightweight flow that can execute concurrently with other threads within the same process.
- An active object is an object that owns a process or thread and can initiate control activity.
- An active class is a class whose instances are active objects.
- Graphically, an active class is rendered as a rectangle with thick lines. Processes and threads are rendered as stereotyped active classes.



❖ **Flow of Control**

In a sequential system, there is a single flow of control. i.e, one thing, and one thing only, can take place at a time.

In a concurrent system, there is multiple simultaneous flow of control i.e, more than one thing can take place at a time.

❖ **Classes and Events**

- Active classes are just classes which represents an independent flow of control
- Active classes share the same properties as all other classes.
- When an active object is created, the associated flow of control is started; when the active object is destroyed, the associated flow of control is terminated
- *Two standard stereotypes* that apply to active classes are, <<***process***>> – Specifies a heavyweight flow that can execute concurrently with other processes. (heavyweight means, a thing known to the OS itself and runs in an independent address space) <<***thread***>> – Specifies a lightweight flow that can execute concurrently with other threads within the same process (lightweight means, known to the OS itself.)
- All the threads that live in the context of a process are peers of one another

❖ **Communication**

- In a system with both active and passive objects, there are *four possible combinations of interaction*
- *First*, a message may be passed from one passive object to another
- *Second*, a message may be passed from one active object to another
- In *inter-process communication* there are two possible styles of communication. *First*, one active object might synchronously call an operation of another. *Second*, one active

object might asynchronously send a signal or call an operation of another object

- A synchronous message is rendered as a full arrow and an asynchronous message is rendered as a half arrow
- *Third*, a message may be passed from an active object to a passive object
- *Fourth*, a message may be passed from a passive object to an active one

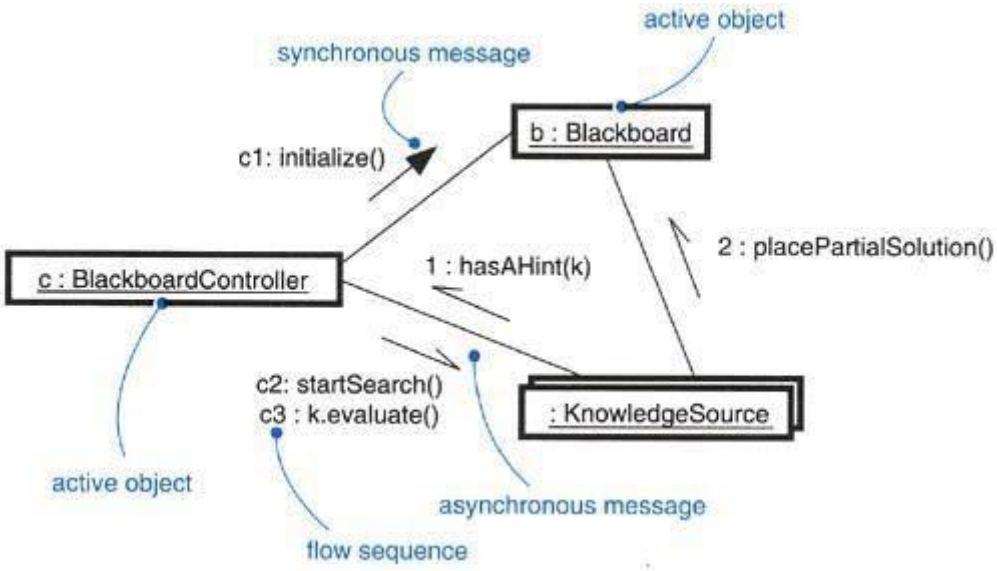


Figure: Communication

❖ Synchronization

- Synchronization means arranging the flow of controls of objects so that mutual exclusion will be guaranteed.
- In object-oriented systems these objects are treated as a critical region
- *Three approaches* are there to handle synchronization:

1. *Sequential* – Callers must coordinate outside the object so that only one flow is in the object at a time.
2. *Guarded* – multiple flow of control is sequential with the help of object's guarded operations. in effect it becomes sequential.
3. *Concurrent* – multiple flow of control is guaranteed by treating each operation as atomic

- Synchronization are rendered in the operations of active classes with the help of constraints

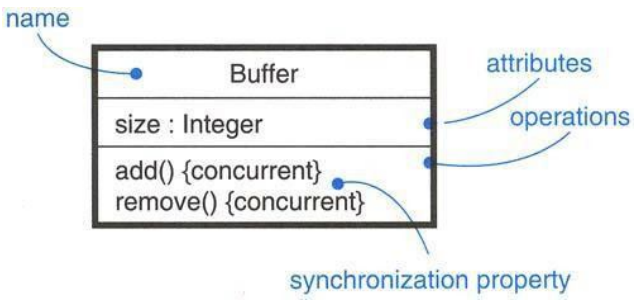


Figure: Synchronization

❖ **Process Views**

- The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.
- This view primarily addresses the performance, scalability, and throughput of the system.

Modelling Multiple Flows of Control

1. Identify the opportunities for concurrent action and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over engineer the process view of your system by introducing too much concurrency.
2. Consider a balanced distribution of responsibilities among these active classes, then examine the other active and passive classes with which each collaborates statically. Ensure that each active class is both tightly cohesive and loosely coupled relative to these neighbouring classes and that each has the right set of attributes, operations, and signals.
3. Capture these static decisions in class diagrams, explicitly highlighting each active class.
4. Consider how each group of classes collaborates with one another dynamically. Capture those decisions in interaction diagrams. Explicitly show active objects as the root of such flows. Identify each related sequence by identifying it with the name of the active object.
5. Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.
6. Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

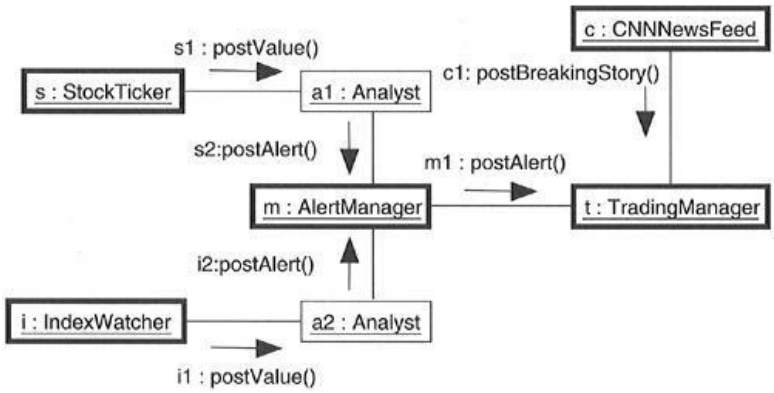


Figure: Modelling Flows of Control

Modelling Inter Process Communication (IPC)

To model inter process communication,

1. Model the multiple flows of control.
2. Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
3. Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
4. Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations.

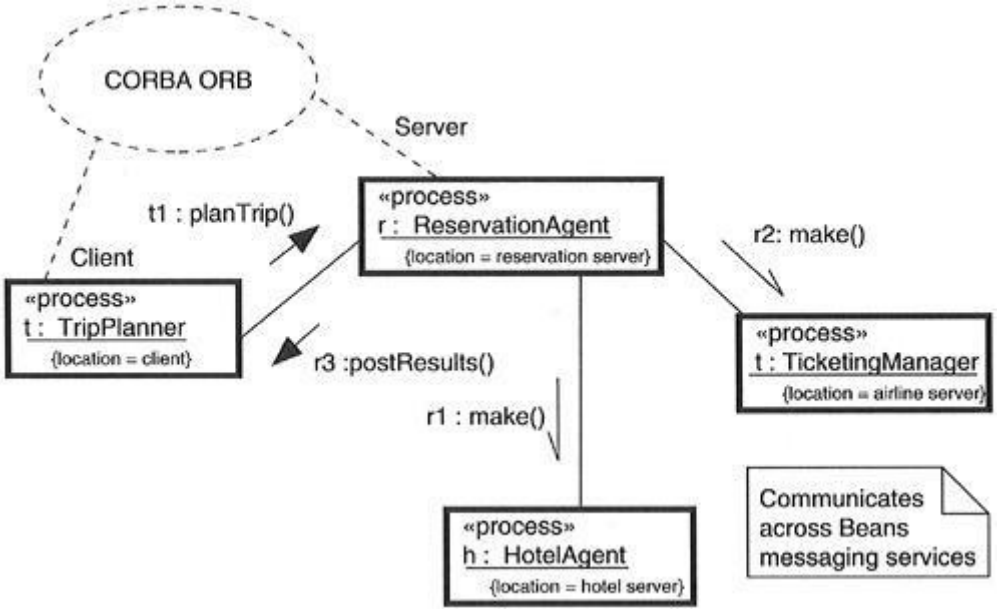


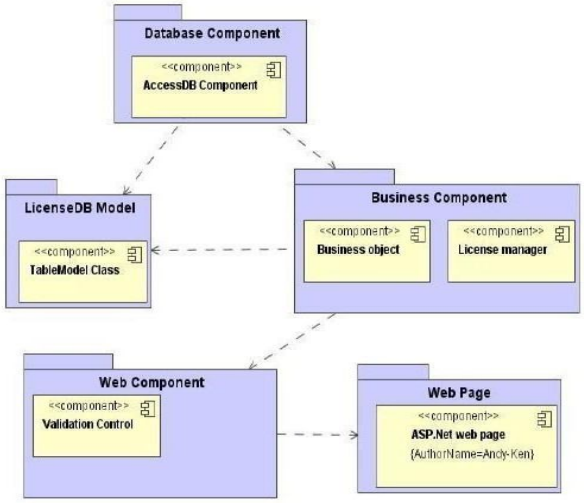
Figure: Modelling Inter Process Communication

15. UML ARCHITECTURAL MODELLING

COMPONENT DIAGRAM

- It shows structural replaceable parts of the system. Its main components are:
  - components
  - interfaces
  - packages

- Component diagrams are different in terms of nature and behaviour. Component diagrams are used to model physical aspects of a system.
- Physical aspects are the elements like physical components, executables, libraries, files, documents etc**.** which resides in a node.
- So component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.
- Figure below shows a component diagram



Component:

- A component is a physical, replaceable part that conforms to and provides the realization of a set of interfaces.
- A component:
  - Encapsulates the implementation of classifiers residing in it.
  - Does not have its own features, but serves as a mere container for its elements
  - Are replaceable or substitutable parts of a system

PURPOSE

- Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.
- Component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files etc.

- Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

- A single component diagram cannot represent the entire system but a collection of diagrams are used to represent the whole.
- The purpose of the component diagram can be summarized as:

  - Visualize the components of a system.
  - Construct executables by using forward and reverse engineering.
  - Describe the organization and relationships of the components.

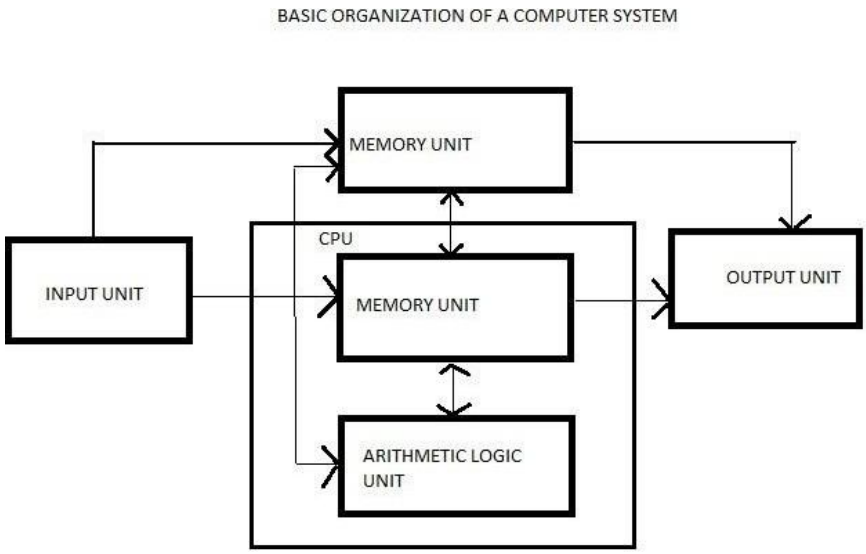
How to draw Component Diagram?

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing using tools.
- Use notes for clarifying important points.

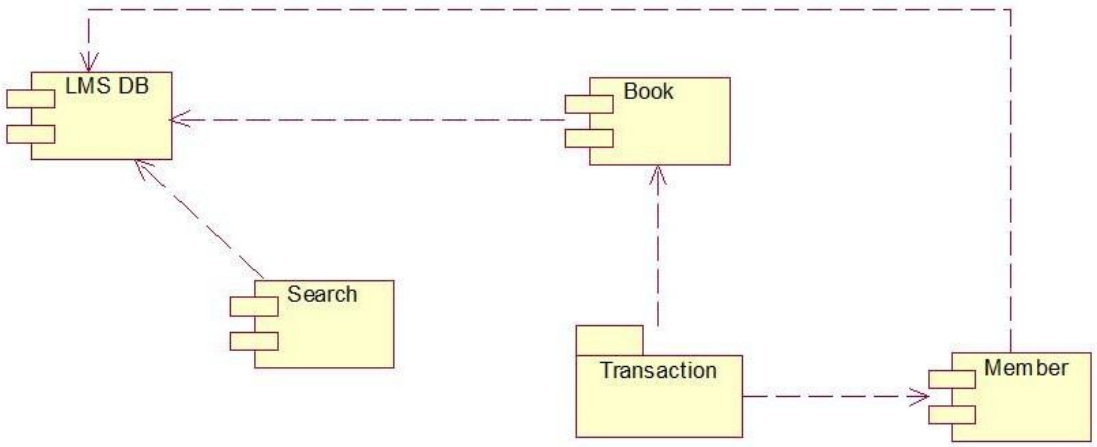
Where to use Component Diagrams?

- Model the components of a system.
- Model database schema.
- Model executables of an application.
- Model system's source code.

COMPONENT DIAGRAM FOR COMPUTER



COMPONENT DIAGRAM FOR LMS



DEPLOYMENT DIAGRAM

- Deployment modelling is a specialized type of structural modelling concerned with modelling the implementation environment of a system.
- In contrast to modelling the components of a system, a deployment model shows you the external resources that those components require.
- Deployment modelling is applied during design activities to determine how deployment activities will make the system available to its users; that is, to determine the elements of the system on which deployment activities will focus
- Deployment
    1. shows configuration of run-time processing nodes and the components hosted on them
    2. addresses the static deployment view of an architecture
    3. is related to component diagram with nodes hosting one or more components
    4. essentially focus on a system‘s nodes, and include: nodes, dependencies and associations relationships, components, packages

Nodes and Components

Components

- participate in the execution of a system.
- represent the physical packaging of otherwise logical elements

Nodes

- execute components
- represent the physical deployment of components

❖ The relationship deploys between a node and a component can be shown using a dependency relationship.

❖ Nodes can be organized:

- ➢ In the same manner as classes and components.
- ➢ By specifying dependency, generalization, association, aggregation, and realization relationships among them.

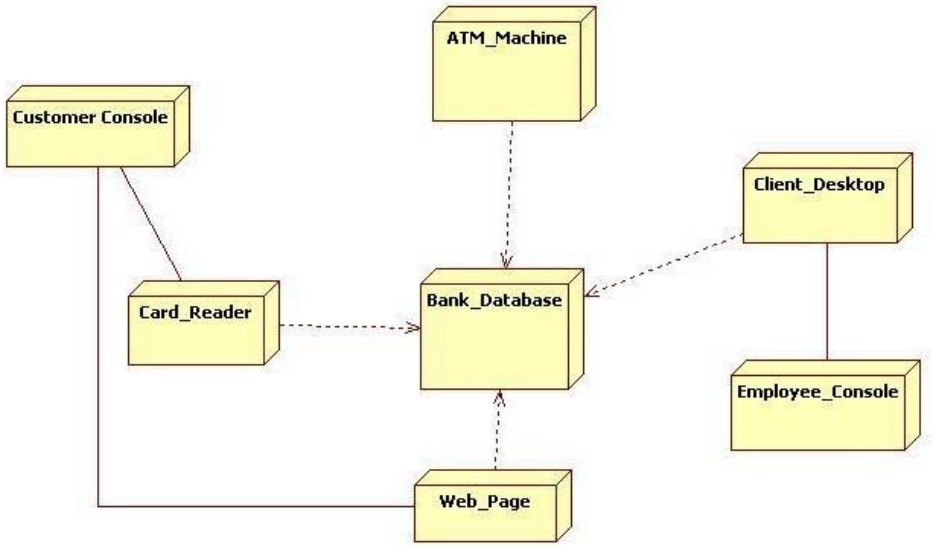
- The most common kind of relationship used among nodes is an association representing a physical connection among then.

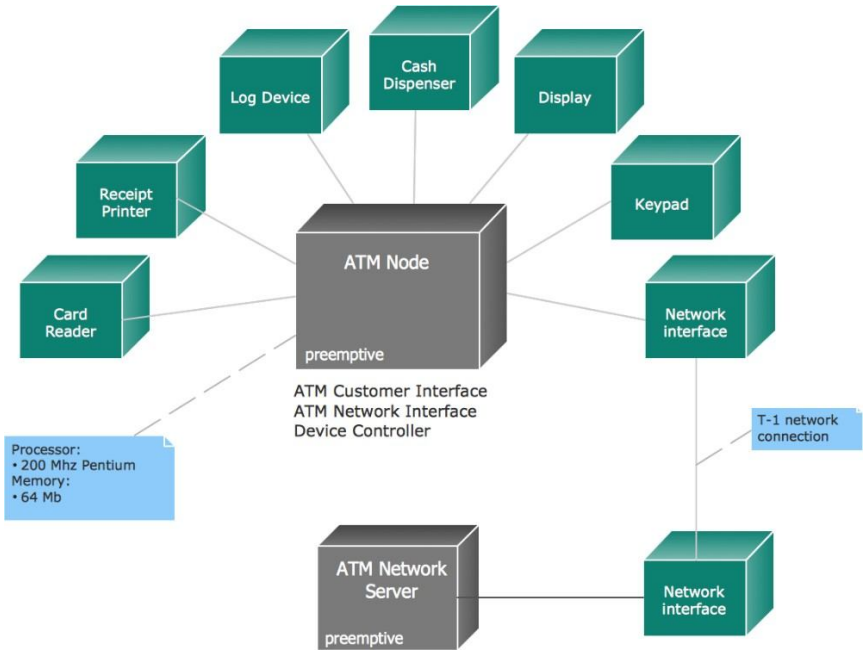
- ❖ A processor is a node that has processing capability. It can execute a component.
- ❖ A device is a node that has no processing capability (at least at the level of abstraction showed).

Modelling Nodes Procedure

1. Identify the computational elements of the system‘s deployment view and model each as a node.
2. Add the corresponding stereotype to the nodes
3. Consider attributes and operations that might apply to each node.

DEPLOYMENT DIAGRAM OF ATM





16. SOFTWARE ARCHITECTURE

Definition:

Software architecture of a program or software system consist of various components (modules) in that system, externally visible properties of those components and the inter-relationship between those components.

- ✓ Software architectures provide high-level abstractions for representing structure, behavior, and key properties of a software system. These abstractions are useful in describing to various stakeholders complex, real-world problems in an understandable manner.
- ✓ Software architectures are described in terms of components, connectors, and configurations. Architectural components describe the computations and state of a system; connectors describe the rules of interaction among the components; finally, configurations define topologies of components and connectors.

DESIGN FRAMEWORKS

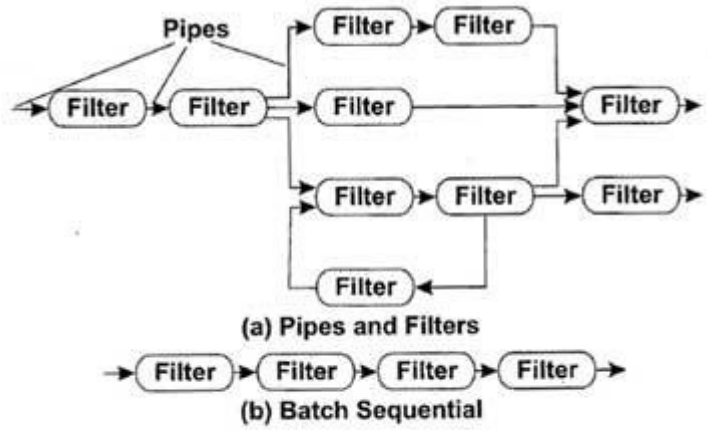
Architectural frameworks provide support for implementing, deploying, executing, and evolving software architectures. A framework is a skeletal group of software modules that may be tailored for building domain-specific applications, typically resulting in increased productivity and faster time-to-market.

DESIGN PATTERN

- In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns can speed up the development process by providing tested, proven development paradigms. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.
- Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioural patterns, and described using the concepts of delegation, aggregation, and consultation.
- The documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution.
- Some of the commonly used architectural patterns are data-flow based architecture, object – oriented architecture, layered system architecture, data-centred architecture and call and return architecture.

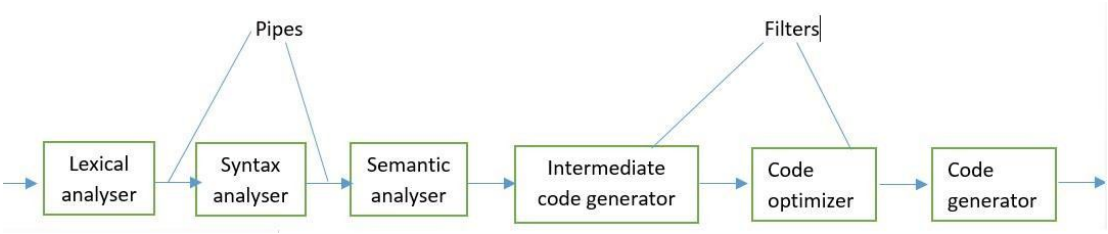
DATA FLOW ARCHITECTURE / PIPE AND FILTER STYLE

- Data flow based architecture is mainly used in systems that accepts some inputs and transform it into desired outputs by applying a series of transformations.
- Each component known as filter, transforms the data and sends this transformed data to other filters for further processing using the connector, known as pipe. Each filter works as an independent entity. A pipe is a unidirectional channel which transports the data received on one end to the other end. Pipe does not change the data in anyway; it merely supplies data to the filter on the receiver end.



Data-flow Architecture

- In most cases, data flow architecture degenerates a batch sequential system. In this system, a batch of data is accepted as input and then a series of sequential filters are applied to transform this data.
- Example of this style is a Compiler which accepts High Level Language and returns a machine dependent form.



Advantages of Data- Flow architecture:

- It supports reusability
- It is maintainable and modifiable.
- It supports concurrent execution

Disadvantages of Data –Flow architecture

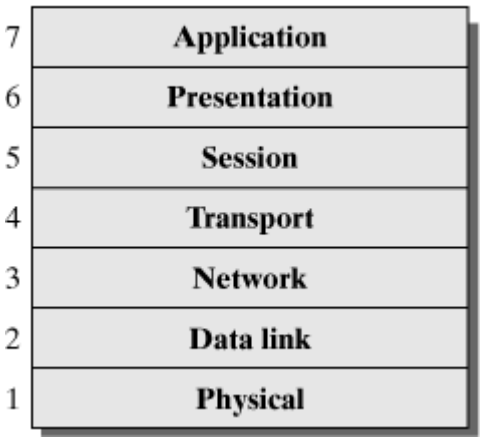
- It often degenerates to batch sequential system.

- It does not provide enough support for applications requires user interaction
- It is difficult to synchronize two different, but related streams.

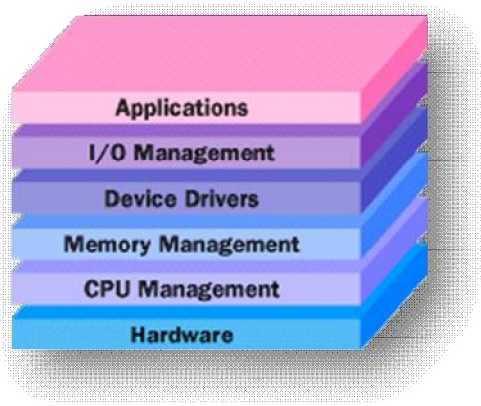
LAYERED ARCHITECTURE

- In layered architecture, entire system is divided into various layers such that, each layer performs a well- defined set of operations. These are layers are arranged in hierarchical manner, each one built upon the one below it.
- Each layer provides a set of services to the layer above it and acts as a client to the layer below it.
- The interaction between layers is provided through protocols (connectors) that define a set of rules to be followed during interaction.
- Example of layered architectural style is ISO OSI Internet Protocol Suite, Operating system layers etc.

ISO/OSI reference model



Operating system function-layers

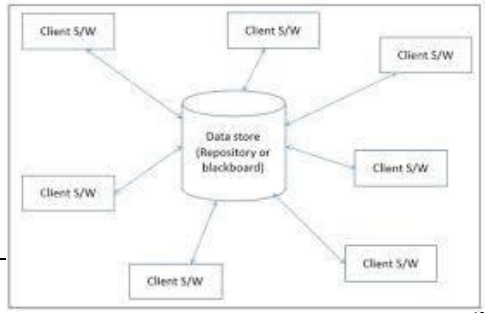


REPOSITORY VIEW / DATA CENTERED VIEW

- A data centered architecture has two distinct components: a central data store (central repository) and a collection of client software (data accessors).
- The data store (database or file) represents the current state of the data and the client software performs several operations like add, delete, update etc., on the data stored in the central repository.
- It has two views:

Repository view: The client who updates the central repository can determine the access permission for various other clients, to view the same updated data.

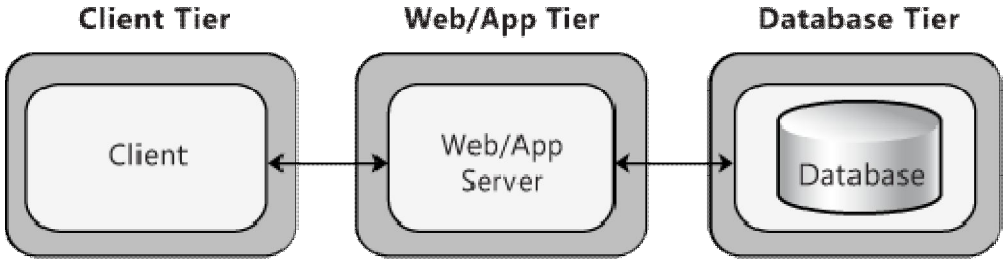
Black board view: Here the data store acts like a black board system in which the data store is transformed into a black board that notifies the client software when the data changes / updates.



- ADVANTAGES:
  - Clients operate independently of one another.
  - Data repository is independent of clients.
  - It facilitates scalability.
  - Supports modifiability.

CLIENT – SERVER STYLE

- It is a variant of main program and subroutines concept, but the clients and servers.
- A server provides different services. A client uses services as (remote) subroutines from one or more servers.
- A general form of this architecture is n – tier architecture. In this style, a client sends request to a server. But in order to service the request, the server sends some request to another server. That is the server acts as a client for the next tier. This hierarchy can continue for some levels, providing a n-tier system.
- A common example is 3 –tier architecture.

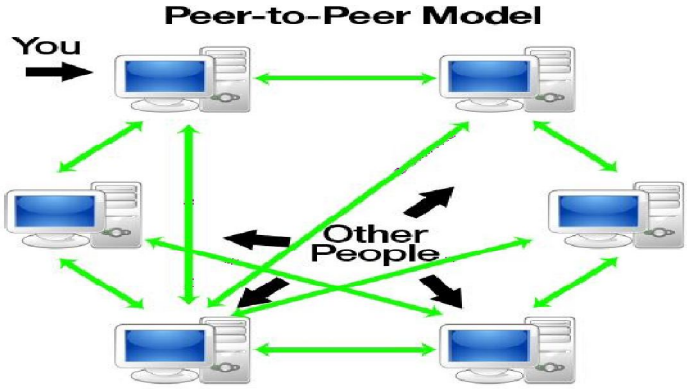


CALL AND RETURN ARCHITECTURE

- This architecture enables to achieve a program structure that is relatively easy to modify and scale. The categories are Main program/subprogram architecture and Remote Procedure Call (RPC) architectures.

PEER – PEER ARCHITECTURE / OBJECT ORIENTED VIEW

- Peer to Peer (P2P) are similar to client – server but all processes can be act as clients and servers.
- It is more flexible but complicated design.
- There will be chance to occur dead lock or starvation.



17. ARCHITECTURAL DESCRIPTION LANGUAGE (ADL)

Architecture description language (ADL)

- ADLs are formal languages that describe or represent software architectures.
- An ADL must explicitly model components, connectors and their configurations.
- ADL must provide tool support for architecture-based development and evolution.
- There is a large variety in ADLs developed by either academic or industrial groups.
- Examples of some ADL's are ACME, ADML, Rapide, Wright, Unicon, Aseop, MetaH, AADL, Darwin, etc.

OVERVIEW OF ADL

- Architecture description languages (ADLs) are emerging as the notation for architecture models. ADLs use graphics and text to express architectural information as shown below.
- ADLs are often supported by tools for creation, modification, browsing, simulation, and analysis.
- ADLs vary widely in the architecture styles they support and that forms of analyses they permit. Like other tools, there is no one ADL that best fits all possible situations.
- ADLs support the routine use of existing designs and components in new application systems.
- ADLs support the evaluation of an application system before it is built.

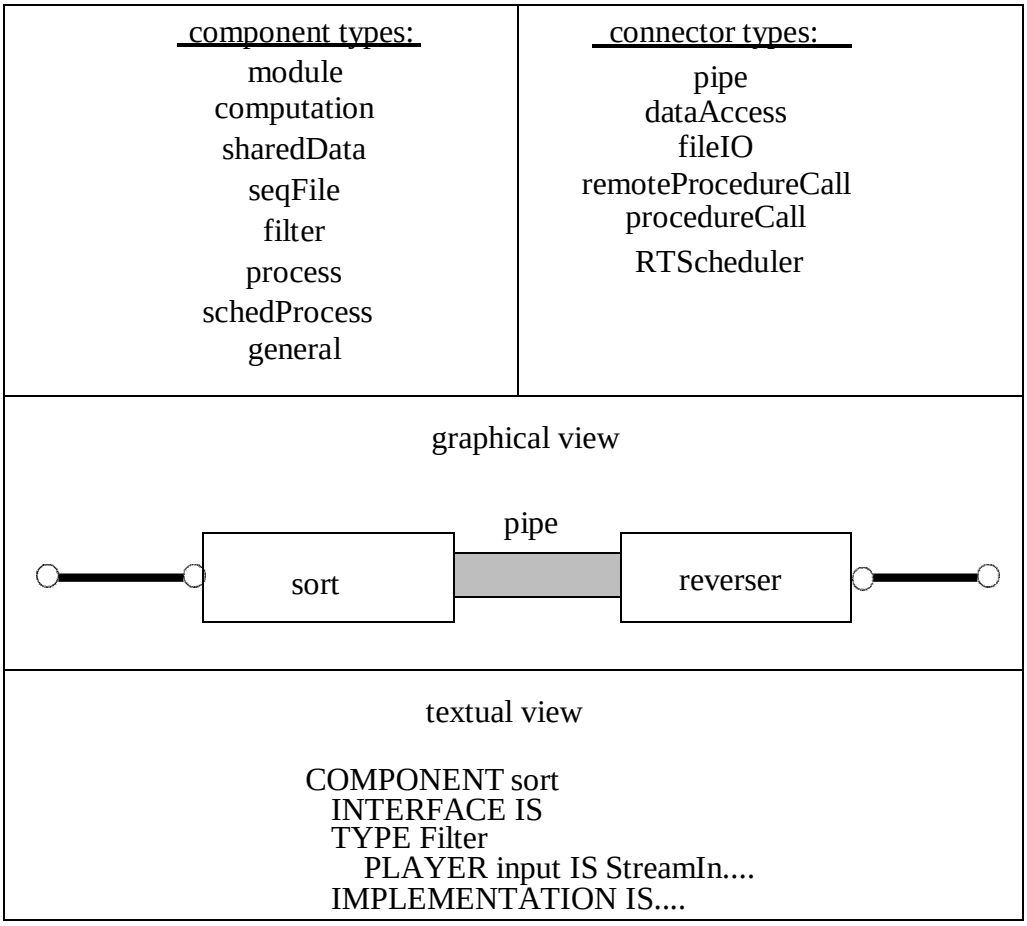
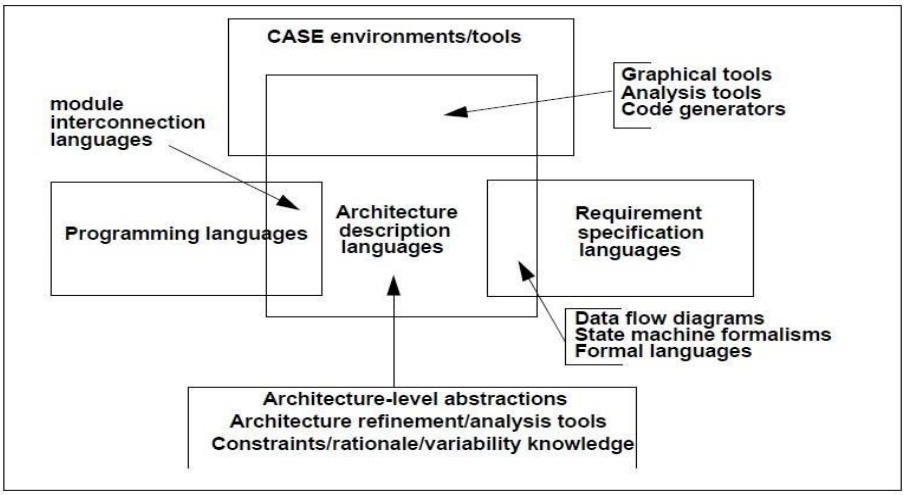


Figure 1: Example ADL - UniCon

ADL Framework



Advantages of ADL

- ADLs are a formal way of representing architecture.
- ADLs reduces ambiguity in design
- ADLs are intended to be both human and machine readable
- ADLs support describing a system at a higher level than previously possible
- ADLs permit analysis and assessment of architectures, for completeness, consistency, ambiguity, and performance
- ADLs can support automatic generation of software systems

Disadvantages of ADL

- There is no universal agreement on what ADLs should represent, particularly as regards the behaviour of the architecture.
- Increased development cost.
- Representations currently in use are relatively difficult to parse and are not supported by commercial tools
- Most ADLs tend to be very vertically optimized toward a particular kind of analysis

ADLs have in common:

- Graphical syntax with often a textual form and a formally defined syntax and semantics
- Features for modelling distributed systems
- Little support for capturing design information, except through general purpose annotation mechanisms
- Ability to represent hierarchical levels of detail including the creation of substructures by instantiating templates

ADLs differ in their ability to:

- Handle real-time constructs, such as deadlines and task priorities, at the architectural level
- Support the specification of different architectural styles. Few handle object oriented class inheritance or dynamic architectures
- Support the analysis of the architecture

- Handle different instantiations of the same architecture, in relation to product line architecture.

ADL ATTRIBUTES

- Architecture creation: Creates the architecture of the system in both graphical view as well as in textual form.
- Architecture validation: Checks the consistency and correctness of the architecture being created. It also checks the semantics and syntax of the particular ADL
- Architecture refinement: Based on the validation result, the architecture is refined with suitable modifications in the graphical and textual views.
- Architecture analysis: The architecture is analysed for time and resource economy (e.g., throughput, memory utilization). Then it is analysed for functionality (e.g. completeness, correctness, security, interoperability). After that the model is analysed for maintainability (e.g., expandability, correctness) and portability (e.g., independence from hardware or software environments). Finally the model is analysed for its reliability or usability
- Application building: The supporting tools for building a compile-able (or executable) software system from the modelled specific system design is identified. The application is built using suitable ADL supporting tools.

18. RATIONAL UNIFIED PROCESS (RUP) / UNIFIED PROCESS (UP)

User Process (UP)

UP is an open software engineering process (SEP) that models the who, when and what of developing software. The basis of UP approach was “divide and conquer” also known as component-based development.

UP elements
UP have three Key elements. They are:

- Iterative and incremental
- use case and risk driven
- Architecture centric

UP is an iterative and incremental process

UP aims to build robust system architecture incrementally. The iterative aspect of UP is to break a large software development project down into a number of smaller “mini projects” which are easier to manage and to complete successfully. Each of these “mini projects” is an iteration. Technical risks are assessed and prioritized early and are revised during each iteration.

Use Case and risk driven
Use cases are used for:
1) identify users and their requirements
2) aid in the creation and validation of the architecture
3) help produce definitions of test cases and procedures
4) direct the planning of iterations
5) drive the creation of user documentation
6) direct the deployment of the system
7) synchronize the content of different models
8) drive traceability throughout models

Architecture-Centric
With the iterative and incremental approach different development activities are done concurrently the system’s architecture ensures that all parts fit together

Iteration
Iterations are subprojects or mini projects of a large software development project. It contains all the elements of a normal software development project i.e – Planning, Analysis and design, Construction, Integration and test, an internal or external release. Baselines are the results of each Iterations.

Baselines
Baseline comprises a partially complete version of the final system and any associated project documentation. It is an internal (or external) release of the set of reviewed and approved artifacts

generated by that iteration. Each baseline,

- provides an agreed basis for further review and development
- can be changed only through formal procedures of configuration and change management.

Increment

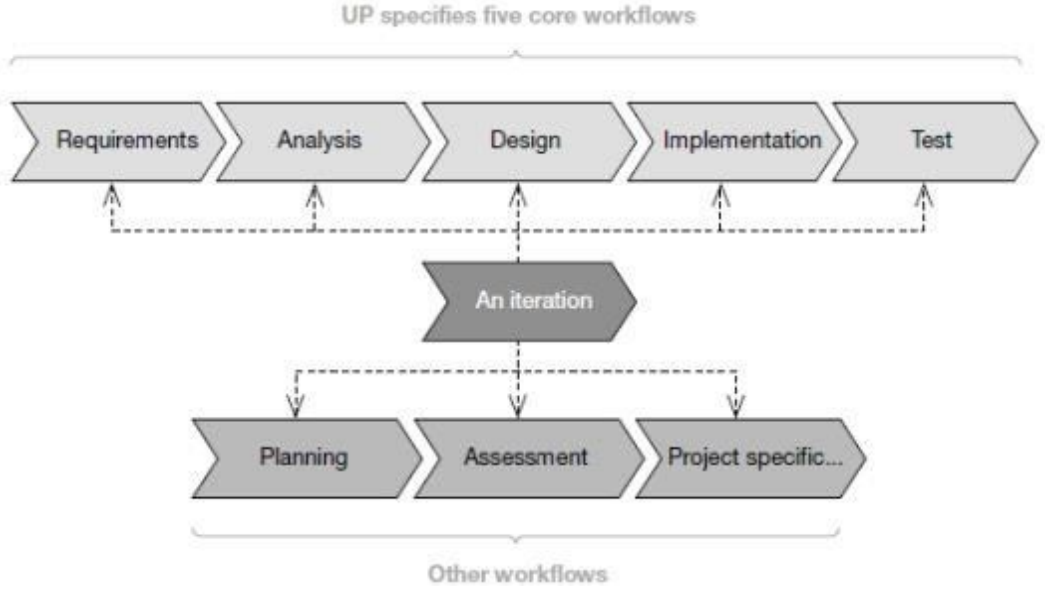
Increments are the difference between two consecutive baselines. They constitute a step toward the final delivered system. Because of the use of *increment* and *baselines* UP is known as an iterative and incremental lifecycle.

Iteration workflows

UP have *five core workflows.* The five core workflows are:

- Requirements – capturing what the system should do
- Analysis – refining and structuring the requirements
- Design – realizing the requirements in system architecture
- Implementation – building the software
- Test – verifying that the implementation works as desired.

Figure below represents the iteration workflows.



UP structure

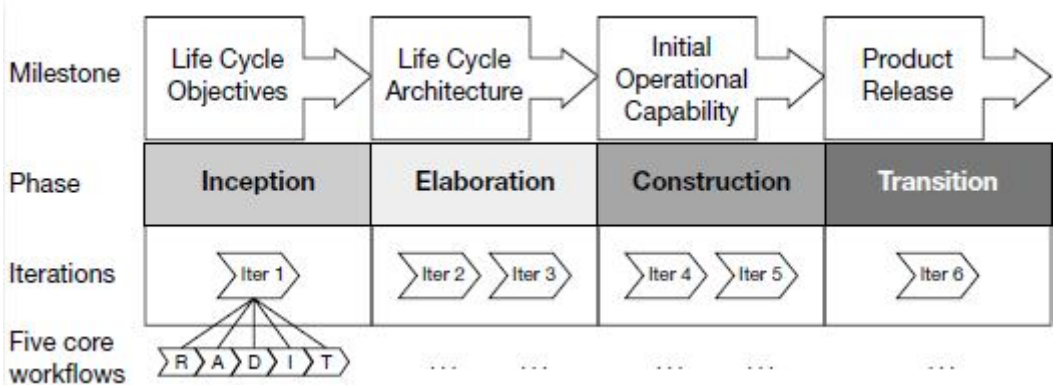


Figure: UP structure

UP *has four phases* – Inception, Elaboration, Construction, and Transition – each of which ends with a major milestone. Within each phase there are one or more iterations, and in each iteration we can execute the five core workflows and any extra workflows.

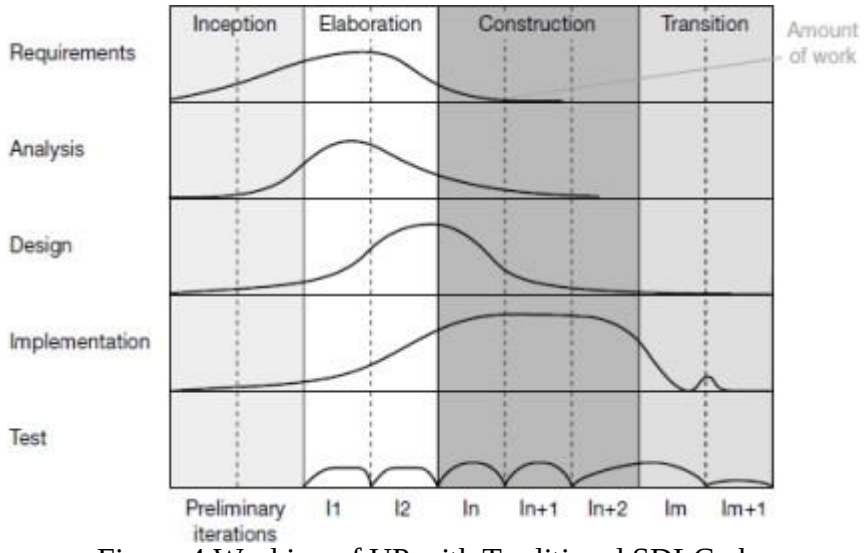


Figure:4 Working of UP with Traditional SDLC phases

- The curves show the relative amount of work done in each of the five core workflows, as the project progresses through the phases.
- The amount of work done in each core workflow varies according to the phase.one of the great feature of UP is that it is a goal-based process rather than a deliverable – based process.
- Each phase ends with a milestone that consists of a set of conditions of satisfaction, and these conditions may involve the creation of a particular deliverable or not depending on the specific needs of your project.

UP PHASES

Every phase has *a goal, a focus* of activity with one or more core workflows emphasized, and *a milestone*.

Inception phase:

- Inception is about initiating the project.
- Inception goals includes – establishing feasibility, creating a business case, capturing essential requirements, identifying critical risks.
- Inception focus is on requirements and analysis workflows. Some design and implementation might also be done if it is decided to build a technical, or proof of concept, prototype.
- Inception Milestone makes use of goal-oriented approach which sets certain goals that must be achieved for the milestone to have been reached. The milestone for Inception is the Life Cycle Objectives. The Life Cycle Objectives includes defining system scope with the help

of use case models, capturing Key requirements, etc.

Elaboration phase:

- Elaboration is about creating a partial but working version of the system – an executable architectural baseline.
- Elaboration goals includes

    - Create an executable architectural baseline
    - Develop suitable algorithms and design patterns.
    - Refine the Risk Assessment
    - Define quality attributes
    - Capture use cases to 80% of the functional requirements
    - Create a detailed plan for the construction phase
    - Formulate a bid that includes resources, time, equipment, staff and cost.

- Elaboration milestone is a Life Cycle Architecture. The Life Cycle Architecture includes UML Static Model, UML Dynamic Model, UML Use Case Model, revising risk assessment through, etc..

Construction Phase:

- Converting the design into a system by using a suitable high level programming language
- Construction evolves the executable architectural baseline into a complete, working system.
- The Goal of Construction is to complete all requirements, analysis and design, and to evolve the architectural baseline generated in Elaboration phase into the final system. analysis – finish the analysis model
- Unit testing is performed to ensure the functionality of each implemented module.

Transition Phase:

- Transition is about deploying the completed system into the user community.
- Transition Goals includes

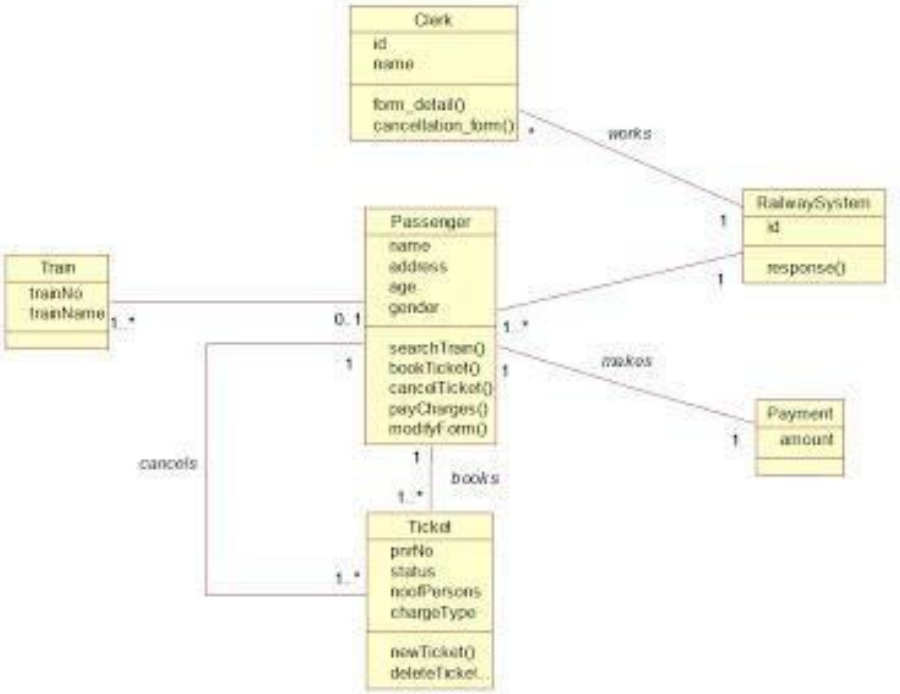
    - correct defects
    - prepare the user site for the new software
    - tailor the software to operate at the user site
    - modify the software if unforeseen problems arise
    - create user manuals and other documentation
    - provide user consultancy
    - conduct a post project review

- Transition milestone is Product Release. Here beta testing, acceptance testing and defect repair are finished and the product is released and accepted into the user community.

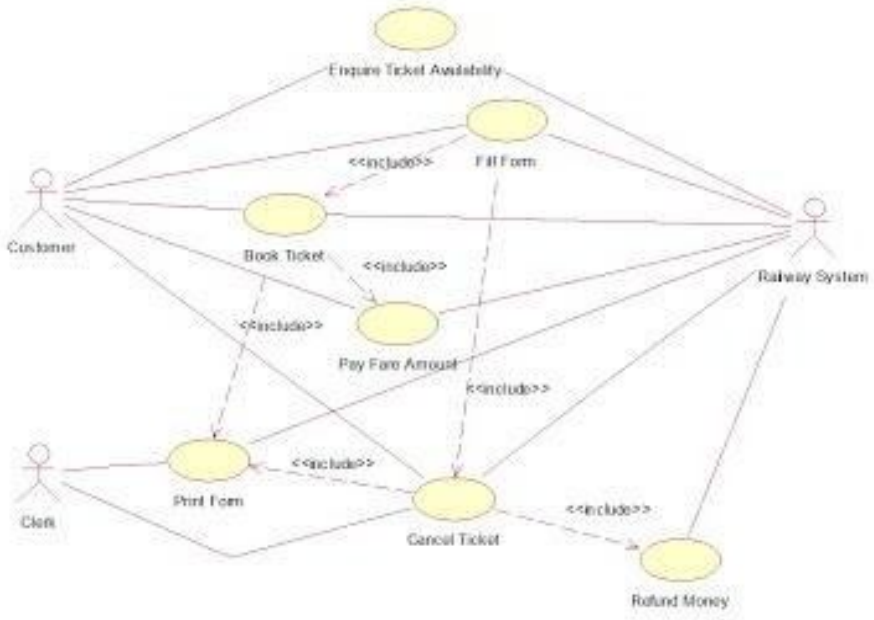
19. SAMPLE UML DIAGRAMS

Railway Reservation System UML Diagrams

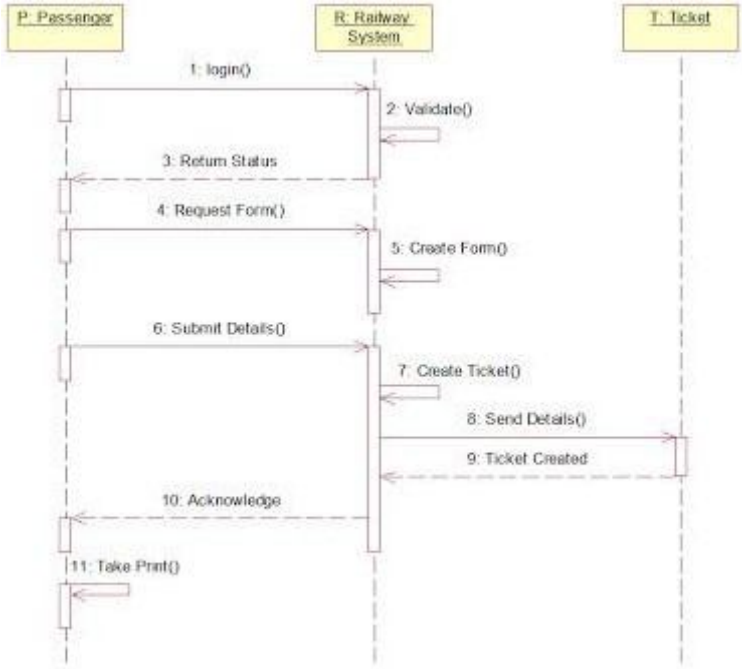
Class Diagram



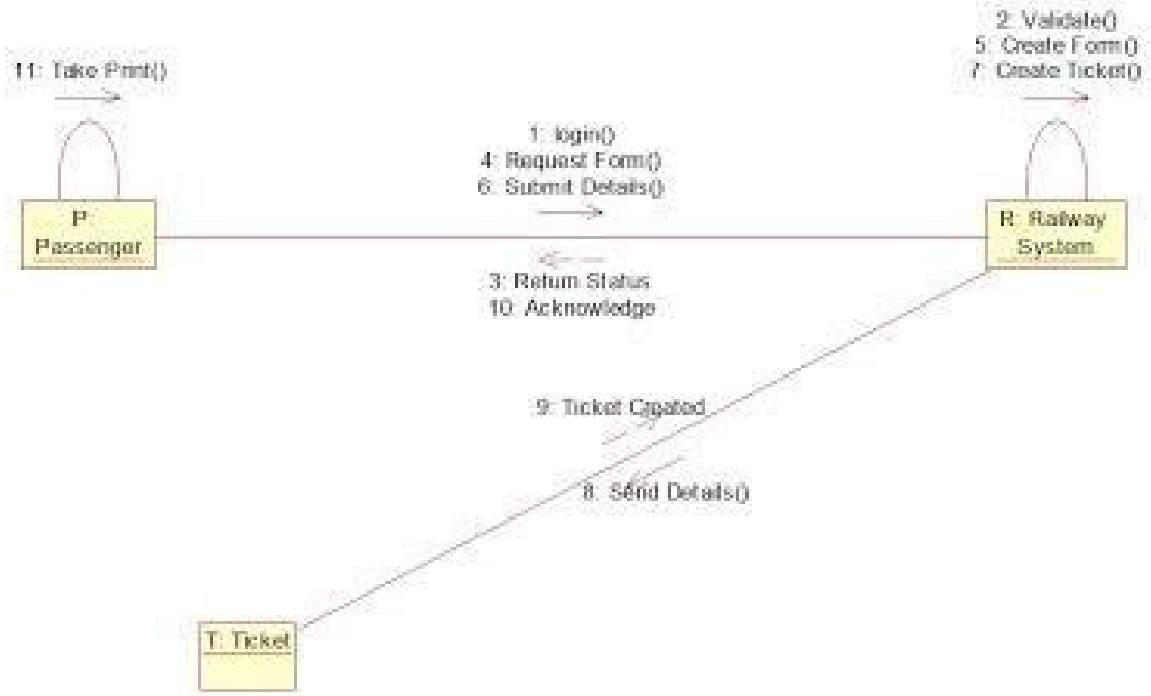
Use Case Diagram



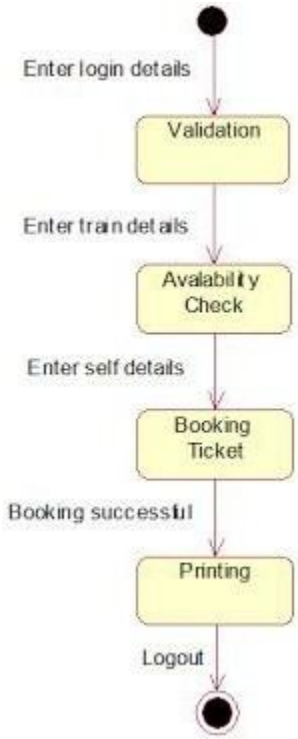
Sequence Diagram



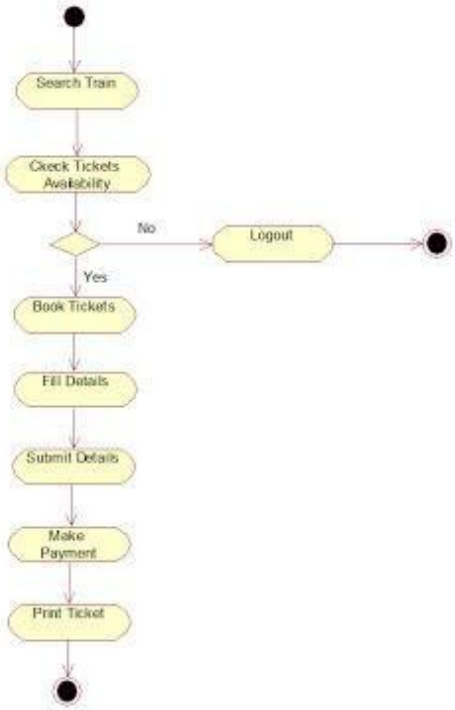
Collaboration Diagram



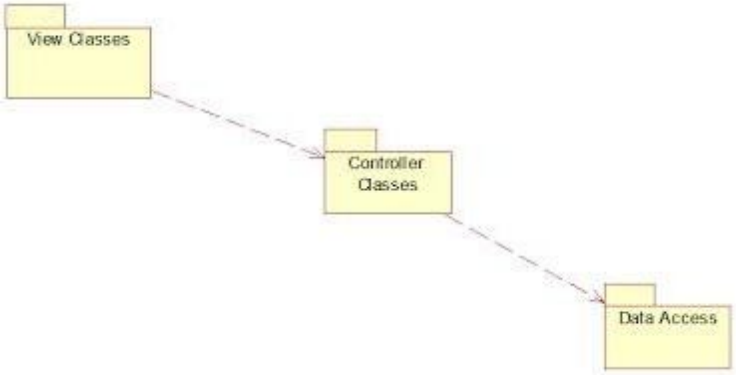
State Chart Diagram



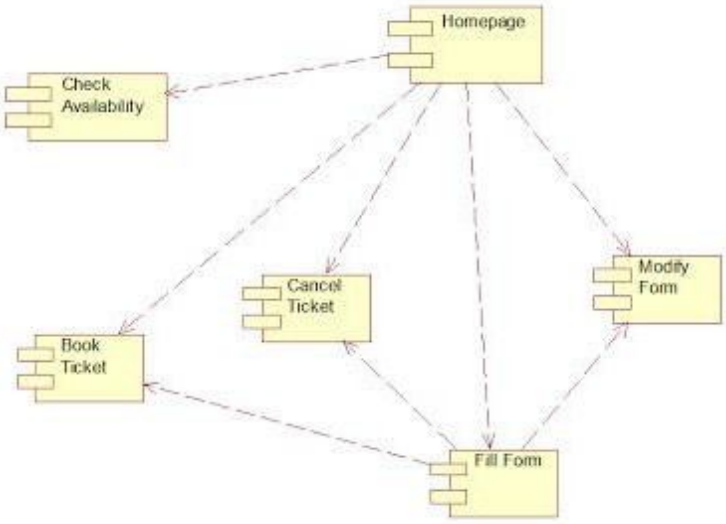
Activity Diagram



Component Diagram

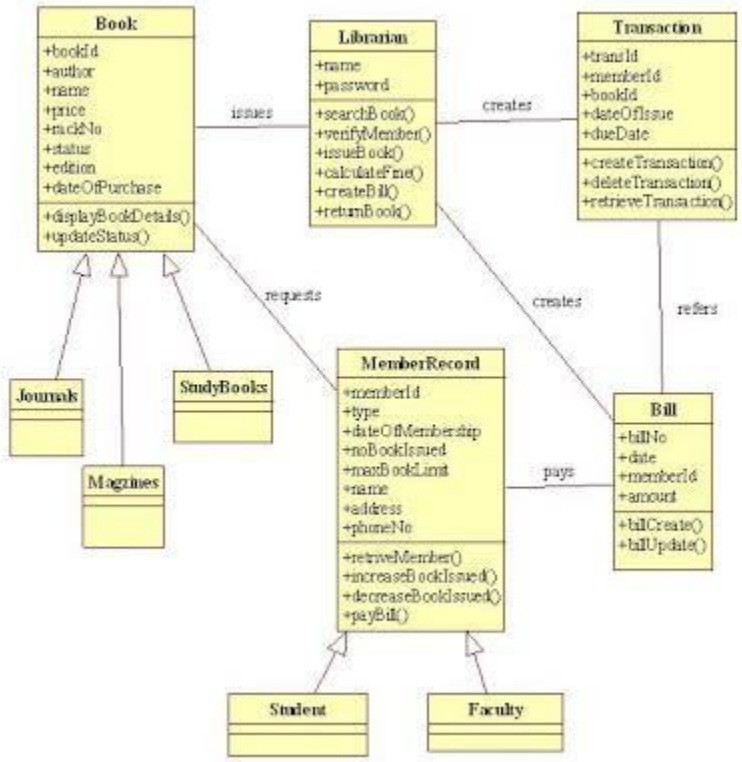


Component Diagram - View Classes Package

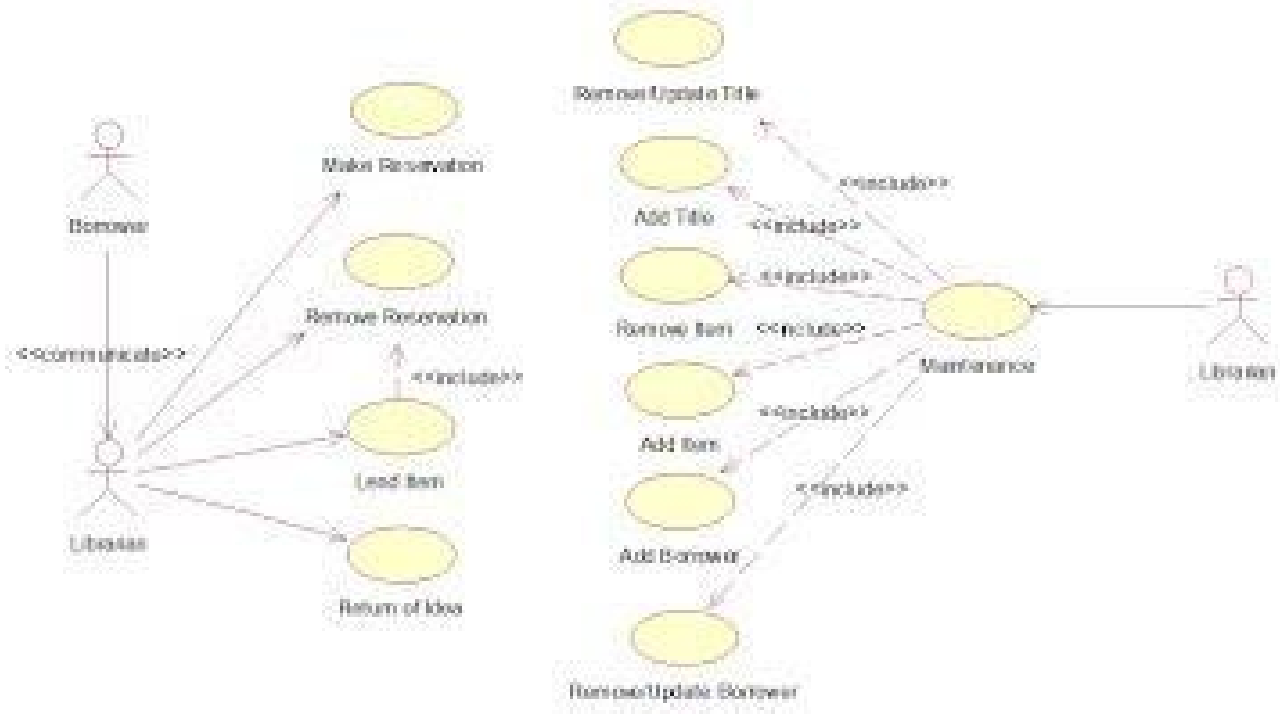


Library Management System UML Diagrams

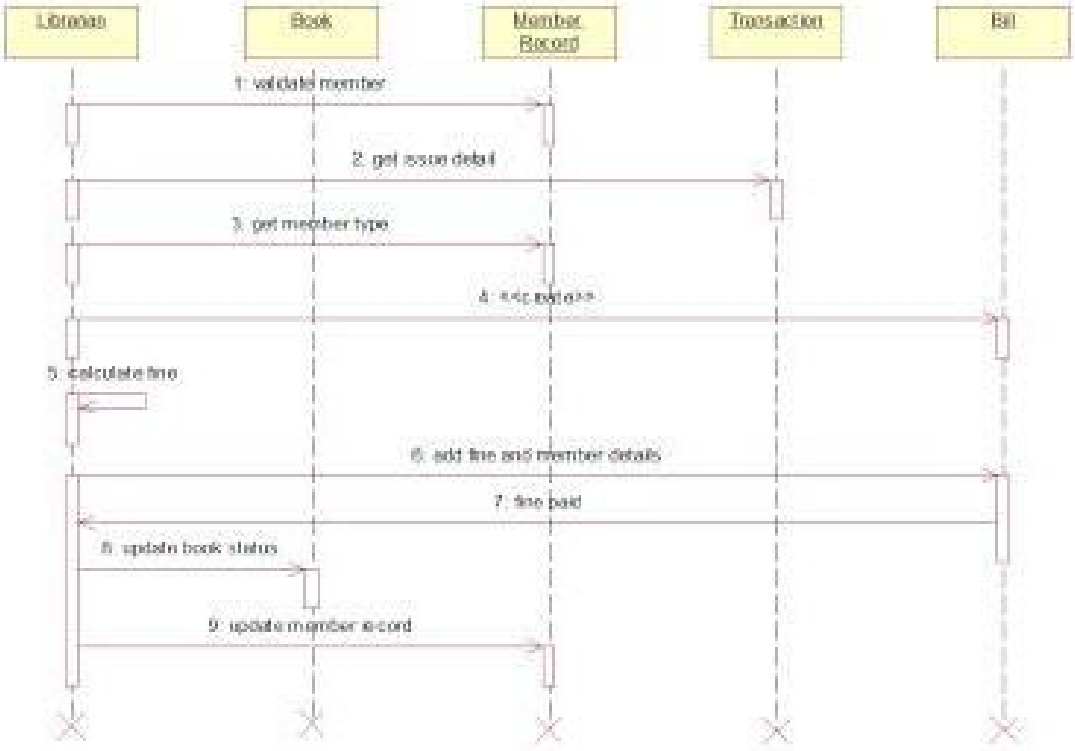
Class Diagram:



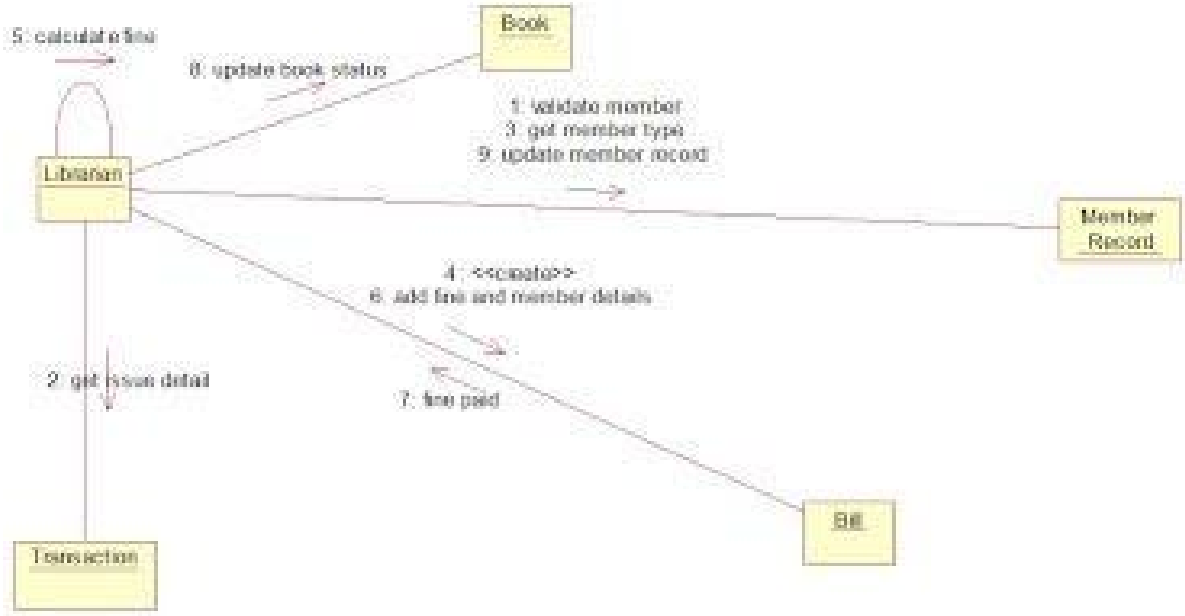
Use Case Diagram:



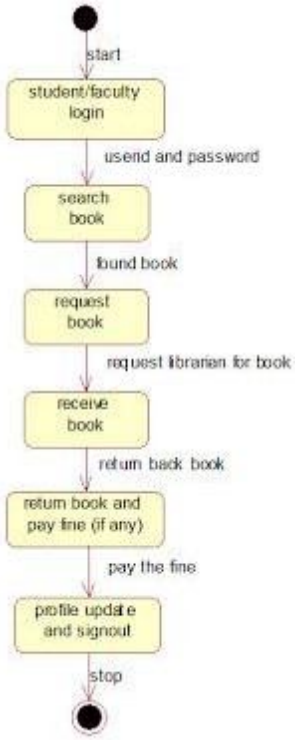
Sequence Diagram:



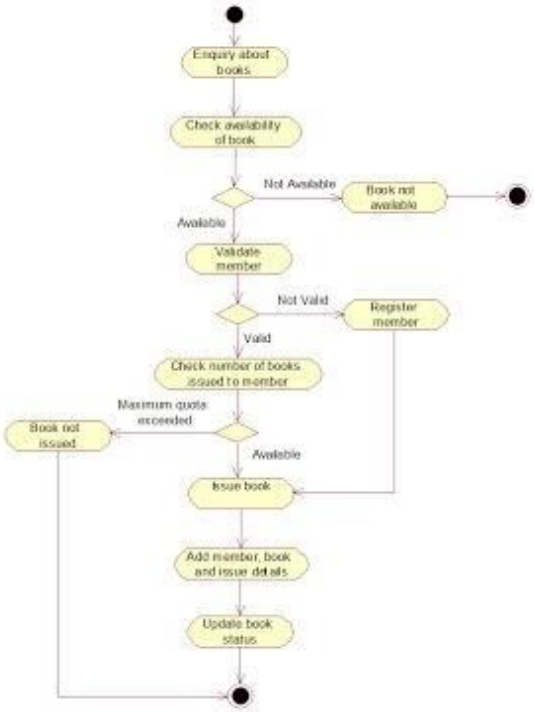
Collaboration Diagram:



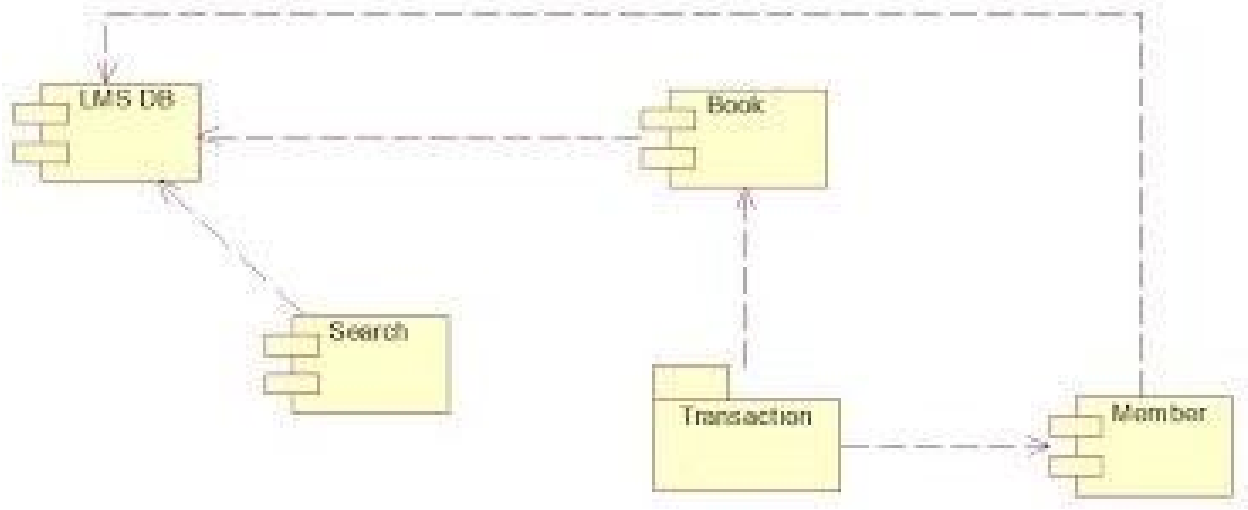
State Chart Diagram:



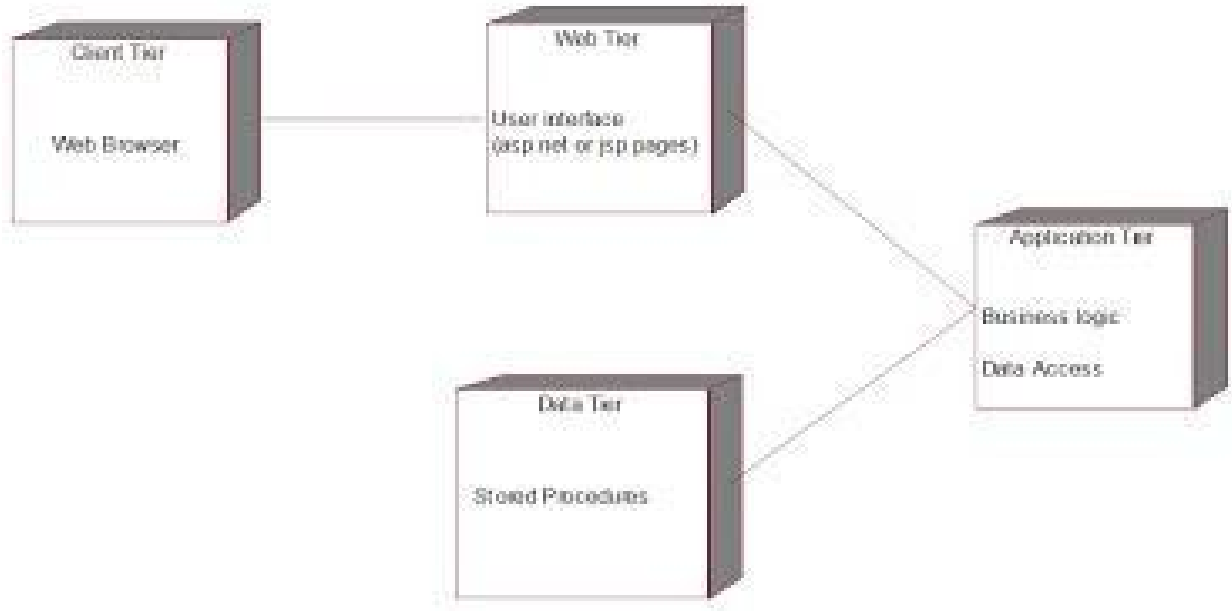
Activity Diagram:



Component Diagram:

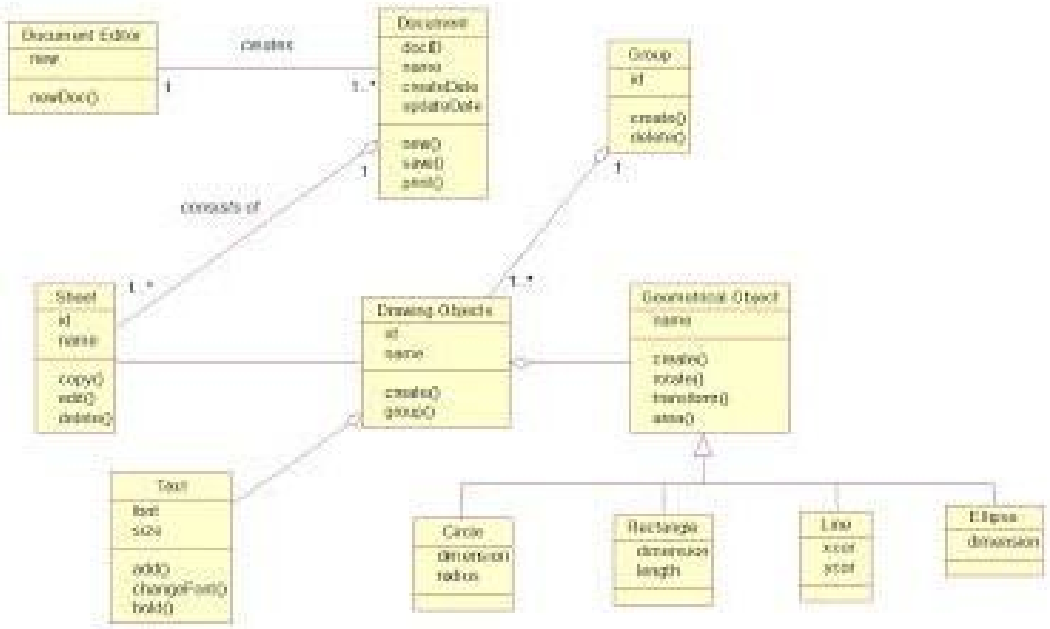


Deployment Diagram:

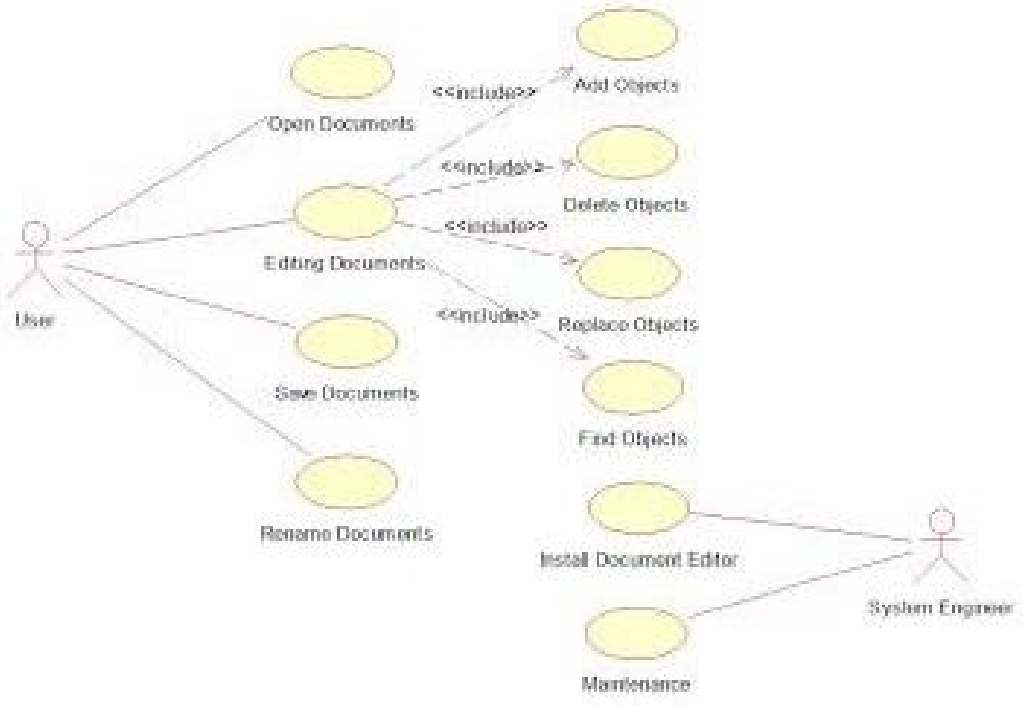


Document Editor UML Diagrams

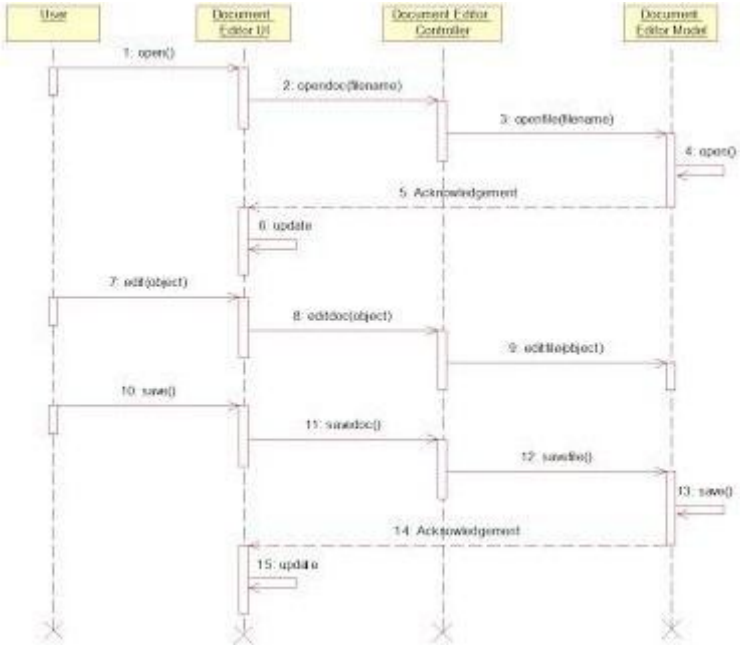
Class Diagram



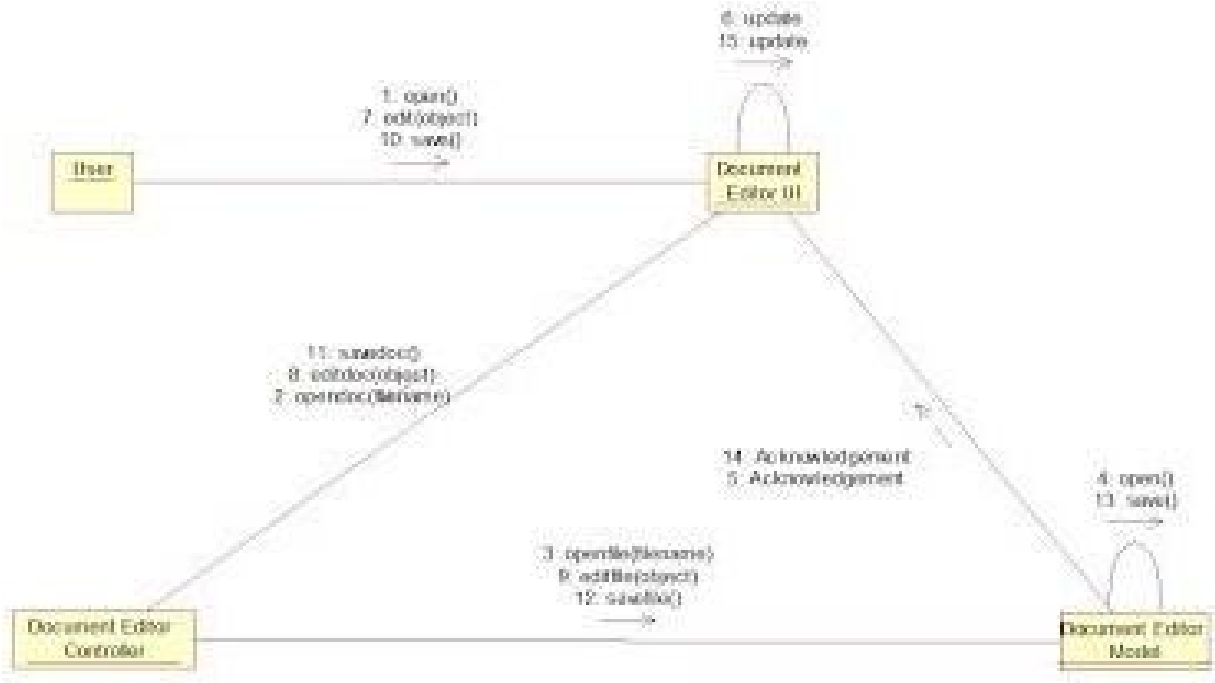
Use Case Diagram



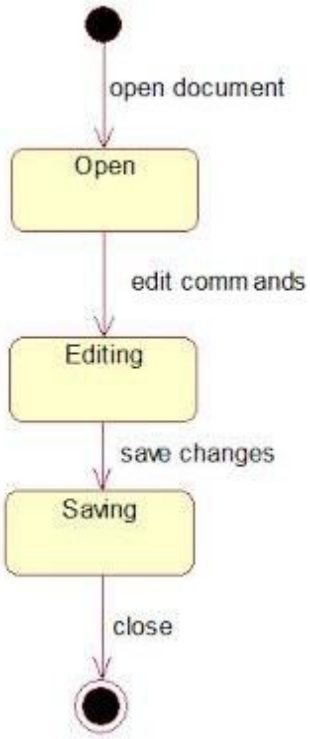
Sequence Diagram



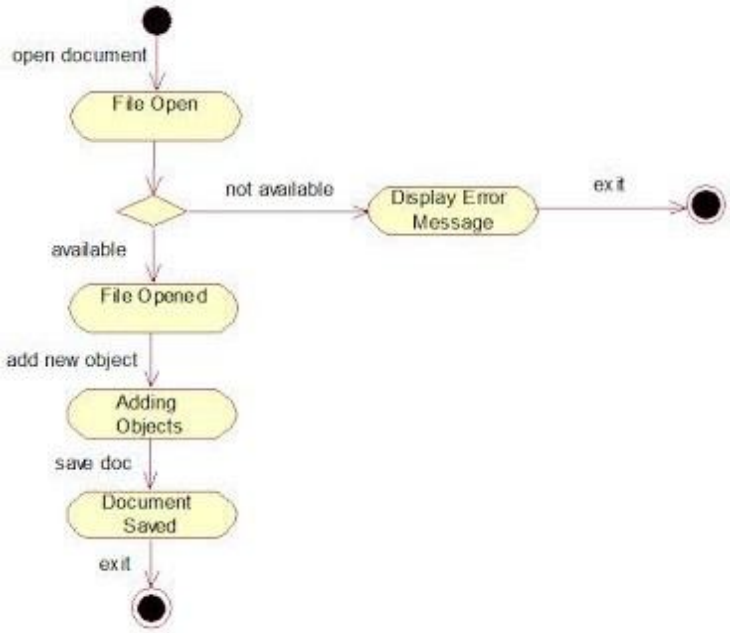
Collaboration Diagram



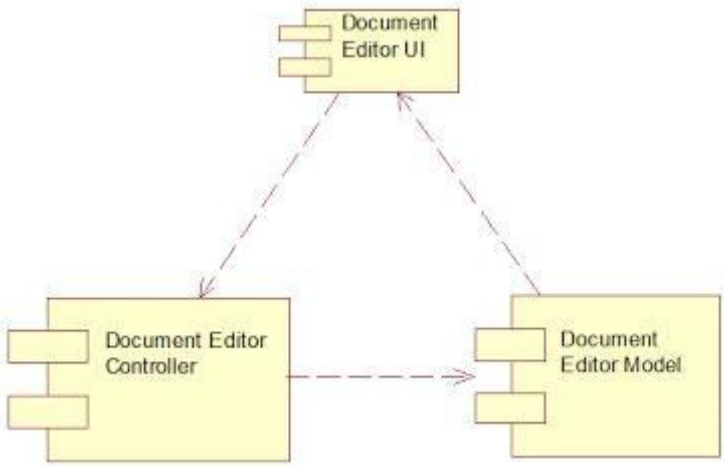
State Chart Diagram



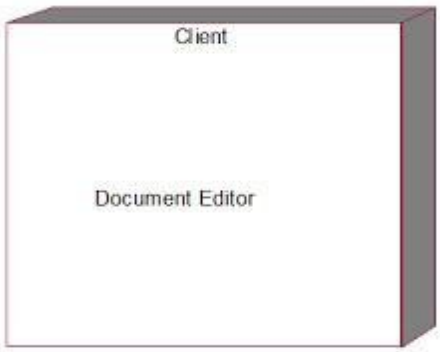
Activity Diagram



Component Diagram



Deployment Diagram



Subject In charge

Mrs. Leena R. Waghulde