Amity School of Engineering & Technology
Amity University

Object Oriented System Design

Code: CSE416

Prepared By

Hari Mohan Pandey
Assistant Professor, CSE Department
hmpandey@amity.edu

Module Notes

Department of Computer Science & Engineering
ODD Semester 2014

UNIT-1

OBJECT ORIENTED SYSTEM DESIGN

1.1 Introduction:

- The term “object oriented” means that software is a collection of discrete objects.
- Object-oriented development method supports for developing software components which are reusable and highly modular in nature.

1.2 Object Oriented Paradigm

- Object-oriented paradigm is a programming paradigm that uses "objects" and their interactions to design applications and computer programs.
- It is based on several techniques, including
  - Inheritance,
  - Modularity
  - Polymorphism, and
  - Encapsulation.
- It was not commonly used in mainstream software application development until the early 1990s.
- Many modern programming languages now support OOP.

1.3 OBJECT-ORIENTED TERMINOLOGIES

Basic concepts of object orientation

1.3.1 CLASS

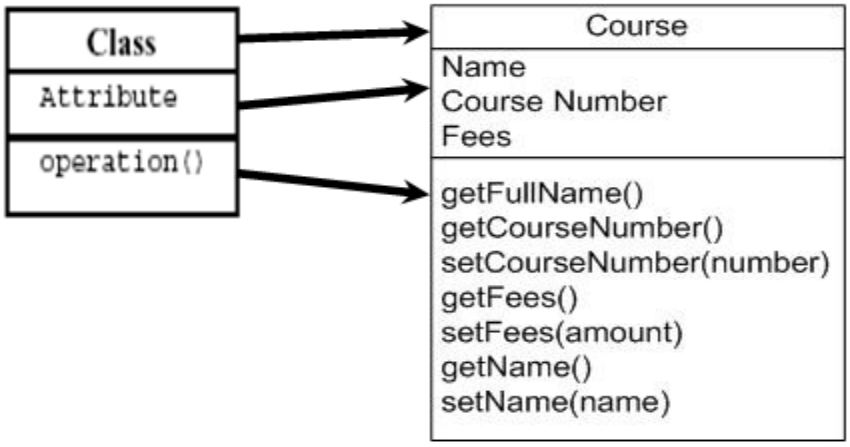
- Class is a template inclusive of data and methods (that act on the data) for creating objects.
- Each instance (object) of a class is unique.
- A class is a more general abstraction.
- Class is a concept that describes a set of objects that are specified in the same way.

- Classes are user defined data types and behave like built in types of a programming language.
- **Example:**

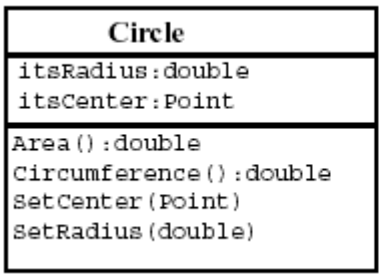
```
Class Student
{
}
main()
{
  int a;              // primitive data type

  Student ahmed;      // User defined data type
                      // (Student is the class)
}
```

Class Declaration

- All objects of a class are having same specification as shown in the figure-1.1 below:



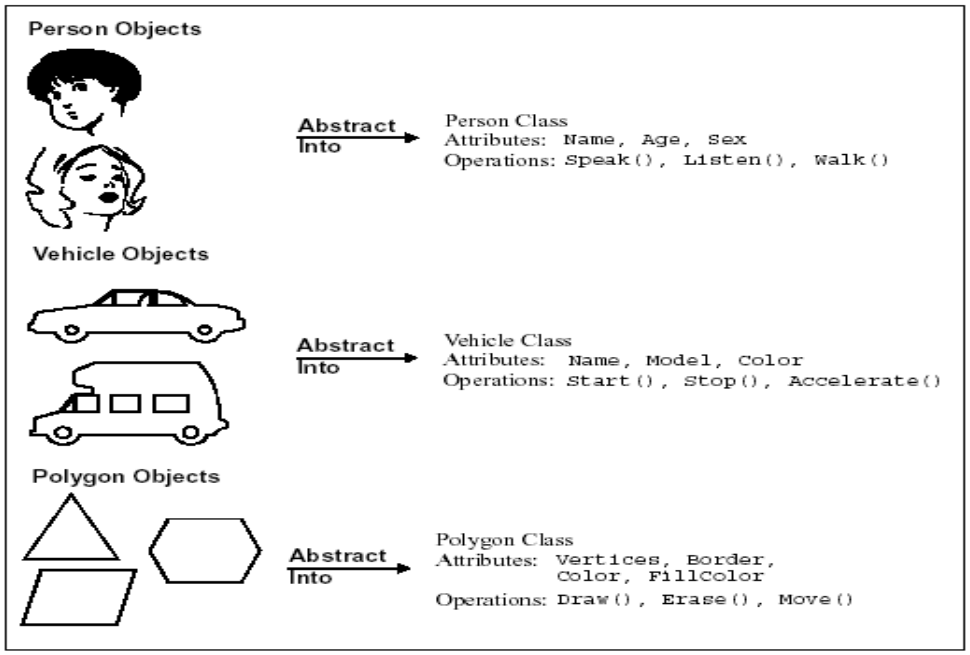
- Programming form of a class is shown below




```
class Circle
{
  public:
    void SetCenter(const Point&);
    void SetRadius(double);
    double Area() const;
    double Circumference() const;
  private:
    double itsRadius;
    Point itsCenter;
};
```

```
class Account {
  private String accountName;
  private double accountBalance;

  public withdraw();
  public deposit();
  public determineBalance();
} // Class Account
```

- More real life example are given below (figure-1.4):



1.3.2 OBJECT

- An object represents a particular instance of a class
- The object has 'state, behavior and identity'

- State represents the particular condition that an object is in at a given moment
- Behavior stands for the things that the object can do that are relevant to model.
- Identity means that every object is unique.
- **A Case:** Assume that there is a person whose name is Ahmed and he is reading and studying then for object:

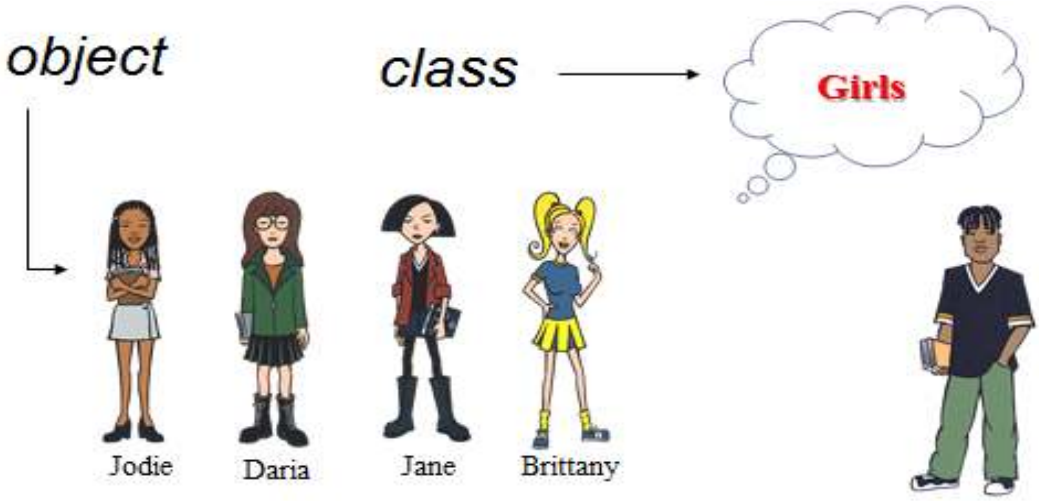
  Object : Person
  Identity : Ahmed
  Behavior: Reading
  State : Studying

1.3.2.1 OBJECT STATE

- The state of an object is the condition of the object or a set of circumstances describing the object
- The concept of object state is fundamental to an understanding of the way that the behavior of an object-oriented software system is controlled so that the system responds in an appropriate way when an external event occurs.
- Each state is represented by the current values of data within the object.
- Each state is also characterized by a difference in behavior.
- For example, the control software must be designed to take account of all possible states of the aircraft like parked, climbing, flying level on auto-pilot, and landing.
- The appropriate control behaviors for each state such as shut down engine, full throttle, climb, descend, turn.

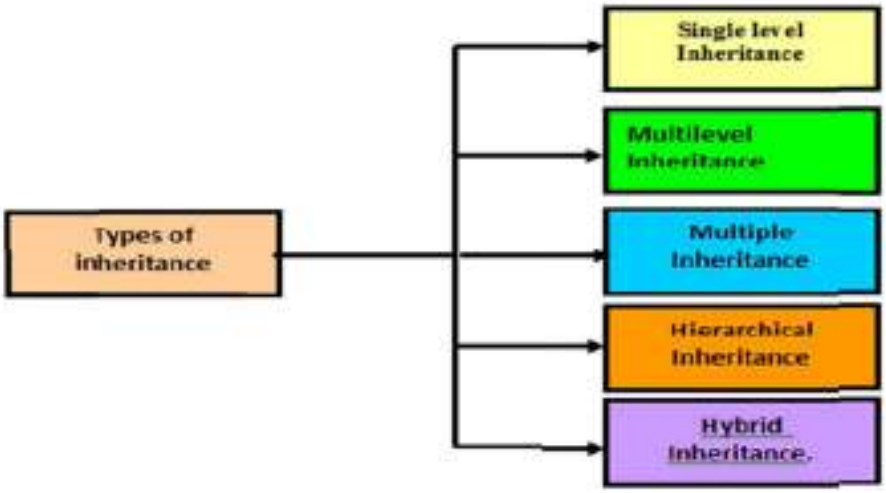
1.3.2.2 Class and Objects

- Classes reflect concepts; objects reflect instances that embody those concepts



1.3.3 INHERITANCE

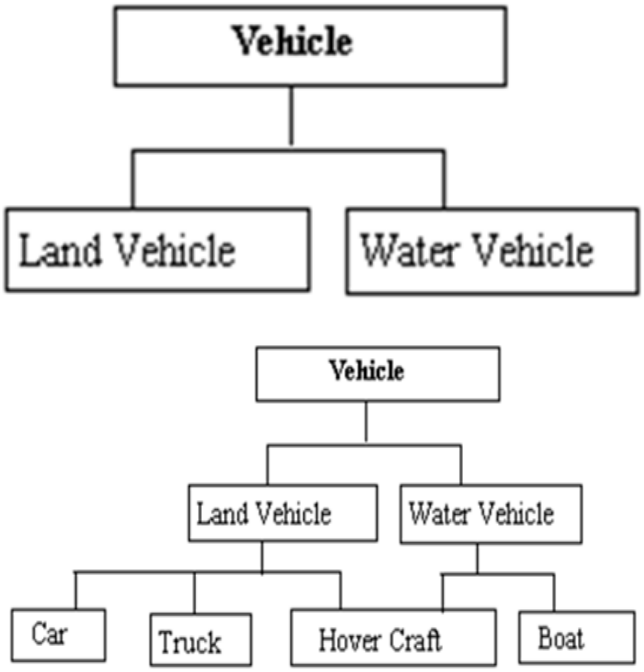
- The concept of deriving a new child (derived) class from an existing parent (super).
- Inheritance allows the objects of one class to acquire the properties of objects of another class.
- This provides code reusability
- Classes can be arranged into hierarchies.
- Generalization is the relationship between super class and sub class.
- There are different types of inheritances like :
  - **Single Level**: In this type there will be one base class and one derived class.
  - **Multilevel Inheritance:** In this case one class is derived from a second class which in turn is derived from a third class.
  - **Multiple**: In this type there will be many base classes but one derived class.
  - **Hierarchical**: In this type there will be one base class but many derived classes.
  - **Hybrid**: This is a combination (***any two or more types of inheritance***) of Multiple and Hierarchical Inheritance.



- Inheritance is a mechanism for implementing generalization in an object-oriented programming language.
- When two classes are related by the mechanism of inheritance, the general class is called super class in relation to the other and the more specialized is called its subclass.
- A subclass inherits all the characteristics of its super class.
- **Inheritance: A Scenario to Understand**

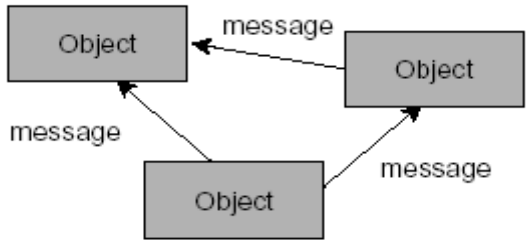
➢ **Scenario:** Let's take an example of Vehicle, used for transportation services. We can categorize vehicle into two categories namely Land Vehicle and Water Vehicle, which we can further categorize into car, truck hover craft, and boat respectively.

➢ **Question???:** Discuss and design diagrammatical representation of the scenario given above and explain the types of inheritance you are using and why? Also tell attributes and operation you want to perform?



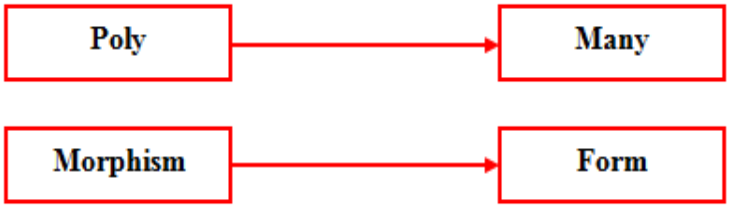
1.3.4 MESSAGE PASSING (Object Communication)

- In object-oriented system, objects communicate with each other by sending message, how people communicate with each other.
- In the object-oriented programming terms it is called as encapsulation.
- Encapsulation defined as a bundle of data together with some operations that act on the data.
- An object knows only its own data and its own operations.
- Each operation has a specific signature.
- Message passing is a way of insulating each object from needing to know any of the internal details of other objects.
- When an object receives a message it can tell instantly whether the message is relevant to it or not based on the valid signature to one of its operations.



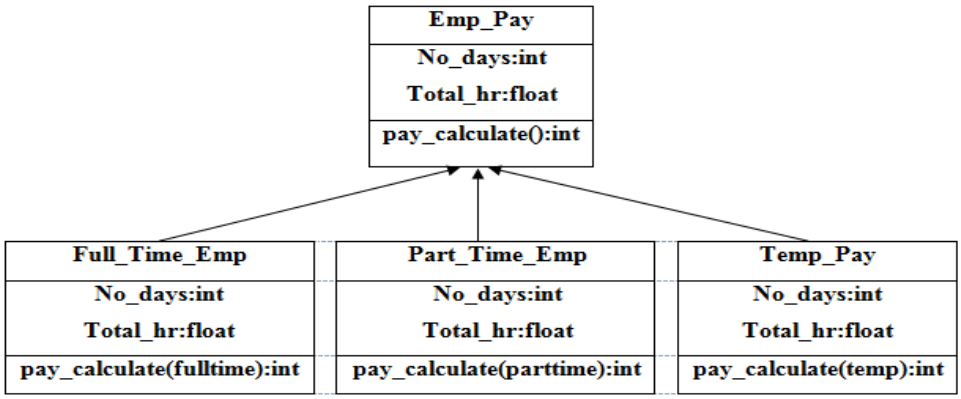
1.3.5 POLYMORPHISM: What?

- If we bi-furcated the word Polymorphism we get "Poly" means many and "Morphism" means form.



- Polymorphism literally means 'an ability to appear as many forms'.
- Polymorphism is a powerful concept for the information systems developer.
- It permits a clear separation between different sub-systems that handle superficially similar tasks in a different manner.
- **For example**:

  There are different ways of calculating an employee's pay. For full-time employees are paid a salary that depends on his / her grade; part-time employees payments are based on number of hours worked. The temporary employee payment differs in that no deductions are made for the company pension scheme.



1.3.6 ENCAPSULATION

- The wrapping up of attribute and operation into one unit is called Encapsulation.
- This allows the data is not accessible to the outside world and only those methods which are wrapped in the class.
- In Object Oriented Programming, encapsulation is an attribute of object design.
- It means that all of the object's data is contained and hidden in the object and access to it restricted to members of that class.
- Programming languages aren't quite so strict and allow differing levels of access to the object's data.
- The first three levels of access shown below are in both C++ and Java
  - **Public:** All Objects can access it.
  - **Protected:** Access is limited to members of the same class or descendants.
  - **Private:** Access is limited to members of the same class.
  - **Package / Internal:** Access is limited to the current package.

1.3.7 DYNAMIC BINDING

- In object oriented programming, dynamic binding refers to determining the exact implementation of a request based on both the request (operation) name and the receiving object at the run-time.
- It often happens when invoking a derived class's member function using a pointer to its super class.

- The implementation of the derived class will be invoked instead of that of the super class.
- It allows substituting a particular implementation using the same interface and enables polymorphism.

1.4 Introduction to UML

- Modeling is the designing of software applications before coding.
- Modeling is an Essential Part of large software projects, and helpful to medium and even small projects.
- *The Unified Modeling Language (UML) is a graphical language for* visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements

  - **Visualizing**: Graphical models with precise semantics
  - **Specifying:** Models are precise, unambiguous and complete to capture all important Analysis, Design, and Implementation decisions.
  - **Constructing:** Models can be directly connected to programming languages, allowing forward and reverse engineering
  - **Documenting:** Diagrams capture all pieces of information collected by development team, allowing sharing and communicating the embedded knowledge.
- UML defines the following nine diagrams

- **Use case diagrams**
- **Class diagrams**
- **Object diagrams**
- **Sequence diagrams**
- **Collaboration diagrams**
- **State Chart diagrams**
- **Activity diagrams**
- **Component diagrams**
- **Deployment diagrams**

- Divided into three major categories:
  - **Structure Diagrams** include the Class Diagram, Object Diagram, Component Diagram, and Deployment Diagram.

- **Behavior Diagrams** include the Use Case Diagram, Activity Diagram, and State Machine Diagram.
- **Interaction Diagrams** include the Sequence Diagram, Collaboration Diagram.

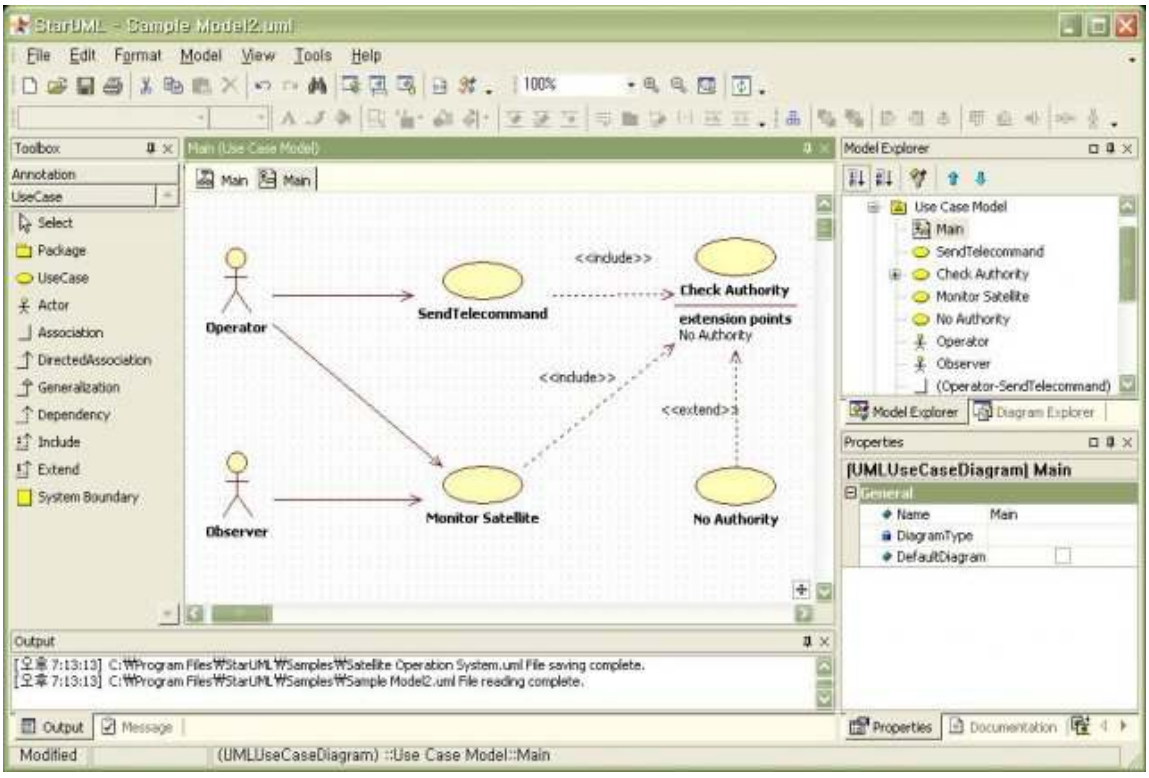
1.5 STAR UML- A modeling tool

- StarUML is an open source project to develop fast, flexible, extensible, featureful, and freely-available UML/MDA platform running on Win32 platform.
- StarUML™ is software modeling platform rather than just a UML tool.
- It is a compelling replacement of commercial UML tools such as Rational Rose, Together and so on.
- StarUML is *multi-lingual project* and not tied to specific programming language.

1.6 Features of Star UML

Some unique features of StarUML were listed below.

- End users want customizable tools. Providing a variety of customizing variables to meet the requirements of the user environment can ensure high productivity and quality.
- No modeling tool provides a complete set of all possible functionalities. A good tool must allow future addition of functions to protect the user's investment costs in purchasing the tool.
- MDA (Model Driven Architecture) technology requires not only independent platforms but multi-platform functionality. Modeling tools confined to specific development environments are not suitable for MDA. The tool itself should become a modeling platform to provide functionality for various platform technologies and tools.
- Integration with other tools is vital for maximization of the tool's efficiency. The tool must provide a high level of extensibility, and allow integration with existing tools or user's legacy tools.



1.7 More about Unified Modeling Language (UML)

UML is a language for visualizing, specifying, constructing and documenting the artifacts of a software intensive system. To understand the UML, you need to form a conceptual model of the language, and this requires learning three major elements: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML. Once you have grasped these ideas, you will be able to read UML models and create some basic ones. As you gain more experience in applying the UML, you can build on this conceptual model, using more advanced features of the language.

UML is a set of notations purposely *composed* to address the problem of specifying Object Oriented Systems managed by the OMG. OO systems pose unique challenge to all stakeholders in the development lifecycle. UML provides a common ***language*** that defines the System under Development (SUD) in a way that is meaningful to all stakeholders.

1.8 Characteristics of UML

- **Abstract:**-Good for presenting important ideas, uncluttered by detail. Like natural language, unlike programming language

- **Precise:-**Helps expose gaps and inconsistencies. Like programming language, unlike natural language
- **Visual (mainly):** Easy to see the main objects and relationships. Used to complement a good narrative in requirements and design documentation. discussing design ideas at the whiteboard
- **Non-prescriptive:** You use it how it best suits your development culture

1.9 Relationships in the UML

There are four kinds of relationships in the UML:

1. Dependency

2. Association

3. Generalization

4. Realization

These relationships are the basic relational building blocks of the UML. You use them to write well-formed models. First, a ***dependency*** is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing).

Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.

Second, an ***association*** is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names which will discuss later

Third, a ***generalization*** is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.

Fourth, a ***realization*** is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them.

1.10 When to use Packages

Grouping Things are the organizational parts of UML models. These are the boxes into which a model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.

A *package* is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time). Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents.

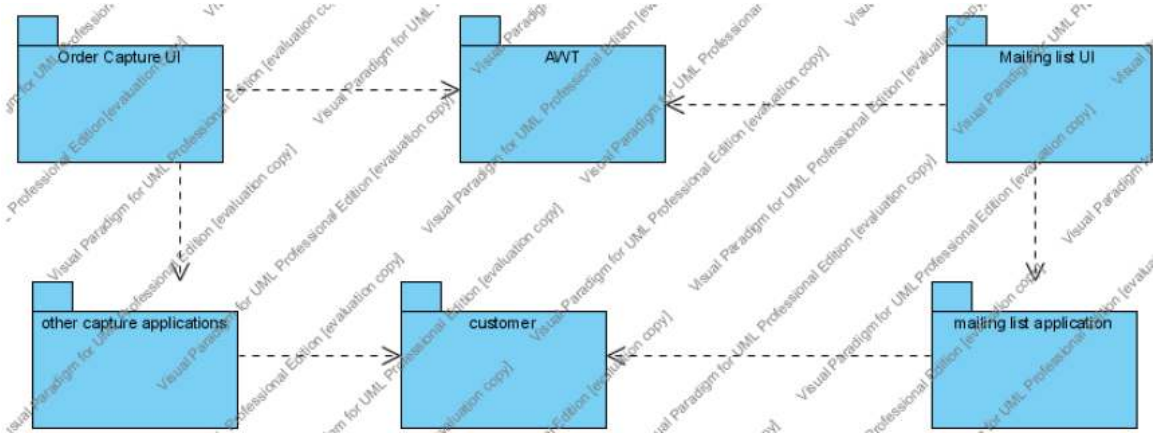
Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

The idea of package can be applied to any model element, not just classes. Without some heuristics to group classes together, the grouping becomes arbitrary.

Package diagram is a diagram that shows packages of classes and the dependencies among them. Strictly speaking, a package diagram is just a class diagram that shows only packages and dependencies. Package diagram is not an official UML diagram.

A dependency exists between two elements if changes to the definition of one element may cause changes to the other. With classes, dependencies exits for various reasons. One class sends a message to another; one class has another as part of its data; one class mentions another as a parameter to an operation. If a class changes its interface, any message it sends may no longer valid.

Example of a package diagram



1.11 UML Diagrams

A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system.
In practice, however, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software-intensive system. For this reason, the UML includes nine such diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. State chart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

A ***class diagram*** shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

An ***object diagram*** shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

A ***use case diagram*** shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Both **sequence diagrams** and **collaboration** diagrams are kinds of interaction diagrams. An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them.
Interaction diagrams address the dynamic view of a system. A *sequence diagram* is an interaction diagram that emphasizes the time-ordering of messages; a *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the

objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

A ***statechart diagram*** shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

An ***activity diagram*** is a special kind of a statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

A ***component diagram*** shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

A **deployment diagram** shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of architecture. They are related to component diagrams in that a node typically encloses one or more components.

This is not a closed list of diagrams. Tools may use the UML to provide other kinds of diagrams, although these nine are by far the most common you will encounter in practice.

1.12 Use Case Diagram

A diagram that shows a set of use cases and actors and their relationships. Use cases represent system functionality, the requirements of the system from the user‟s perspective. Use case diagrams show actor and use case together with their relationships. The use cases represent functionality of a system or a classifier, like a subsystem or a class, as manifested to external interactors with the system or the classifier.

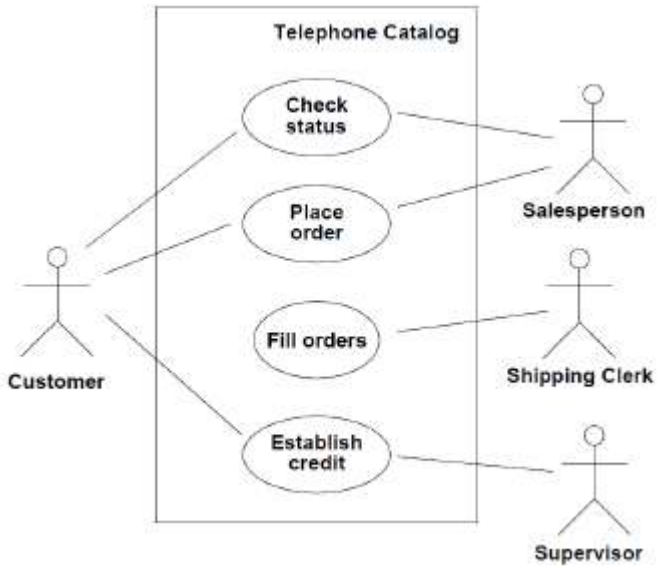
1.13 Notations in Use Case Diagram

Use case: A description of a set of sequences of actions, including variants, that system performs that yields an observable value to an actor.

Actor:-The people or systems that provide or receive information from the system; they are among the stakeholders of a system. Actors that stimulate the system and are the initiators of

events are called primary actors (active)Actors that only receive stimuli from the system are called secondary actors (passive)

A basic Use Case diagram for sales order processing



1.14 Guide lines for drawing Use Case Diagrams

1. Use cases should ideally begin with a verb – i.e. generates report. Use cases should NOT be open ended – i.e. Register (instead should be named as Register New User)

2. Avoid showing communication between actors.

3. Actors should be named as singular. i.e student and NOT students. NO names should be used – i.e. John, Sam, etc.

4. Do NOT show behaviour in a use case diagram; instead only depict only system functionality.

1.15 Use case diagram does not show sequence.

There are several standard relationships among use cases or between actors and use cases.

- Association – The participation of an actor in a use case, i.e. instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases.
- Extend – An extend relationship from use case A to use case B indicates that an instance of use case B may be extended (subject to specific conditions specified in the extension) by the behavior specified by A. The behavior is inserted at the location defined by the extension point in B which is referenced by the extend relationship.
- Generalization – A generalization from use case A to use case B indicates that A is a specialization of B.
- Include – An include relationship from use case A to use case B indicates that an instance of the use case A will also include the behavior as specified by B. The behavior is included at the location which defined in A.

1.16 Use Case Specification

Use case specification is synonymous to use case description .and use case definition and can used interchangeably. Use case specification defines information that pertains to a particular use case which is important in understanding the purpose behind the use case. Use case specification is written for every use case. A use case specification has one or more flow of events or pathways association with it.

A flow of events or pathway is a textual description embodying sequence of events with regards to the use case and is part of the use case specification. Flow of events is understood by the customer. A detailed description is necessary so that one can better understand the complexity that might be involved in realizing the use cases.

Use case specification serves as a „bridge" between stakeholders of a system and the development team.

Types of Flow of Events

Basic Flow of Events@ Happy Path – You get to the ATM and successfully withdraw money

Alternate Flow of Events @ Alternate Pathway - You get to the ATM but could not withdraw money due to insufficient funds in your account.

Exception Flow of Events @ Exception Pathways @ Unhappy Pathway – You get to the ATM machine but your valid pin number is not accepted.

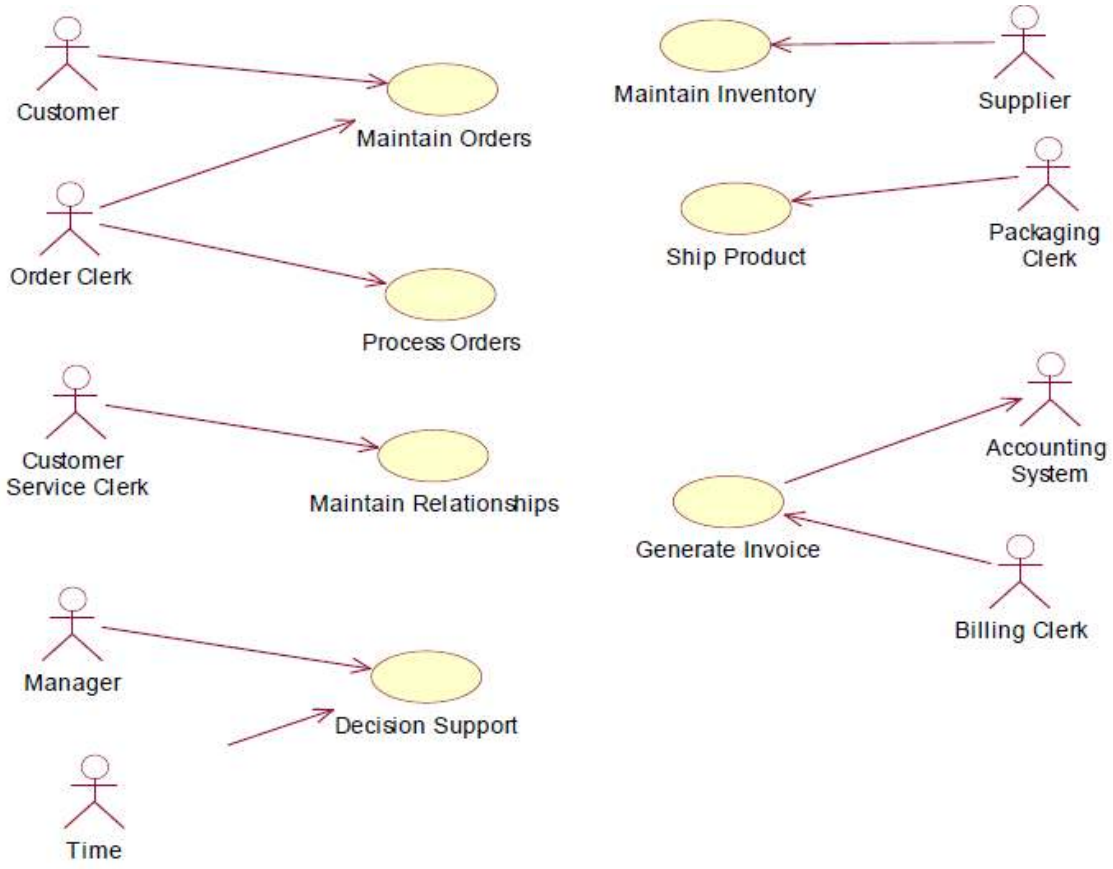
Case Study – Remulak Productions

Case Study

Remulak Productions is a very small company located in Newport Hills, Washington, a suburb of Seattle. Remulak specializes in finding hard-to-locate musical instruments –in particular guitars – ranging from traditional instruments to ancient varieties no longer produced. Remulak also sells rare and in-demand sheet music. Further, it is considering adding to its product line other products that are not as unique as its instruments; include recording and mixing technology, microphones, and recordable company disk (CD) players. Remulak is a very ambitious company; it hopes to open a second order-processing location on the East Coast within a year.

Most of Remulak‟s orders are taken in-house by order entry clerks. However, some of its sales come from third-party commercial organizations. Remulak‟s founder, owner, and president, Jeffery Homes, realizes that he cannot effectively run his company with its current antiquated order entry and billing system. Your challenge is to design and implement a system that not only meets the company‟s immediate needs but also is flexible enough to support other types of products in the future.

Suggested Use Case Diagram



Use Case Specification

Use Case: Process Orders
Name: Process Orders.
Description:
This use-case starts when an order is either initiated or inquired about. It handles all aspects of the initial definition and authorization of an order, and it ends when the order clerk completes a session with a customer.
Author(s): Rene Becnel.
Actor(s): Order clerk.
Location(s): Newport Hills, Washington.
Status: Pathways defined.
Priority: 1.
Assumption(s):
Orders will be taken by the order clerk until the customer is comfortable with the specialized services being provided.
Precondition(s):
Order clerk has logged into the system.
Post condition(s):
• Order is placed.
• Inventory is reduced.
Primary (Happy) Path:
• Customer calls and orders a guitar and supplies, and pays with a credit card.
Alternate Pathway(s):
• Customer calls and orders a guitar and supplies, and uses a purchase order.
• Customer calls and orders a guitar and supplies, and uses the Remulak easy finance plan to pay.
• Customer calls and orders an organ/ and pays with a credit card.
• Customer calls and orders an organ, and uses a purchase order.
Exception Pathway(s):
• Customer calls to place an order using a credit card, and the card is invalid.
• Customer calls with a purchase order but has not been approved to use the purchase order method.
• Customer calls to place an order, and the desired items are not in stock.

1.17 Subsystems

A subsystem is a coherent and independent component of a system. Each subsystem can then be designed independently without affecting the others

Typically, have a .client-server relationship: client calls on the supplier (sends a .message.) supplier performs some service and replies with result.

Client must know interface of supplier, but supplier does not have to know interfaces of clients.

A subsystem is not an object nor a function, but a package of classes

Sub systems are at once a part of a large system. Communication between sub-systems is, by definition, through interfaces

Subsystems are provided primarily to help in the organization aspects a block diagram. Subsystems do not define a separate block diagram.

1.18 When to use a subsystem?

An object oriented subsystem encapsulates a coherent set of responsibilities in order to ensure that it has integrity and can be maintained. A subsystem has a specification of the behavior collectively offered by the model elements contained in the subsystem. This specification of the subsystem consists of operations on the subsystem, together with specification elements such as use cases, state machines, etc.

This is generally a good basis for subsystem structuring because objects that communicate frequently with each other are good candidates for being in the same subsystem. The object depicted on several collaboration diagrams can only be part of one subsystem.

If one object is located at a geographically remote location from another object, the two objects should be in different subsystems (for an example, clients and servers)

Objects that are part of the same composite object should be in the same subsystem.

On the other hand, objects in a aggregate subsystem grouped by functional similarity might span geographical boundaries.

The subdivision of an information system into subsystems has the following advantages.

- It produces smaller units of development
- It helps to maximize reuse at the component level
- It helps the developers to cope with complexity
- It improves maintainability
- It aids portability

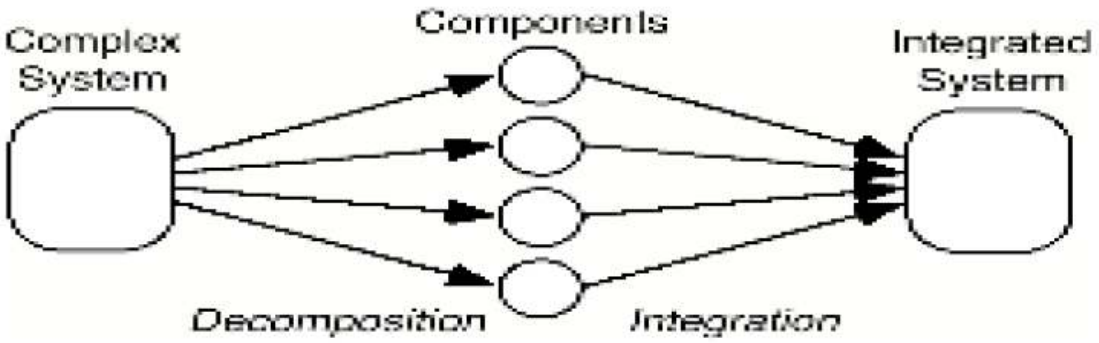
Dividing a system into subsystems is an effective strategy for handling the complexity. Sometimes it is only feasible to model a large complex system piece by piece, with the subdivision forced on the developers by the nature of the application. Splitting a system into subsystem can also aid reuse, as each subsystem may correspond to a component that is suitable for reuse in other applications.

There are two general approaches to the division of a software system into subsystems. These are known as **layering** – so called because the different subsystems usually represent different levels of abstraction- and **partitioning**, which usually means that each subsystem focuses on a different aspect of the functionality of the system as a whole. In practice both approaches are often used together on one system, so that some of its subsystems are divided by layering, while others are divided by partitioning.

System decomposition

In order to develop a complex product or large engineering system, it is common practice to decompose the design problem into smaller sub-problems which can be handled more easily. If any of the sub-systems are still too complex, they may in turn be further decomposed. Development teams are assigned to each design problem which may represent a component or sub-system of the larger system. One important level of integration takes place within each development team. This is the now common practice of concurrent engineering, in which a cross-functional team addresses the many design and production concerns simultaneously. However, to assure that the entire system works together, the many sub-system development teams must work together. This latter form of integration is often called system engineering.

Figure graphically depicts the relationship of problem decomposition and system integration.



Distributed Development

Distributed information systems have become more common as communications technology has improved and have also become more reliable. An information system may be distributed over computers at the same location or at different locations. A distributed information system may be supported by software products such as distributed database management systems or object request brokers or may adopt service oriented architecture.

This software architecture is based on Object Request Broker (ORB) technology, but goes further than the Common Object Request Broker Architecture (CORBA) by using shared, reusable business models (not just objects) on an enterprise-wide scale. The benefit of this architectural approach is that standardized business object models and distributed object computing are combined to give an organization flexibility to improve effectiveness organizationally, operationally, and technologically

The benefit of this architectural approach is that standardized business object models and distributed object computing are combined to give an organization flexibility, scalability, and reliability and improve organizational, operational, and technological effectiveness for the entire enterprise. This approach has proven more cost effective than treating the individual parts of the enterprise

Distributed/collaborative enterprise builds its new business applications on top of distributed business models and distributed computing technology. Applications are built from standard interfaces with "plug and play" components. At the core of this infrastructure is an off-the-shelf, standards-based, distributed object computing, messaging communication component such as an Object Request Broker (ORB) that meets Common Object Request Broker Architecture (CORBA) standards.

This messaging communication hides the following from business applications:

- the implementation details of networking and protocols
- the location and distribution of data, process, and hosts
- production environment services such as transaction management, security, messaging reliability, and persistent storage

The message communication component links the organization and connects it to computing and information resources via the organization's local or wide area network (LAN or WAN). The message communication component forms an enterprise-wide standard mechanism for accessing computing and information resources. This becomes a standard interface to heterogeneous system components.

Specification and realization elements
Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things. Things in the UML There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models. ***Structural things*** are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things. First, a logical *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

A **class** implements one or more interfaces. Graphically, a class is rendered as a rectangle; usually including its name, attributes, and operations, as in
Second, an ***interface*** is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior.

An interface defines a set of **operation** specifications (that is, their signatures) but never a set of operation implementations. Graphically, an interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface.

Third, collaboration defines an **interaction** and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations. Graphically, collaboration is rendered as an ellipse with dashed lines, usually including only its name.

Fourth, a ***use case*** is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name.

Behavioral Things *are* the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things.

First, an interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. The behavior of a society of objects or of an individual operation may be specified with an interaction. An interaction involves a number of other elements, including messages, action sequences (the behavior invoked by a message), and links (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation.

Second, a **state machine** is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its

responses to those events. The behavior of an individual class or a collaboration of classes may be specified with a state machine.

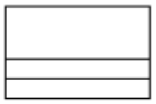
A **state machine** involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its sub states, if any.

Annotational Things *Annotational things* are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note.

Structural Modeling

Structural model: a view of an system that emphasizes the structure of the objects, including their classifiers, relationships, attributes and operations.

Structural modeling –Core Elements

Construct	Description	Syntax
class	a description of a set of objects that share the same attributes, operations, methods, relationships and semantics.	
interface	a named set of operations that characterize the behavior of an element.	
component	a modular, replaceable and significant part of a system that packages implementation and exposes a set of interfaces.	
node	a run-time physical object that represents a computational resource.	

Structural modeling –Core Relationships

Construct	Description	Syntax
association	a relationship between two or more classifiers that involves connections among their instances.	
aggregation	A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.	
generalization	a taxonomic relationship between a more general and a more specific element.	
dependency	a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).	

Construct	Description	Syntax
realization	a relationship between a specification and its implementation.	

Static Structural Diagrams: Shows a graph of classifier elements connected by static relationships. They are of two types

1. Class diagram: classifier view
2. Object diagram: instance view

What are the United process (UP) phases - Case study – the NextGen POS system, Inception

UNIT-2

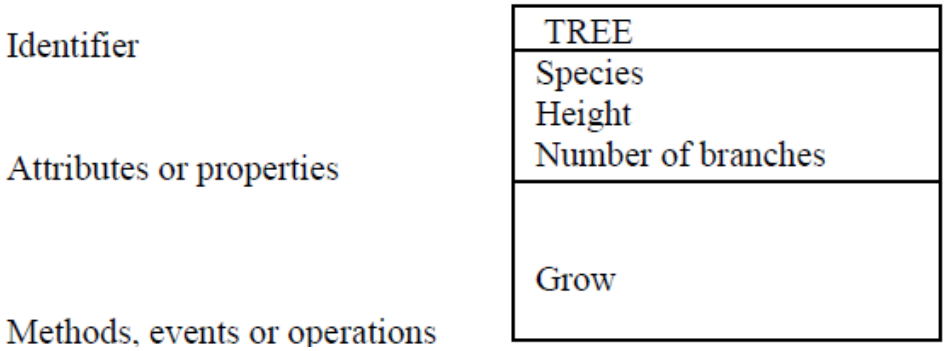
MODELS AND CLASS

2.1 Class Diagram

A class diagram is a graph of Classifier elements connected by their various static relationships. Note that a "class" diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links. Perhaps a better name would be "static structural diagram" but "class diagram" is shorter and well established.

We can start to show the details of a class by using a simple box or rectangle. The identifier or name of the class goes at the top, below this are shown the properties or attributes, and below this we show the methods or events.

Let us look at an example. We will describe a tree, the sort with branches and leaves



We know that a tree is of a particular species, it has height which will change gradually throughout its life, as will the number of branches. These are all properties or attributes of any tree. Any tree will also be able to do various things. These are its methods or events, and in the example above we have shown two possibilities; grow, and die. Clearly we could add many more methods and attributes.

2.2 Attributes

- The attributes has a different meaning and would be manipulated differently for each of these data types and it is important to determine during analysis which meaning is appropriate.
- An attribute data type is declared in UML using the following syntax:

Name ':' type-expression '=' initial-value '{property-string'}'

- The name is the attribute name,

- The type-expression is its data type
- The initial-value is the value the attribute is set to when the object is first created
- The property-string describes a property of the attribute such as constant or fixed

2.3 Operations

- Each operation also has to be specified in terms of the parameters that it passes and returns.
- The syntax used for an operation is:

Operation name '('parameter-list ')' ':' return-type-expression

- An operation's signature is determined by:
  - Operation's name,
  - The number and type of its parameters and
  - The type of the return value if any.

2.4 IDENTIFICATION OF CLASS

- For identifying class, there is technique called noun identification technique.
- It is a technique which can be used to identify classes.
- It is done in two steps:
  - Identify candidate classes by picking up all the nouns and noun phrases out of a requirements specification of the system
  - Discard candidates which are inappropriate for any reason, renaming the remaining classes if necessary
- Inappropriate classes:
  - **Redundant (the same class given more than one name)**
  - **Vague**
  - **An event or an operation (noun refers to something which is done to, by or in the system)**
  - **An attribute**
- **Example**

Books and journals: The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

Borrowing: The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.

Results of noun identification

• Library	Outside the scope of the system
• Books	Good
• Journals	Good
• Copy of a given book	Good
• Short term Loan	Is an event of lending book
• Library member	Good
• Week	A measure of time not a thing
• Members of library	The same as library member

Candidate classes

- Book
- Journal
- Copy (of a book)
- Library member
- Member of staff

2.5 Design Guidelines

The design guidelines are:

- **Design clarity**: A design should be made as easy to understand as possible.
- **Don't over design**: Design flexibility to produce designs that may not only satisfy current requirement but may also be capable of supporting a wide range of future requirements.
- **Control inheritance hierarchies:** inheritance hierarchies should be neither too deep nor too shallow. If a hierarchy is too deep it is difficult for the developer to understand easily what features are inherited.

- **Keep message and operations simple:** it is better to limit the number of parameters passed in a message to no more than three.
- **Design Volatility:** A good design will be stable in response to changes in requirements
- **Evaluate by Scenario:** An effective way of testing the suitability of a design
- **Design by Delegation:** A complex object should be decomposed into component objects forming a composition or aggregation.
- **Keep classes separate**: it is better not to place one class inside another. The internal class is encapsulated by the other class and cannot be accessed independently.

2.6 DESIGNING ASSOCIATIONS

- **Association** is the **relationship between two classes.**
- When the association is between two classes, it's called **Binary Association**.
- When the association is between two instances of the same class, then its called **reflexive or unary association**.
- **Example:**
  - Binary Association: **Employee *works for* Employee**
  - Unary Association**: A Employee supervises Employees**
- An association between two classes indicates the possibility that links will exist between instances of the classes.
- How attributes is a property of the object in a class, similarly, a link attribute is a property of the links in an association.
- The links provide the connections necessary for message passing to occur.
- Each link becomes one instance of the class.

For a given association between object A and object B, there can be three possible categories (the following are called **multiplicity**).

1. **One to One**. Exactly one instance of Class A is associated with exactly one instance of Class B and vice versa. Example: A department has exactly one Head and One Head can lead only one department

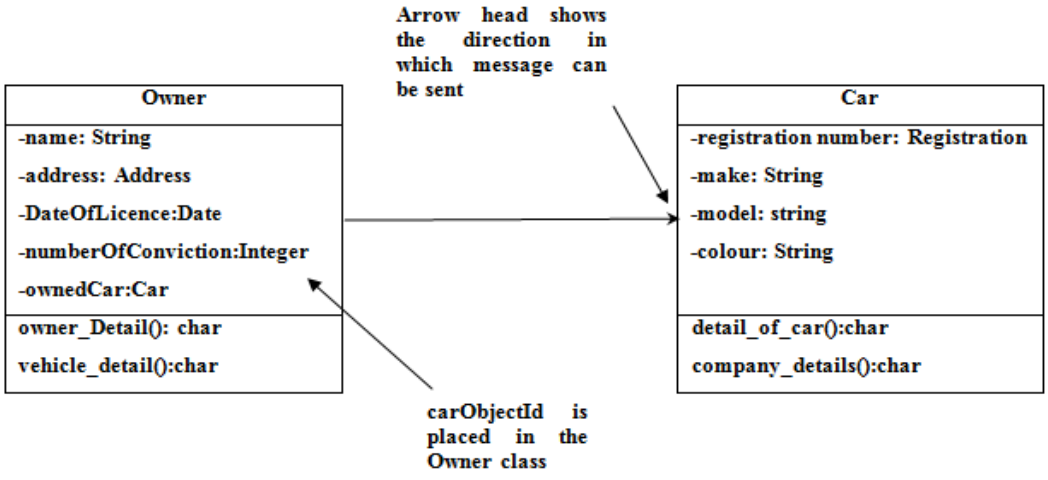
If an association is only traversed in one direction, it may be implemented as a pointer-an attribute which contains an object reference.

A one-way association: the arrow head on the association line show the direction along which it may be navigated.

Before an association can be designed it is important to decide in which direction or directions messages may be sent.

Essentially, if an object needs to send a message to a destination object it must have the destination object's identifier either passed as a parameter in an incoming message just when it is required, or the destination object's identifier must be stored in the sending object.

An association that has to support message passing in both directions is 2-2 association.
A 2-way association is indicated with arrowheads at both ends.



2. **One to Many**: One instance of Class A can have many instance of Class B. From perspective of Class B, there can be only one Class A

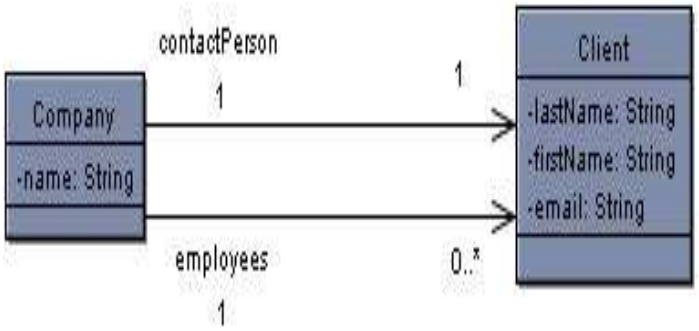
Example: In a department employees many Professors, but a professor works only for one department

3. **Many to Many**: For a given instance of Class A there can be many instance of Class B and From Class B perspective there can be many instances of Class A.
   Example: A student enrolls in many courses and a course has many students.

2.6.1 Associations – some examples

Classes can also contain references to each other. The Company class has two attributes that reference the Client class.

Although this is perfectly correct, it is sometimes more expressive to show the attributes as associations.



The above two associations have the same meaning as the attributes in the old version of the Contact class.

The first association (the top one) represents the old **contactPerson** attribute. There is one contact person in a single Company. The multiplicity of the association is one to one meaning that for every **Companythere** is one and only one **contactPerson** and for each **contactPerson** there is one **Company**.

In the bottom association there are zero or many employees for each company. Multiplicities can be anything you specify.

Some examples are shown:

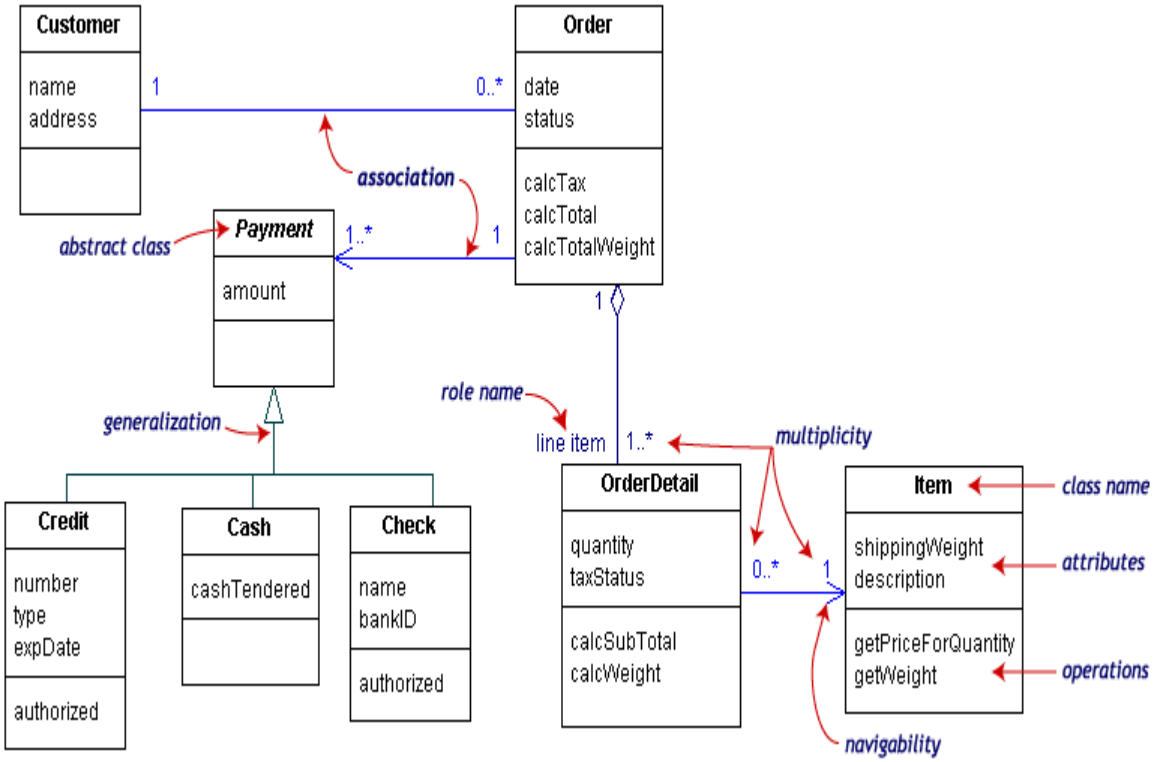
0	Zero
1	One
1..*	one or many
1..2, 10..*	one, two or ten and above but not three through nine

2.7 Class diagrams in Detail

- A Class diagram gives an overview of a system by showing its classes and the relationships among them.
- Class diagrams are static -- they display what interacts but not what happens when they do interact.

Example

The class diagrams below models a customer order from a retail catalog. The central class is the Order. Associated with it is the Customer making the purchase and the Payment. A Payment is one of three kinds: Cash, Check, or Credit. The order contains **OrderDetails** (line items), each with its associated Item.



UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes, such as Payment, are in italics. Relationships between classes are the connecting links.

2.8 Our class diagram has three kinds of relationships.

- **Association** -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.
- **Aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, Order has a collection of **OrderDetails**.
- **Generalization** -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. Payment is a superclass of Cash, Check, and Credit.

An association has two ends. An end may have a role name to clarify the nature of the association. For example, an **OrderDetail** is a line item of each Order.

A navigability arrow on an association shows which direction the association can be traversed or queried. An **OrderDetail** can be queried about its Item, but not the other way around. The arrow also lets you know who "**owns**" the association's implementation; in this case, **OrderDetail** has an Item. Associations with no navigability arrows are bi-directional.

The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers. In our example, there can be only one Customer for each Order, but a Customer can have any number of Orders.

This table gives the most common multiplicities.

Multiplicities	Meaning
0..1	Zero or one instance. The notation n . . m indicates n to m instances.
0..* or *	No limit on the number of instances (including none).
1	exactly one instance
1..2	above but not three
1..*	at least one instance

Every class diagram has classes, associations, and multiplicities. Navigability and roles are optional items placed in a diagram to provide clarity.

2.9 Activity diagrams

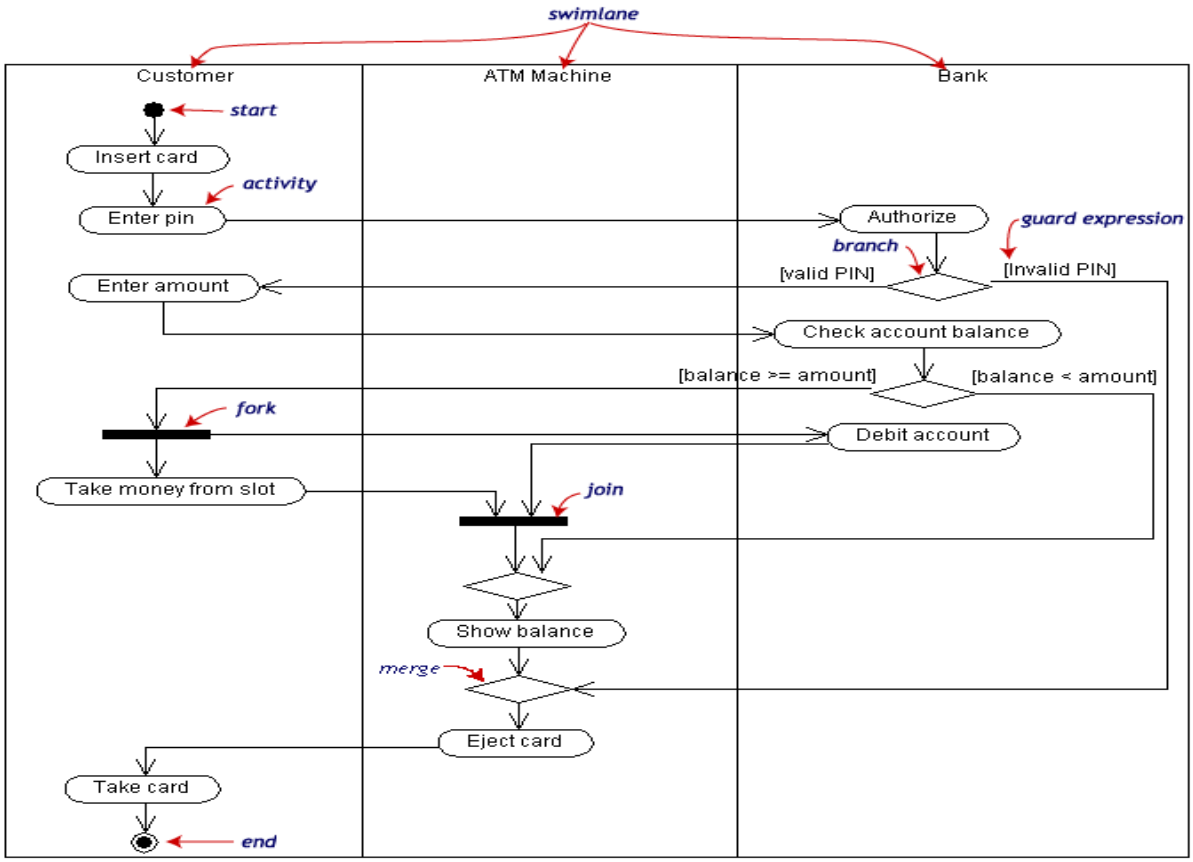
An activity diagram is essentially a fancy flowchart. Activity diagrams and statechart diagrams are related. While a statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an activity diagram focuses on the

flow of activities involved in a single process. The activity diagram shows the how those activities depend on one another.

For our example, we used the following process.

"Withdraw money from a bank account through an ATM."

The three involved classes (people, etc.) of the activity are Customer, ATM, and Bank. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are rounded rectangles.



Activity diagrams can be divided into object swimlanes that determine which object is responsible for which activity. A single transition comes out of each activity, connecting it to the next activity.

A transition may branch into two or more mutually exclusive transitions. Guard expressions (inside [ ]) label the transitions coming out of a branch. A branch and its

subsequent merge marking the end of the branch appear in the diagram as hollow diamonds.

A transition may fork into two or more parallel activities. The fork and the subsequent join of the threads coming out of the fork appear in the diagram as solid bars.

2.10 DESIGN OF OBJECT – ORIENTED METHOD

- The approach for object oriented methodology is called object Modeling Technique (OMT).
- The object oriented methodology consists of building a model of an application domain and then adding implementation details to it during the design of the system.
- There are four different stages for applying OMT model:
  - Analysis,
  - System design,
  - Object Design,
  - Implementation

Analysis

- The analyst first reads the statement of the problem and then builds a model of the real world situation showing its important properties.
- Problem statements are generally not complete or correct.
- Hence the analyst must sit with the user and try to understand the problem.
- The analysis model describes
  - What the desired system must do
  - How it will be done
- It is important to note the following points about the analysis model
  - It should not contain any implementation decisions such as data structures.

- A good model should be simple and be understood by everybody including non-programmers.

System design

- The system designer makes high-level decisions about the overall architecture.
- During the system design, the target system is divided into a number of subsystems.
- The following are the decisions that have to be made by the system designer
  - Organize the system into sub systems
  - Allocate subsystems to processes and tasks
  - Choose an approach for the management of data stores
  - Handle access to global resources
  - Choose the implementation of control in software
  - Handle boundary conditions.

Object Design

- The object designer builds a design model based on the analysis model but containing the implementation details.
- The designer adds details to the design model in accordance with the analysis model.
  - The focus in the design model is the data structures and algorithms designed to implement each class.

Implementation

- During the implementation stage, the object classes and relationships are finally translated into a particular programming language, database or hardware implementation.
- The target language to be used determines the design decisions to some extent, but the design should not depend on the fine details of the programming language.

- During implementation, it is necessary to follow good software engineering procedures so that the system remains flexible and extensible.

2.11 MODELS USED IN OBJECT MODELING TECHNIQUE

The OMT methodology uses three types of models to describe a system:

The object Model:

- It describes the static structure of the objects in a system that change over time.
- This model contains object diagrams.
- An object diagram is a graph whose nodes are object classes and whose arcs are relationships among classes.

The dynamic model

- It describes all aspects of a system that change over time.
- The dynamic model is used to specify and implement the control aspects of a system.
- The dynamic model contains state diagrams.
- The state diagram is a graph whose nodes are states and whose arcs are transitions between states caused by events.

The functional model

- It describes the data value transformations within the system.
- The functional model contains data flow diagrams.
- A data flow diagram represents a computation.
- A data flow diagram is a graph whose nodes are processes and whose arcs are data flows.

2.12 Introduction to Modeling

Systems analysts and designers produce models of systems. A business analyst will start by producing a model of how an organization works. A system analyst will produce a more abstract model of the objects in that business and how they interact with one another; a designer will produce a model of how a new computerized system will work

within the organization. In any development project that aims at producing useful artifacts, the main focus of both analysis and design activities is on models.

2.12.1 Modeling

Modeling is a proven and well accepted engineering approach. Engineers build architectural models of houses and high rises to help their users, visualize the final product. In software engineering, through modeling following aims were achieved.

1. Model helps the users to visualize a system as it is or as it want to be
2. Models permits to specify the structure or behavior of a system
3. Models gives a template that guides in the construction of the system
4. Models helps to document the decisions made

In software, there are several ways to approach a model. Two most common ways are from an algorithmic perspective and from an object oriented perspective. In the second approach, the main building of the software system is object or class. Simply put an object is a thing, generally drawn from the vocabulary of the problem domain or solution domain. A class is a description of a set of common objects. Every object has identity, state and behavior of its own.

The heart of object-oriented problem solving is the construction of a model. The model abstracts the essential details of the underlying problem from its usually complicated real world. Several modeling tools are wrapped under the heading of the UML and one example was furnished below.

2.12.2 Usefulness of model

The models are useful in several different ways as follows:

- A model is quicker and easier to build
- A model can be used in simulations, to learn more about the thing it represents.
- A model can evolve as we learn more about a task or problem.
- A model can represent real or imaginary things form any domain.

2.12.3 Diagram

Analysts and designer use diagrams to build models of system in the same way as architects use drawings and diagrams to model buildings.

- Communicate ideas
- Generate new ideas and possibilities
- Test ideas and make predications
- Understand structures and relationships

In the unified Modeling Language specification it is important to follow the rules about diagrams.

Standards for diagrams are important as they promote communication in the same way as a common language.

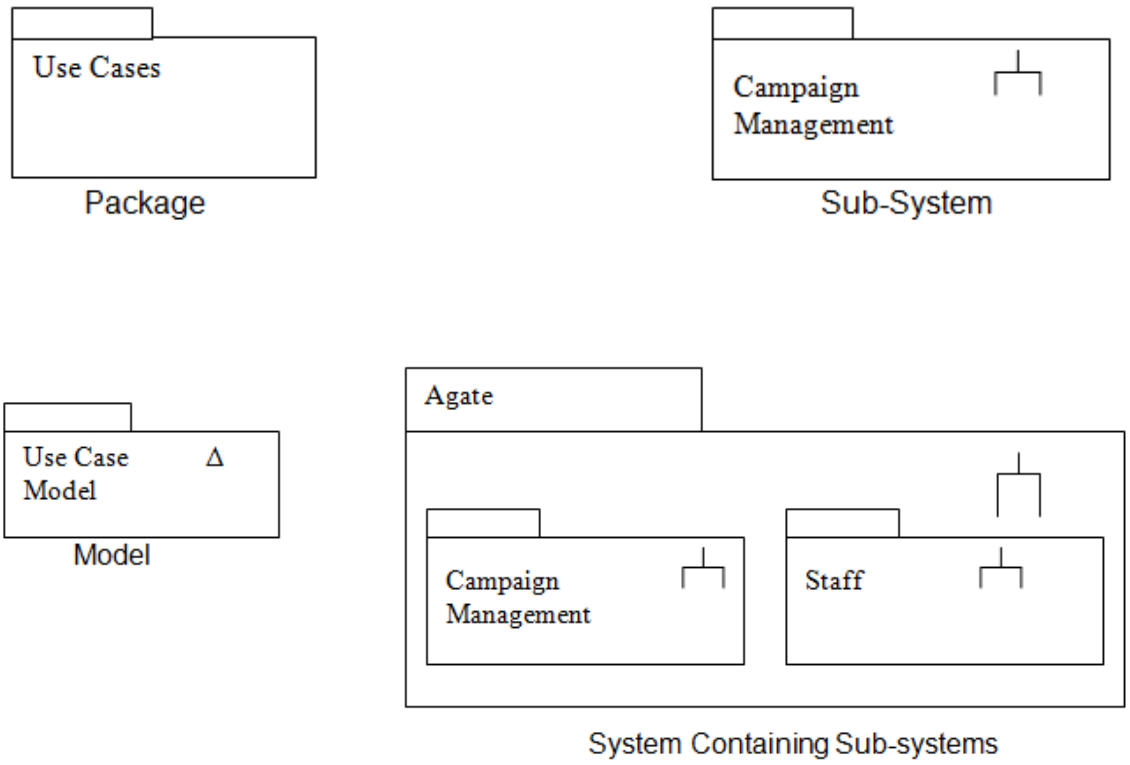
UML diagrams are made up of four elements

- Icons
- Two-dimensional symbols
- Paths
- Strings

- UML diagrams are graphs – composed of various kinds of shapes known as nodes, joined together by lines, known as paths
- Two dimensional symbols that represent activities, linked by arrows that represent the transition from one activities to another and the flow of control through the process that is being modeled.
- The start and finish of each activity graph is marked by special symbols – icons – the dot for the initial state and the dot in a circle for the final state. The activities are labeled with strings, and strings are also used at the decision points (i.e. diamond shapes) to show the conditions that are being tested.

2.12.4 Models in UML

- In UML there are a number of concepts that are used to describe systems and the ways in which they can be broken down and modeled.
- A system is the overall thing that is being modeled.
- A sub-system is a part of a system, consisting of related elements of the system.
- A model is an abstraction of a system or sub-system form particular perspective or view.
- A diagram is a graphical representation of a set of elements in the model of the system.
- UML provides a notation for modeling sub-systems and models that uses an extension of the notation for packages in UML.
- Packages are a way of organizing model elements and grouping them together.



References

1. Bennett, S., S. McRobb and R. Farmer. *Object oriented systems analysis and design using UML*. 3rd ed. McGraw-Hill, 2006.

2. Fowler, M. *UML distilled: a brief guide to the standard object modeling language*. 3rd ed. Pearson Education, 2003

3. http://portal.etsi.org/mbs/Languages/UML/uml_example.asp#Sd

4. http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/state.htm

5. http://www.geocities.com/SiliconValley/Network/1582/uml-example.htm

6. http://www.agilemodeling.com/artifacts/classDiagram.htm -

7. http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm

8. http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf - 4/10/06

9. http://www.smartdraw.com/tutorials/software-uml/uml4.htm

10. http://www.visualcase.com/screenshots.htm

11. http://www.aonix.com/ameos.html

12. http://gridbus.cs.mu.oz.au/~raj/254/Lectures/Lecture3.pdf

13. http://staruml.sourceforge.net/en/about.php

Compiled By: Hari Mohan Pandey, Assistant Professor, CSE Department