D.N.R.COLLEGE (AUTONOMOUS): BHIMAVARAM

M.Sc COMPUTER SCIENCE DEPARTMENT

Object Oriented Software Engineering

E-CONTENT

II M.Sc(Computer Science)&MCA

Presented by

K.SUPARNA

UNIT- I

Introduction to Object Oriented Software Engineering: Nature of the Software, Types of Software, Software Engineering Activities, Software Quality

Introduction to Object Orientation: Data Abstraction, Inheritance & Polymorphism, Reusability in Software Engineering, Examples: Postal Codes, Geometric Points.

Requirements Engineering: Domain Analysis, Problem Definition and Scope, Types of Requirements, Techniques for Gathering and Analyzing Requirements, Requirement Documents, Reviewing Requirements,

Case Studies: GPS based Automobile Navigation System, Simple Chat Instant Messaging System.

UNIT- II

Unified Modeling Language & Use Case Modeling: Introduction to UML, Modeling Concepts, Types of UML Diagrams with Examples; User-Centred Design, Characteristics of Users, Developing Use Case Models Of Systems, Use Case Diagram, Use Case Descriptions, The Basics of User Interface Design, Usability Principles.

Class Design and Class Diagrams: Essentials of UML Class Diagrams, Associations And Multiplicity, Generalization, Instance Diagrams, Advanced Features of Class Diagrams,

Process of Developing Class Diagrams, Interaction and Behavioral Diagrams: Interaction Diagrams, State Diagrams, Activity Diagrams, Component and Deployment Diagrams.

UNIT- III

Software Design and Architecture: Design Process, Principles Leading to Good Design, Techniques for Making Good Design Decisions, Good Design Document, Software Architecture, Architectural Patterns: The Multilayer, Client-Server, Broker, Transaction Processing, Pipe & Filter And MVC Architectural Patterns.

Design Patterns: Abstraction-Occurrence, General Hierarchical, Play-Role, Singleton, Observer, Delegation, Adaptor, Façade, Immutable, Read-Only Interface and Proxy Patterns.

UNIT- IV

Software Testing: Effective and Efficient Testing, Defects in Ordinary Algorithms, Numerical Algorithms, Timing and Co-ordination, Stress and Unusual Situations, Testing Strategies for Large Systems.

Software Project Management: Introduction to Software Project Management, Activities of Software Project Management, Software Engineering Teams, Software Cost Estimation,

Project Scheduling, Tracking And Monitoring.

Software Process Models: Waterfall Model, The Phased Released Model, The Spiral Model, Evolutionary Model, The Concurrent Engineering Model, Rational Unified Process.

UNIT- I

Introduction to Object Oriented Software Engineering:

Object-oriented software engineering (commonly known by acronym **OOSE**) is an object-modeling language and methodology.

An **object-modeling language** is a standardized set of symbols used to model a software system using an object-oriented framework.

A modeling language is usually associated with a methodology for object-oriented development. The modeling language defines the elements of the model.

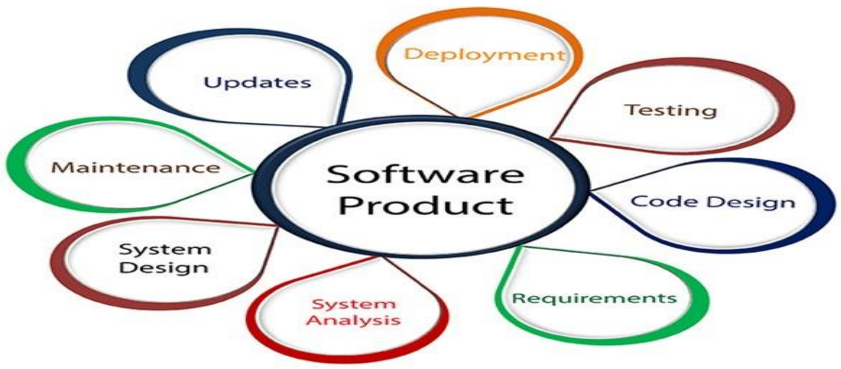
E.g., that a model has classes, methods, object properties, etc. The methodology defines the steps developers and users need to take to develop and maintain a software system. Steps such as Define requirements, Develop code, and Test system.

Methodology is the systematic, theoretical analysis of the methods applied to a field of study. It comprises the theoretical analysis of the body of methods and principles associated with a branch of knowledge.

The term **software engineering** is the product of two words, **software**, and **engineering**. The **software** is a collection of integrated programs.

Software subsists of carefully-organized instructions and code written by developers on any of various particular computer languages. Computer programs and related documentation such as requirements, design models and user manuals.

Engineering is the application of scientific and practical knowledge to invent, design,build, maintain, and improve frameworks, processes, etc.



Define software process.

Software process is defined as the structured set of activities that are required to develop the software system.

What are the fundamental activities of a software process?

1. Specification.
2. Design and implementation.
3. Validation.
4. Evolution

What is System Engineering?

System Engineering means designing, implementing, deploying and operating systems which include hardware ,software and people.

What is requirement engineering?

Requirement engineering is the process of establishing the services that the customer requires from the system and the constraints under which it operates and is developed.

What is Software Engineering?

Software Engineering is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

Why is Software Engineering required?

Software Engineering is required due to the following reasons:

- To manage Large software
- For more Scalability
- Cost Management
- To manage the dynamic nature of software
- For better quality Management

****

Need of Software Engineering

Huge Programming: It is simpler to manufacture a wall than to a house or building, similarly, as the measure of programming become extensive engineering has to step to give it a scientific process.

Adaptability: If the software procedure were not based on scientific and engineering ideas, it would be simpler to re-create new software than to scale an existing one.

Cost: As the hardware industry has demonstrated its skills and huge manufacturing has let down the cost of computer and electronic hardware. But the cost of programming remains high if the proper process is not adapted.

Dynamic Nature: The continually growing and adapting nature of programming hugely depends upon the environment in which the client works. If the quality of the software is continually changing, new upgrades need to be done in the existing one.

Quality Management: Better procedure of software development provides a better and quality software product.

****

Characteristics of a good software engineer

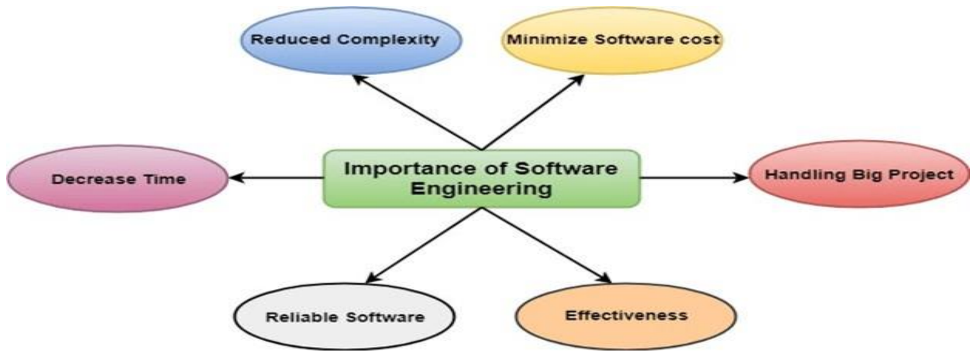
The features that good software engineers should possess are as follows:

- Exposure to systematic methods, i.e., familiarity with software engineeringprinciples.

- Good technical knowledge of the project range (Domain knowledge).
- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation.
- Sound knowledge of fundamentals of computer science.
- Intelligence.
- Ability to work in a team
- Discipline, etc.

****

Importance of Software Engineering



The importance of Software engineering is as follows:

1. **Reduces complexity:** Big software is always complicated and challenging to progress. Software engineering has a great solution to reduce the complication of any project. Software engineering divides big problems into various small issues. And then start solving each small issue one by one. All these small problems are solved independently to each other.
2. **To minimize software cost:** Software needs a lot of hard work and software engineers are highly paid experts. A lot of manpower is required to develop software with a large number of codes. But in software engineering, programmers project everything and decrease all those things that are not needed. In turn, the cost for software productions becomes less as compared to any software that does not use software engineering method.
3. **To decrease time:** Anything that is not made according to the project always wastes time. And if you are making great software, then you may need to run many codes to get the definitive running code. This is a very time-consuming procedure, and if it is not well handled, then this can take a lot of time. So if you are making your software according to the software engineering method, then it will decrease a lot of time.
4. **Handling big projects:** Big projects are not done in a couple of days, and they need lots of patience, planning, and management. And to invest six and seven months of any company, it requires heaps of planning, direction, testing, and maintenance. No one can say that he has given four months of a company to the task,

and the project is still in its first stage. Because the company has provided many resources to the plan and it should be completed. So to handle a big project without any problem, the company has to go for a software engineering method.

5. **Reliable software:** Software should be secure, means if you have delivered the software, then it should work for at least its given time or subscription. And if any bugs come in the software, the company is responsible for solving all these bugs. Because in software engineering, testing and maintenance are given, so there is no worry of its reliability.
6. **Effectiveness:** Effectiveness comes if anything has made according to the standards. Software standards are the big target of companies to make it more effective. So Software becomes more effective in the act with the help of software engineering.

****

<u>NATURE OF SOFTWARE</u>: The nature of the software medium has manyconsequences for systems engineering (SE) of software-intensive systems.

<u>These four properties are:</u>

1. complexity,
2. conformity,
3. changeability,
4. invisibility.

Software and software projects are unique for the following reasons:

- Software has no physical properties;
- Software is the product of intellect-intensive teamwork;
- Estimation and planning for software projects is characterized by a high degree ofuncertainty, which can be at best partially mitigated by best practices;
- Risk management for software projects is predominantly process-oriented;
- Software alone is useless, as it is always a part of a larger system; and
- Software is the most frequently changed element of software intensive systems.

****

<u>TYPES OF THESOFTWARE</u>

Software, which is abbreviated as SW or S/W, is a set of programs that enables the hardware to perform a specific task. All the programs that run the computer are software. The software can be of three types: system software, application software, and programming software.

System Software: The system software is the main software that runs the computer. When you turn on the computer, it activates the hardware and controls and coordinates their functioning. The application programs are also controlled by system software.

Some other examples of system software include:

- Operating System:

○ BIOS.

○ An assembler

2) **Application Software:**

Application software is a set of programs designed to perform a specific task. It does not control the working of a computer as it is designed for end-users. A computer can run without application software. Application software can be easily installed or uninstalled as required. It can be a single program or a collection of small programs. Microsoft Office.

Suite, Adobe Photoshop, and any other software like payroll software or income tax software are application software. As we know, they are designed to perform specific tasks. Accordingly, they can be of different types such as:

1. **Word Processing Software.**
2. **Spreadsheet Software.**
3. **Multimedia Software.**
4. **Enterprise Software.**

3) **Programming Software:**

It is a set or collection of tools that help developers in writing other software or programs. It assists them in creating, debugging, and maintaining software or programs or applications. We can say that these are facilitator software that helps translate programming language such as Java, C++, Python, etc., into machine language code. So, it is not used by end-users. For example, compilers, linkers, debuggers, interpreters, text editors, etc. This software is also called a programming tool or software development tool.

Some examples of programming software include:

1. **Eclipse:** It is a java language editor.
2. **Coda:** It is a programming language editor for Mac.
3. **Notepad++:** It is an open-source editor for windows.
4. **Sublime text:** It is a cross-platform code editor for Linux, Mac, and Windows.

1. **Networking and Web Applications Software** – Networking Software provides the required support necessary for computers to interact with each other and with data storage facilities. The networking software is also used when software is running on a network of computers (such as the World Wide Web). It includes

all network management software, server software, security and encryption software, and software to develop web-based applications like HTML, PHP, XML, etc.

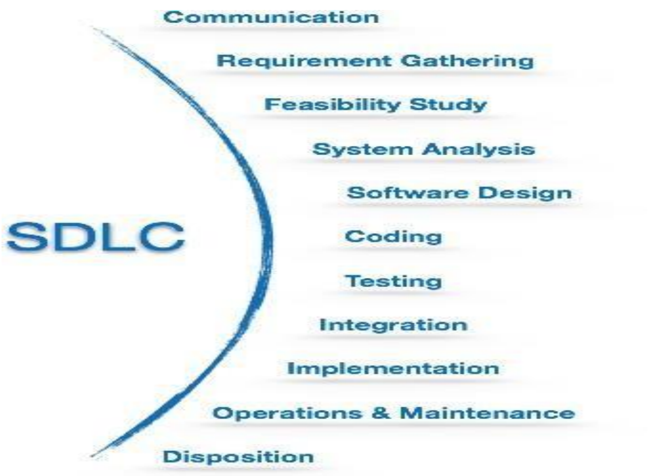
2. **Embedded Software –** This type of software is embedded into the hardware normally in the Read-Only Memory (ROM) as a part of a large system and is used to support certain functionality under the control conditions. Examples are software used in instrumentation and control applications like washing machines, satellites, microwaves, etc.
3. **Reservation Software –** A Reservation system is primarily used to store and retrieve information and perform transactions related to air travel, car rental, hotels, or other activities. They also provide access to bus and railway reservations, although these are not always integrated with the main system. These are also used to relay computerized information for users in the hotel industry, making a reservation and ensuring that the hotel is not overbooked.
4. **Business Software –** This category of software is used to support business applications and is the most widely used category of software. Examples are software for inventory management, accounts, banking, hospitals, schools, stock markets, etc.
5. **Entertainment Software –** Education and entertainment software provides a powerful tool for educational agencies, especially those that deal with educating young children. There is a wide range of entertainment software such as computer games, educational games, translation software, mapping software, etc.
6. **Artificial Intelligence Software –** Software like expert systems, decision support systems, pattern recognition software, artificial neural networks, etc. come under this category. They involve complex problems which are not affected by complex computations using non-numerical algorithms.
7. **Scientific Software –** Scientific and engineering software satisfies the needs of a scientific or engineering user to perform enterprise-specific tasks. Such software is written for specific applications using principles, techniques, and formulae specific to that field. Examples are software like MATLAB, AUTOCAD, PSPICE, ORCAD, etc.
8. **Utilities Software –** The programs coming under this category perform specific tasks and are different from other software in terms of size, cost, and complexity. Examples are anti-virus software, voice recognition software, compression programs, etc.
9. **Document Management Software –** Document Management Software is used to track, manage and store documents in order to reduce the paperwork. Such systems are capable of keeping a record of the various versions created and modified by different users (history tracking). They commonly provide storage, versioning, metadata, security, as well as indexing and retrieval capabilities.

****

<u>Software Engineering Activities:</u>

Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software engineering to develop the intended software product.

SDLC Activities:- SDLC provides a series of steps to be followed to design and develop a software product



efficiently. SDLC framework includes the following steps:

<u>Communication:</u> This is the first step where the user initiates the request for a desired software product. He contacts the service provider and tries to negotiate the terms. He submits his request to the service providing organization in writing.

<u>Requirement Gathering:</u> This step onwards the software development team works to carry on the project. The team holds discussions with various stakeholders from problem domain and tries to bring out as much information as possible on their requirements. The requirements are contemplated and segregated into user requirements, system requirements and functional requirements. The requirements are collected using a number of practices as given -

- studying the existing or obsolete system and software,
- conducting interviews of users and developers,
- referring to the database or
- Collecting answers from the questionnaires.

<u>Feasibility Study:</u> After requirement gathering, the team comes up with a rough plan of software process. At this step the team analyzes if software can be made to fulfill all requirements of the user and if there is any possibility of software being no more useful. It is found out, if the project is financially, practically and technologically feasible for the organization to take up. There are many algorithms available, which help the developers to conclude the feasibility of a software project.

<u>System Analysis:</u> At this step the developers decide a roadmap of their plan and try to bring up the best software model suitable for the project. System analysis includes Understanding of software product limitations, learning system related problems or changes to be done in existing systems beforehand, identifying and addressing the impact of project on organization and personnel etc. The project team analyzes the scope of the project and plans the schedule and resources accordingly.

Software Design: Next step is to bring down whole knowledge of requirements and analysis on the desk and design the software product. The inputs from users and information gathered in requirement gathering phase are the inputs of this step. The output of this step comes in the form of two designs; logical design and physical design. Engineers produce meta-data and data dictionaries, logical diagrams, data-flow diagrams and in some cases pseudo codes.

Coding: This step is also known as programming phase. The implementation of software design starts in terms of writing program code in the suitable programming language and developing error-free executable programs efficiently.

Testing: An estimate says that 50% of whole software development process should be tested. Errors may ruin the software from critical level to its own removal. Software testing is done while coding by the developers and thorough testing is conducted by testing experts at various levels of code such as module testing, program testing, product testing, in-house testing and testing the product at user's end. Early discovery of errors and their remedy is the key to reliable software.

Integration: Software may need to be integrated with the libraries, databases and other program(s). This stage of SDLC is involved in the integration of software with outer world entities.

Implementation: This means installing the software on user machines. At times, software needs post-installation configurations at user end. Software is tested for portability and adaptability and integration related issues are solved during implementation.

Operation and Maintenance: This phase confirms the software operation in terms of more efficiency and less errors. If required, the users are trained on, or aided with the documentation on how to operate the software and how to keep the software operational. The software is maintained timely by updating the code according to the changes taking place in user end environment or technology. This phase may face challenges from hidden bugs and real-world unidentified problems.

Disposition: As time elapses, the software may decline on the performance front. It may go completely obsolete or may need intense up gradation. Hence a pressing need to eliminate a major portion of the system arises. This phase includes archiving data and required software components, closing down the system, planning disposition activity and terminating system at appropriate end-of-system time.

****

Introduction to Object Orientation: Object-Oriented Design

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data. For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data. The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the super class.

The different terms related to object design are:

1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.
3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.
4. **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.
5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.
6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate super classes. This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.
7. **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

****

REUSABILITY IN SOFTWARE ENGINEERING:

Reuse: building on the work and experience of others.

The following are some of the types of reuse practiced by software engineers, in increasing order according to the potential amount of work that can be saved by the reuse.

Reuse of expertise: Software engineers who have many years of experience working on projects can often save considerable time when it comes to developing new systems because they do not need to re- think many issues: their past experience tells them what needs to be done. If such people write articles describing their experiences, this can help others to do better engineering work.

Reuse of standard designs and algorithms. There are thousands of algorithms and other aspects of designs described in various books, standards documents and articles. These represent a tremendous wealth for the software designer, since all he or she needs to do is to implement them if they are appropriate to the current task.

Reuse of libraries of classes or procedures, or of powerful commands built into languages and operating systems. Libraries and commands represent implemented algorithms, data structures and other facilities. Software developers always do this kind of reuse to some extent since all programming languages come with some basic libraries.

Reuse of frameworks. Frameworks are libraries containing the structure of entire applications or subsystems. To complete the application or subsystem, you merely need to fill in certain missing details. A framework can be written in any programming language.

Reuse of complete applications. You can take complete applications and add a small amount of extra software that makes the applications behave in special ways the client wants.

****

<u>Example: POSTALCODES:</u>

The example is divided into three elements. The first element is a hierarchy representing postal codes of different countries. The second element is a new exception class. The third element is the postal test class that allows the user to enter postal codes and test the facilities of the postal code hierarchy.

<u>The postal code hierarchy:</u>

The following are some design decisions you should study in postal code and its subclasses:

Postal code is declared as abstract, meaning that no instance can be created.

Two of its operations, validate and get country, are abstract, meaning that they mustbe given concrete implementations in subclasses.

The operation validate is protected, and is called by the constructor. It's concrete implementations in each subclass will throw a postal code exception if the format ofthe code is invalid.

All the instance variables are declared private. All other classes, including subclasses, can only access them using methods. This helps to improveencapsulation.

There is a to string method, as should be provided in most java classes. There are three examples of subclasses of postal code. Each of these implements the twoabstract operations. For example, the validate method of one subclass,

Canadian postal code, ensures that the format is XNX NXN, where N is a number and X is a letter; the first letter is also taken from a restricted set. The other implementations of validate ensure that US postal codes have an all- numeric format, while British postal codes adhere to their more complex alphanumeric format.

<u>The postal code Exception class:</u>

Postal Code Exception illustrates the concept of the user- defined exception class. Instances of this class are thrown when an error is found while validating a postal code.

A class that manipulates postal codes could choose to handle such exceptions in any way itwishes.

The user interface class Postal Test:

The user interface class, Postal Test, has only a static main method and one private static helper method called get Input. The code prompts the user for input and then attempts to create an instance of one of the subclasses of Postal Code. If a Postal Code Exception is thrown, it tries to create an instance of other subclasses until none remain. Then it prints out information about the result.

It would be possible to put all the code from Postal Test into Postal Code- the mainmethod in Postal Code would then simply be used to test the class.

****

GEOMETRIC POINTS

The classes described in this section represent points on a 2- dimensional plane. From mathematics, we know that to represent a point on a plane, you can use X and y coordinates, which are called Cartesian coordinates. Alternatively, you can use polar coordinates, represented by a radius(often called rho) and an angle(often called theta). In the code we have provided, you can interchangeably work with a given point as Cartesian coordinates or polar coordinates.

Point CP.	**Point CP Test**
Get X ().	Main ()
get Y()	
get Rho()	
get Theta()	

Convert Storage To Cartesian ()

Convert Storage To Polar ()

To String ()

Classes for representing points using both Cartesian and polar coordinates. Only the operations are shown

Java already has classes for representing geometric points. Take a few moments to look at classes Point2D and point in the java documentation. We will call the point class presented here Point CP; it's main distinguishing feature from the built- in java classes is that it can handle both Cartesian and polar coordinates. We also provide a class called Point CP Test which, like Postal Test, simply provides a user interface for testing. The public methods of both classes are shown in figure.

Class Point CP contains two private instances variables that can store X and y, or else rho and theta. No matter which storage format is used, all four possible parameters can be computed. Users of the class can also call methods Convert Storage To Polar or Convert Storage To Cartesian in order to explicitly convert the internal storage of an instance to the alternative format.

****

Requirements Engineering

Domain Analysis:

The process by which a software engineer learns about the domain to betterunderstand the problem.

The domain is the general field of business or technology in which the clientswill use the software.

A domain expert is a person who has a deep knowledge of the domain.

Benefits of performing domain analysis:

1. **Faster development**- you will be able to communicate with the stakeholders moreeffectively, hence you will be able to establish requirements more rapidly.
2. **Better system**- knowing the subtleties of the domain will help ensure that the solutions you adopt will more effectively solve the customers problem.
3. **Anticipation of extensions**- armed with domain knowledge, you will obtaininsights into emerging trends and you will notice opportunities for future development.

Domain analysis document:

1. Introduction: name of the domain and give the motivation for performing the analysis.

2. Glossary: describe the meanings of all terms used in the domain that are either not part of everyday language or else have special meanings.

3. General knowledge about the domain: summarize important facts or rules that are widely known by the domain experts. Such knowledge includes scientific principles, business processes, analysis techniques and how any technology works.

4. Customers and users: describe who will or might buy the software, and in what industrial sectors they operate.

5. The environment: describe the equipment and systems used.

6. Tasks and procedures currently performed: make a list of what the various people do as they go about their work.

7. Competing software: describe what software is available to assist the users and customers, including software that is already in use and software on the market. Discussit's advantages and disadvantages.

8. Similarities to other domains: understanding what is generic versus what is specific will help you to create software that might be more reusable or more widely marketable.

****

PROBLEM DEFINITION AND SCOPE:

A problem can be expressed as difficulty the users or customers are facing (OR) as an opportunity that will result in some benefit such as improved productivityor sales.

The solution to the problem normally will entail developing software.

Defining the scope

Narrow the scope by defining a more precise problem List all the things you might imagine the system doing.

- Exclude some of these things if too broad
- Determine high- level goals if too narrow
- Example: A university registration system
- Initial list of problems with very broad scope:
- Browsing courses, registering, fee payment, room allocation, exam scheduling.
- Narrowed scope: browsing courses, registering, fee payment
- Scope of another system: room allocation, exam scheduling.

In the university registration example you could consider a students goal to be ' completing the registration process'. However, you can see that the students higher level goal might be, ' obtaining their degree in the shortest reasonable time while taking courses that they find most interest and fulfilling'. This new goal sheds a different light on the problem; you might consider adding features to the system that would not otherwise have occurred to you, such as actively proposing courses based on an analysis of the students academic and personal-interest profiles.

****

REQUIREMENTS:

It is a statement describing either

- An aspect of what the proposed system must do, or a constraint on the systemsdevelopment.
- In either case it must contribute in some way towards adequately solving the customers problem;
- The set of requirements as a whole represents a negotiated agreement among thestakeholders.

A collection of requirements is a requirements document.

TYPES OF REQUIREMENTS:

1. Functional requirements. Describes what system should do.

Functional requirements describe what the system should do; in other words, they describethe services provided for the users and for other systems.

The functional requirements should include

1. Everything that a user of the system would need to know regarding what the system does, and
2. Everything that would concern any other system that has to interface to this system.

The functional requirements can be further categorized as follows:

1. What inputs the system should accept
2. What outputs the system should produce

3. What data the system should store that other systems might use
4. What computations the system should perform
5. The timing and synchronization of the above

2.Non functional requirements:

Quality requirements- Constraints on the design to meet specified levels of quality.

Platform requirements- Constraints on the environment and technology of the system

Process requirements.-Constraints on the project plan and development methods.

Quality requirements

a. Quality requirements ensure the system possesses quality attributes such as usability, efficiency, reliability, maintainability and reusability.
b. The following are some of the main categories of quality requirements:
c. **Response time**: for systems that process a lot of data or use a network extensively, you should require that the system gives result or feedback to the user in a certain minimum time.
d. **Throughput**: computations or transactions per minute.
e. **Resource usage**: for systems that use non trivial amounts of such resources as memory and network bandwidth, you should specify the maximum amount of these resources that the system will consume.
f. **Reliability:** reliability is measured as the average amount of time between failures or the probability of a failure in a given period.
g. **Availability**: availability measures the amount of time that a server is runningand available to respond to users.
h. **Recovery from failure**: they state that if the hardware or software crashes, or the power fails, then the system will be able to recover within a certain amount of time, and with a certain minimal loss of data.
i. **Allowances for maintainability and enhancement**: in order to ensure that the system can be adapted in the future, you should describe changes that are anticipated for subsequent releases.
j. **Allowances for reusability**: it is desirable in many cases to specify that a certain percentage of the system,e.g.40% , measured in terms of lines of code, must be designed generically so that it can be reused.

Platform requirements

1. This type of requirements constraints the environment and technology of the system:
2. **Computing platform**: it is normally important to make it clear what hardware
   and operating system the software must be able to work on. Such requirements specify the least powerful platforms and declare that it must work on anything more recent or more powerful.
3. **Technology to be used:** common examples are to specify the programming language or database system. Such requirements are normally stated to ensurethat all systems in an organization use the same technology- this reduces theneed to train people in different technologies.

Process requirements:

The final type of requirements contains the project plan and development methods:

Development process to be used: in order to ensure quality, some requirements documents specify that certain processes be followed. A reference should be made to otherdocuments that describe the process.

Cost and delivery date: These are important constraints. However, they are usually not placed in the requirements document, but are found in the contract for the system or are left to a separate project plan document.

****

<u>Techniques For Gathering And Analysis Requirements</u>

You can gather requirements from the same sources of information as you used for domain analysis: i.e. from the various stakeholders, from other software systems, and from any documentation that might be available.

1.Observation: Taking a notebook and shadowing important potential users as they do their work, writing down everything they do.

You can also ask users to talk as they work, explaining what they are doing. You can videotape the session so that you can analyze it in more detaillater.

<u>2.Interviewing:</u>

1. It is a widely used technique. Conduct a series of interviews.
2. Ask about specific details such as maximum and minimums, whether there areany exceptions to rules and what possible changes might be anticipated.
3. Ask about the stakeholders vision for the future. This question may elicit innovative ideas and suggest what flexibility should be built into the system.
4. If a customer or user presents concrete ideas for their view of the system, ask if they have any alternative ideas.
5. Ask what would be a minimally acceptable solution to the problem.
6. Ask for other sources of information. The stakeholder you are interviewing may have interesting documents or may know someone with usefulknowledge.
7. Have the interviewee draw diagrams. The diagram could show such things as
8. The flow of information, the chain of command, how some technology works.

<u>3.Brainstorming</u>

1. Brainstorming is an effective way to gather information from a group of people.

2. The general idea is that the group sits around a table and discusses some topic with the goal of generating ideas.

3. The following is a suggested approach to organizing and running an effective.

Brainstorming session:

1. Call a meeting with representation from all stakeholders. Effective brainstorming session can be run with five to 20 people.

   Appoint an experienced moderator (also known as a facilitator)- that is, someone who knows how to run brainstorming meeting and will lead the process. The moderator may participate in the discussion if he or she wishes.

2. Arrange the attendees around the periphery of a table and give them plenty of paper to work with.

i. Decide on a trigger question. This is a key step in the process. A trigger question is one for which the participants can provide simple one line answers that are more than just numbers or yes or no responses.

ii. Ask each participant to follow these instructions:

iii. Think of an answer to the trigger question, no matter how trivial orquestionable the answer is

Write the answer down in one or two lines on a sheet of paper, one idea per sheet.

1. Pass the paper to the neighbor on your left to simulate his or herthoughts
2. Look at the answers passed from your neighbor to the right and pass these on to your left as well. Use the ideas you have read to simulate your own ideas.
3. Continue step 5 until ideas stop flowing or a fixed time(5-15 minutes) passes.
4. Moving around the table, ask everybody to read out one of the ideas on the sheets that happen to be In front of them.
5. After a fixed time period, or after all ideas have been recorded on the flip-Chart, the group may take a series of votes to prioritize them.
6. The concept of passing ideas clockwise around the table.

Prototyping:

1. The simplest kind: paper prototype.
2. A set of pictures of the system that are shown to users in sequence to explainwhat would happen.
3. The most common: a mock- up of the system' s UI
4. Written in a rapid prototyping language
5. Does not normally perform any computations, access any databases or interactwith any other systems.
6. May prototype a particular aspect of the system.

****

TYPES OF REQUIREMENTS DOCUMENTS

To perform good software engineering, it is always appropriate to write down requirements. The level of detail of the requirements can, however, vary significantly from project to project. At one extreme, there are documents that informally outline the requirements using a few paragraphs or simple diagrams. At the other extreme, there are specifications that contain thousands of pages of intricate detail.

Requirements documents for large systems are normally arranged in a hierarchy:

Level of detail required in a requirements document

a. **The size of the system**: A large system will need more detailed requirements for several reasons. First, there is simply more to say. Second, the system will need to be divided into subsystems so that different teams can work on each part.

b. **The need to interface to other systems**: Even a small system will need to have well- described requirements if other systems or subsystems are going to use its services or communicate with it.

c. **The target audience**: The requirements must be written at a high enough level so that the potential users can read them.

d. **The contractual arrangements for development**: If you are arranging a contract by which a third party will develop software for you, then you will have to specify the requirements with considerable precision.

e. **The stage in requirements gathering**: At an early stage in requirements gathering, it is important not to write large volumes of precise and detailed requirements.

f. **The level of experience with the domain and the technology:** If you are developing software in a well known domain and using well known technology, then you

should be able to procedure a complete requirements document before starting to design the system.

g**.The cost incurred if the requirements are faulty.** Any system that, if itfails, the environment must be precisely specified.

****

REVIEWING REQUIREMENTS:

Each individual requirement should be carefully reviewed. In order to be acceptable,a requirement should:

1. **Have benefits that outweigh the costs of development**: Cost- benefit analysis is an important skill in software engineering. You sum, in financial terms, the benefits of the requirement and compare this to the sum of the costs.
2. **Be important for the solution of the current problem**. Many ideas might be useful to implement, and might have benefits that outweigh their costs. One of the most important rules in software engineering is the 80-20 rule, which says that 80 % of the usersproblem can often be solved with 20 % of the work. You should initially consider producing only that first 20 % of the system. The 80-20 rule is also called the Pareto principle.
3. **Be expressed using a clear and consistent notation**. Each requirement should be expressed using language that the customers can understand and should be consistent with the other requirements. Requirements are normally expressed in a natural language such as English.
4. **Be unambiguous.** It is typical to find that an English sentence can have more than one interpretation.
5. **Be logically consistent**. You should check consistency with any standards, with other requirements in the document, with higher level requirements and with the requirements for other subsystems.
6. **Lead to a system of sufficient quality.** A requirement should contribute to a system that is sufficiently usable, safe, efficient, reliable and maintainable.

7. **Be realistic with a variable resources**. A requirement is realistic if the development team has the expertise and technology to implement it on the required platform within the budget and time available.
8. **Be verifiable**. The requirements document will not only be the basis for a systems design, but also for testing the system. There must be some way that the system can be tested so as to clearly conclude whether or not the requirement has been correctly implemented.
9. **Be interested uniquely identifiable**. It is important to be able to refer to each individual requirement. This is necessary in requirements review meetings so that people can indicate which requirement they want to discuss. It is also necessary in design documents to be able to say which requirement is being implemented by a given aspect of the design, a quality called traceability. In some documents, each requirement is given a unique number.
10. **The document should be sufficiently complete.** Requirements document should have sections covering the following types ofinformation.

a. **Problem**: provide a succinct description of the problem the system is solving.

b. **Background information**: Give information that will help readers understand the requirements.

c. **Environment and system models**: provide the context in which the system runs and a global overview of the system or subsystem.

d. **Functional requirements**: Describe the services provided to the user and to other systems.

e. **Non functional requirements**: describe any constraints that must be imposed on the design of the system.

****

<u>Case Studies:</u>

Requirements specifications for high assurance secure systems are rare in the open literature. This paper presents a case study in the development of a requirements document for a multilevel secure system. The system is secure, yet combines popular commercial components with specialized high assurance ones. Functional and non-functional requirements pertinent to security are discussed. A multi-dimensional threat model is presented. The threat model accounts for the developmental and operational phases of system evolution and for each phase accounts for both physical and non-physicalthreats.

*****

<u>GPS based Automobile Navigation System:</u>

The following example requirements document is for an embedded system that will be installed in special purpose hardware in cars.

Requirements for GANA software

1. **Problem**. GANA software will help drivers navigate by giving them directions totheir destination.
2. **Background information**. See domain analysis document.
3. **Environment and system models**. GANA software is to run on special GANAhardware, the hardware provides the following to the software:
   a) GPS position information
   b) a wireless internet connection to a map database
   c) position of a trackball
   d) a colour 10 cm by 10 cm LCD screen

   e) six buttons at the bottom of the screen and
   f) Input from the cars other systems containing data about speed and Turing ofthe steering wheel. This requirements document describes the software only.
4. **Functional requirements.**

1. The system uses GPS information to calculate which map to display. The system also integrates information about the cars speed and history of turns made in order to refine it's accuracy about the vehicles location
2. The system has two main interaction modes: in setup mode, the user consults maps and specifies the destination; in navigation mode, the system assists the user to navigate to the destination.
3. Setup mode: When the system is on, and the vehicle is stationary, it enters setup mode. If the vehicle is moving, the system enters navigation mode. In setup mode, the system displays a map. The default map is in 1: 25000 scale and is centered on the users current position. At this scale, the map covers a square with km side.

   When the users current position is within the visible part of the map, the system always indicates it with a red arrow. The arrow points in the direction the user is heading.

   The system also displays in orange the shortest route from the current position to the center of the map. It will not be possible to display the entire route if the current position is not displayed.

   When the user manipulates the trackball, the screen scrolls the map in the direction of rotation of the trackball, as if the user were grabbing the map.

   The LCD screen displays the labels Zoom out, Zoom In , Go current, Go destination, Set Destination and Navigate above the six buttons. The button works as follows:

- Zoom in and zoom out displays new maps. The scale of the map appears at the top right of the screen. There may be a delay retrieving a map, in which case the system displays the message.
- Retrieving map. If the map or network is unavailable for any reason, the system displays: sorry map not available near the top of the screen and continues to display the previous map.
- When the user presses Zoom in , the map scale is doubled so that a smaller region is displayed, with more in detail.
- When the user presses Zoom out the map scale is divided by 2 so that a larger region is displayed, with less local detail..
- When the user presses set destination, the location at the center of the screen is set as the destination. The shortest route from the current position to the destination is highlighted in red and is adjusted as the car moves.
- The shortest route to the set destination is shown on the top of the shortest route to the center of the screen (orange), and hence has precedence.
- When the user presses Go current, the map jumps so that it is centered of the current location.
- When the user presses Go destination, the map jumps so that it is centered over the destination. If no destination has been set, the destination defaults to the current location.
- When the user presses navigate or the vehicle starts moving, the system enters navigation mode described below.

4. Navigation mode:

A detailed map is never displayed in navigation mode since the userwould not be able to concentrate on driving while looking at the map.

If no destination has been set, the system just displays the name ofthe current highway or street and municipality in large type in addition, if a destination has been set, the system displays the following in as large a size as possible:

1. An arrow pointing up if the driver should drive starlight ahead, a left arrow if the driver should turn left, a right arrow if the driver should turn right and a U-turn symbol of the driver should turn around.

   The system displays the labels speak now, volume up, volume down, guide on, guide off and setup above the six buttons. The buttons works as follows:
2. **'Speak now'** produces a computer generated voice, reading the instructionsthat are on display. Every time the user presses the button, any reading in progress is canceled and the instructions are immediately read again startingfrom the beginning.
3. '**Volume up** 'and '**volume down** 'adjust sound output.
4. '**Guide on'** causes a computer generated voice to automatically readthe instructions one minute

in advance of any required driver action, such as exiting the highway, being needed.

5. ‘**Guide off’** cancels this function; the user would have to read the screen or press speak now. In situations where navigational action is required more frequently than once a minute, the voice reads the next instruction as soon asthe system detects that the driver has responded to the previous instruction.
6. **Setup**’ switches to setup mode if the car is stationary. If the car is not stationary, the setup button is heated out and is inactive.

   If the driver does not respond as expected to the instructions, and takes a different route, the system immediately calculates a new route.

5. <u>Quality requirements</u>

1. The system will be robust in the case of failure of the internet connection or failureto receive the GPS signal, maintaining whatever service it can.
2. The system will be designed in a flexible way such that changes in wireless internetor GPS technology can be incorporated in future releases.
3. The system will be designed anticipating incorporation of input from an inertialnavigation unit that would take over in cases where GPS signals fail.

UNIT-I

The UML stands for Unified modeling language, is a standardized general-purpose visual modeling language in the field of Software Engineering. It is used for specifying, visualizing, constructing, and documenting the primary artifacts of the software system. It helps in designing and characterizing, especially those software systems that incorporate the concept of Object orientation. It describes the working of both the software andhardware systems.

UML was created by the Object Management Group (OMG) and UML 1.0specification draft was proposed to the OMG in January 1997.

Characteristics of UML

The UML has the following features:

- It is a generalized modeling language.
- It is distinct from other programming languages like C++, Python, etc.
- It is interrelated to object-oriented analysis and design.
- It is used to visualize the workflow of the system.
- It is a pictorial language, used to generate powerful modeling artifacts.

****

<u>Goals of UML:-</u> A picture is worth a thousand words, this idiom absolutely fits describing UML. Object-oriented concepts were introduced much earlier than UML. At that point of time, there were no standard methodologies to organize and consolidate the object-oriented development. It was then that UML came into picture.

There are a number of goals for developing UML but the most important is to define some general purpose modeling language, which all modelers can use and it also needs to be made simple to understand and use.

UML diagrams are not only made for developers but also for business users, common people, and anybody interested to understand the system. The system can be a software or non-software system. Thus it must be clear that UML is not a development method rather it accompanies with processes to make it a successful system.

In conclusion, the goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today’s complex environment.

****

<u>Modeling Concepts:</u>

Unified Modeling Language (UML) Models represent systems at different levels of detail. Some models describe a system from a higher, more abstract level, while other models provide greater detail. UML models contain model elements, such as actors, use cases, classes, and packages, and one or more diagrams that show a specific perspective of a system. A model can also contain other, more detailed models.

You create and manage models using modeling projects in the Project Explorer view. The contents of a modeling project are organized into three types of logical folders: diagrams, models, and profiles. This structure displays the logical containment of the UML model elements, regardless of where they are stored physically. The models in a modeling project are displayed under the Models folder, or node. These nodes are not the physical model files, which have the .emx as a file name extension, but are the root model elements of the models. Similarly, the corresponding diagrams and profiles are displayed under the Diagrams folder and Profiles folder respectively.

You can use modeling diagrams to capture system use cases in a use-case model during the requirements gathering phase, you define the application domain in an analysis model during the system analysis phase, and you refine the application model in a design model during the detailed design phase.

You can use models to do the following things:

- Visually represent a system that you want to build
- Communicate your vision of a system to customers and colleagues
- Develop and test the architecture of a system
- Use the UML diagrams to direct code generation

****

<u>UML-Building Blocks</u>

UML is composed of three main building blocks, i.e., things, relationships, and diagrams. Building blocks generate one complete UML model diagram by rotating around several different blocks. It plays an essential role in developing UML diagrams. The basic UML building blocks are enlisted below:

1. Things
2. Relationships
3. Diagrams

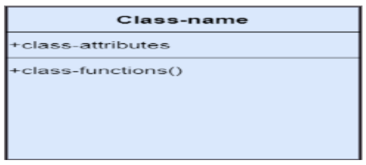
1. <u>Things:</u>

Anything that is a real world entity or object is termed as things. It can be divided into several different categories:

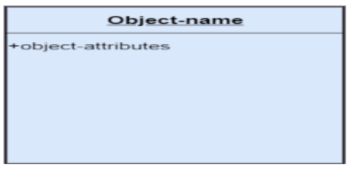
1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

<u>1. Structural things</u>: Nouns that depicts the static behavior of a model is termed as structural things. They display the physical and conceptual components. They include class, object, interface, node, collaboration, component, and a use case.

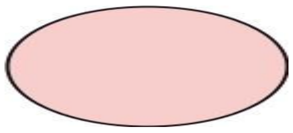
Class: A Class is a set of identical things that outlines the functionality and properties of an object. It also represents the abstract class whose functionalities are not defined. Its notation is as follows;



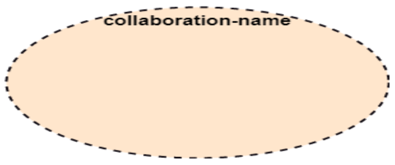
Object: An individual that describes the behavior and the functions of a system. The notation of the object is similar to that of the class; the only difference is that the object name is always underlined and its notation is given below;



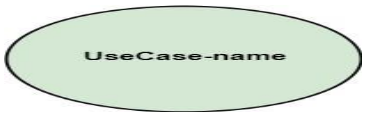
Interface: A set of operations that describes the functionality of a class, which is implemented whenever an interface is implemented.



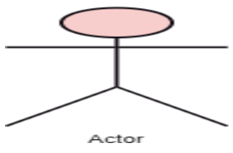
Collaboration: It represents the interaction between things that is done to meet the goal. It is symbolized as a dotted ellipse with its name written inside it.



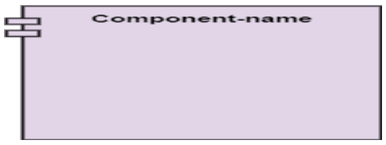
Use case: Use case is the core concept of object-oriented modeling. It portrays a set ofactions executed by a system to achieve the goal.



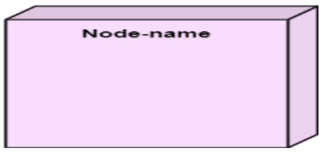
Actor: It comes under the use case diagrams. It is an object that interacts with the system, for example, a user.



Component: It represents the physical part of the system.



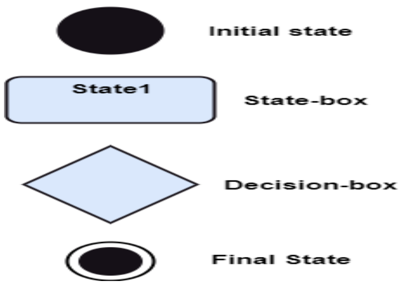
Node: A physical element that exists at run time.



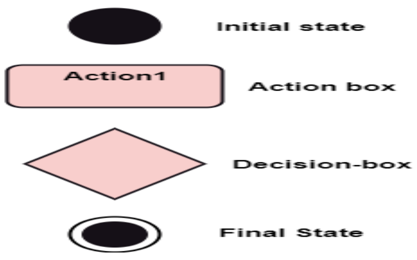
2.Behavioral Things

They are the verbs that encompass the dynamic parts of a model. It depicts the behavior of a system. They involve state machine, activity diagram, interaction diagram, grouping things, annotation things

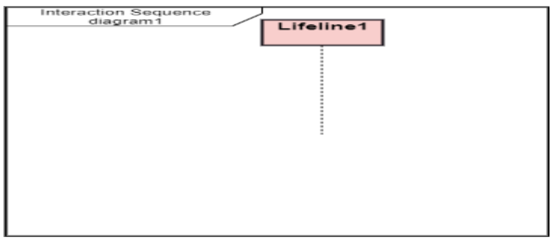
State Machine: It defines a sequence of states that an entity goes through in the software development lifecycle. It keeps a record of several distinct states of a system component.



Activity Diagram: It portrays all the activities accomplished by different entities of a system. It is represented the same as that of a state machine diagram. It consists of an initial state, final state, a decision box, and an action notation.



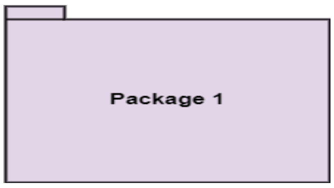
Interaction Diagram: It is used to envision the flow of messages between several components in a system.



3.Grouping Things

It is a method that together binds the elements of the UML model. In UML, the package is the only thing, which is used for grouping.

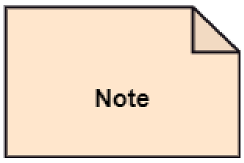
Package: Package is the only thing that is available for grouping behavioral and structural things.



4.Annotation Things

It is a mechanism that captures the remarks, descriptions, and comments of UML model elements. In UML, a note is the only Annotational thing.

Note: It is used to attach the constraints, comments, and rules to the elements of the model. It is a kind of yellow sticky note.



2.Relationships:

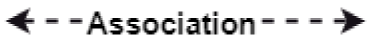
In UML diagrams, relationships are used to link several things. It is a connection between structural, behavioral, or grouping things. There are four types of relationships given below:

1. Association
2. Dependency
3. Generalization
4. Realization

1.Association:

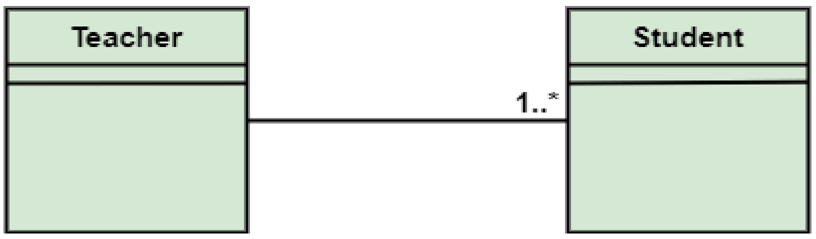
Association relationship is a structural relationship in which different objects are linked within the system. It exhibits a binary relationship between the objects representing an activity. It depicts the relationship between objects, such as a teacher, can be associated with multiple teachers.

It is represented by a line between the classes followed by an arrow that navigates the direction, and when the arrow is on both sides, it is then called a bidirectional association. We can specify the multiplicity of an association by adding the adornments on the line that will denote the association.

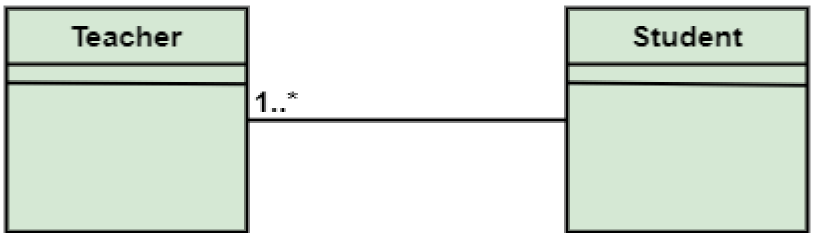


Example:

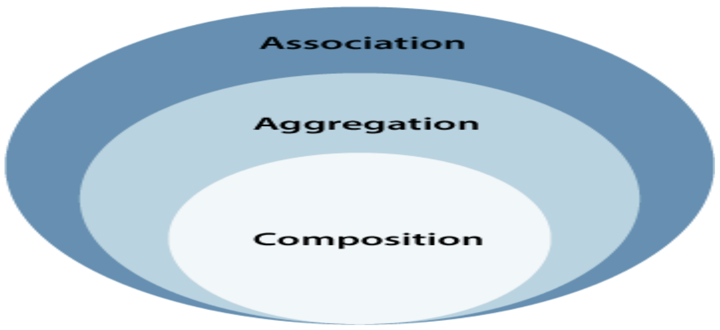
1) A single teacher has multiple students.



2) A single student can associate with many teachers.



The composition and aggregation are two subsets of association. In both of the cases, the object of one class is owned by the object of another class; the only difference is that in composition, the child does not exist independently of its parent, whereas in aggregation, the child is not dependent on its parent i.e., standalone. An aggregation is a special form of association, and composition is the special form of aggregation.

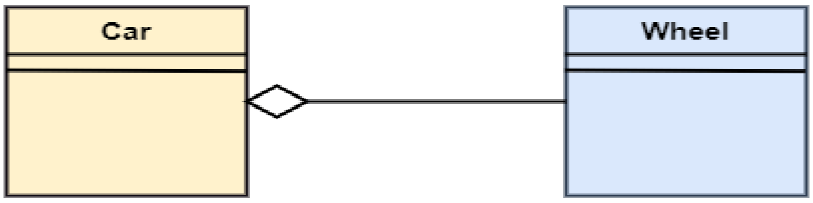


Aggregation:

Aggregation is a subset of association, is a collection of different things. It represents has a relationship. It is more specific than an association. It describes a part-whole or part-of relationship. It is a binary association, i.e., it only involves two classes. It is a kind of relationship in which the child is independent of its parent.

For example:

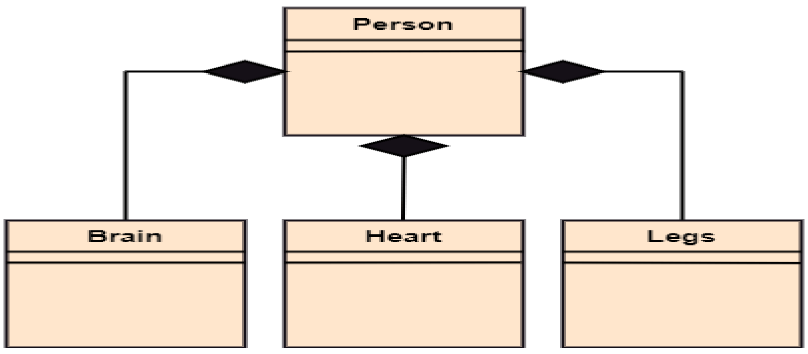
Here we are considering a car and a wheel example. A car cannot move without a wheel. But the wheel can be independently used with the bike, scooter, cycle, or any other vehicle. The wheel object can exist without the car object, which proves to be an aggregation relationship.



Composition:

The composition is a part of aggregation, and it portrays the whole-part relationship. It depicts dependency between a composite (parent) and its parts (children), which means that if the composite is discarded, so will its parts get deleted. It exists between similar objects.

As you can see from the example given below, the composition association relationship connects the Person class with Brain class, Heart class, and Legs class. If the person is destroyed, the brain, heart, and legs will also get discarded.

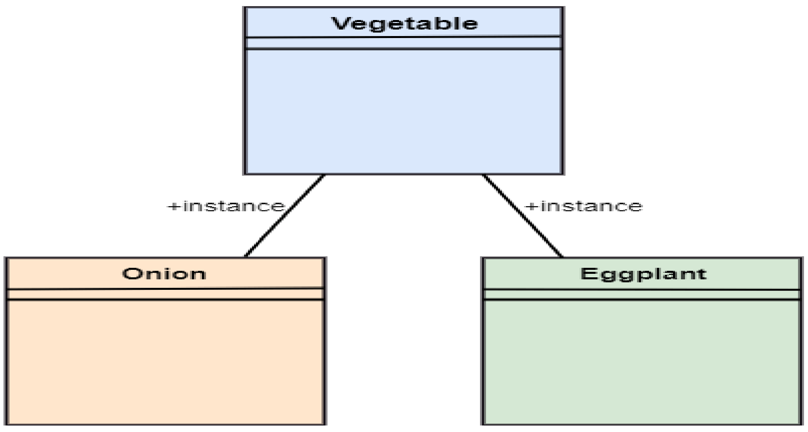


UML- Association:

Association is the semantic relationship between classes that shows how one instance is connected or merged with others in a system. The objects are combined either logically or physically.

Reflexive Association:

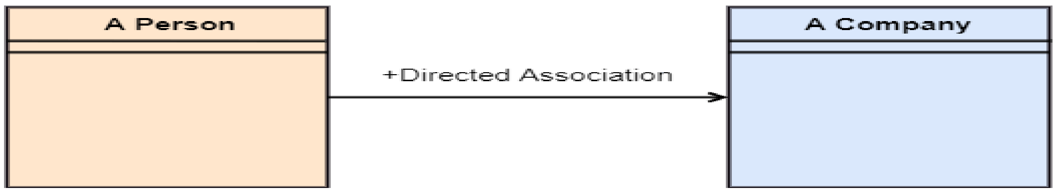
In the reflexive associations, the links are between the objects of the same classes. In other words, it can be said that the reflexive association consists of the same class at both ends. An object can also be termed as an instance.



Directed Association:

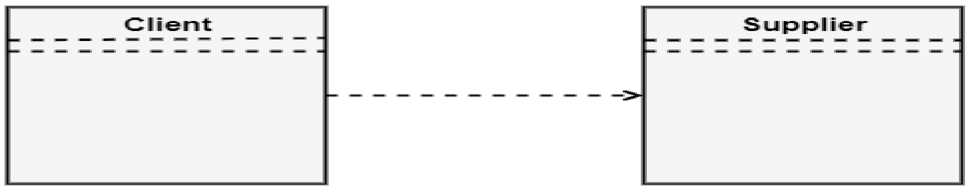
The directed association is concerned with the direction of flow inside association classes. The flow of association can be shown by employing a directed association. The directed association between two classes is represented by a line with an arrowhead, which indicates the navigation direction. The flow of association from one class to another is always in one direction.

It can be said that there is an association between a person and the company. The person works for the company. Here the person works for the company, and not the company works for a person.



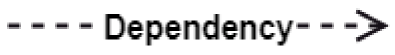
2.UML-Dependency

Dependency depicts how various things within a system are dependent on each other. In UML, a dependency relationship is the kind of relationship in which a client (one element) is dependent on the supplier (another element). It is used in class diagrams, component diagrams, deployment diagrams, and use-case diagrams, which indicates that a change to the supplier necessitates a change to the client. An example is given below:



Dependency: Dependency is a kind of relationship in which a change in target element affects the source element, or simply we can say the source element is dependent on the target element. It is one of the most important notations in UML. It depicts the dependency from one entity to another.

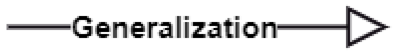
It is denoted by a dotted line followed by an arrow at one side as shown below,

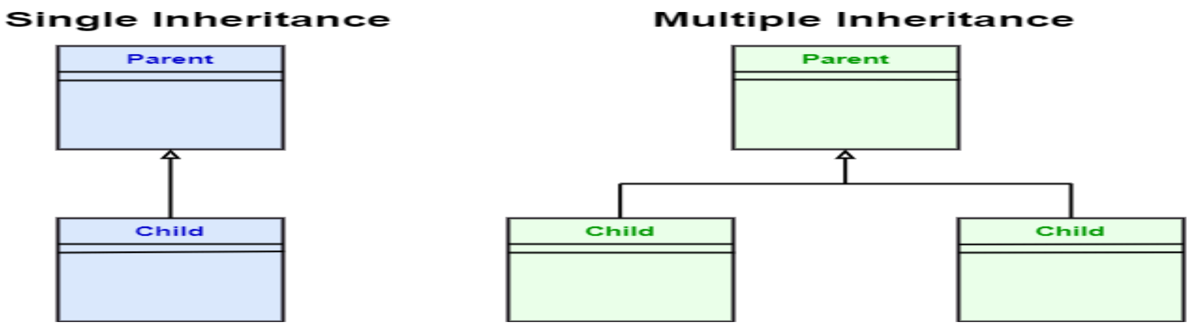


UML-Generalization:

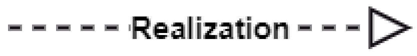
In UML modeling, a generalization relationship is a relationship that implements the concept of object orientation called inheritance. The generalization relationship occurs between two

entities or objects, such that one entity is the parent, and the other one is the child. The child inherits the functionality of its parent and can access as well as update it.
It is denoted by a straight line followed by an empty arrowhead at one side.





Realization: It is a semantic kind of relationship between two things, where one defines the behavior to be carried out, and the other one implements the mentioned behavior. It exists in interfaces.
It is denoted by a dotted line with an empty arrowhead at one side.



TYPES OF UML DIAGRAMS

The diagrams are the graphical implementation of the models that incorporate symbols and text. Each symbol has a different meaning in the context of the UML diagram. And each diagram many sets of a different dimension, perspective, and view of the system.
UML diagrams are classified into three categories that are given below:

1. Structural Diagram
2. Behavioral Diagram
3. Interaction Diagram

1. Structural Diagrams:- The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.
These static parts are represented by classes, interfaces, objects, components, andnodes. The four structural diagrams are –

i. Class diagram
ii. Object diagram
iii. Component diagram
iv. .Deployment diagram

i. Class Diagram:- Class diagrams are the most common diagrams used in UML. Class

diagram consists of classes, interfaces, associations, and collaboration. Class diagrams basically represent the object-oriented view of a system, which is static in nature.
Active class is used in a class diagram to represent the concurrency of the system.
Class diagram represents the object orientation of a system. Hence, it is generally used for development purpose. This is the most widely used diagram at the time of system construction.
ii. Object Diagram:- Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system.
Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.
The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.
iii. Component Diagram:- Component diagrams represent a set of components and their relationships. These components consist of classes, interfaces, or collaborations. Component diagrams represent the implementation view of a system.
During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components.
Finally, it can be said component diagrams are used to visualize the implementation.

iv.Deployment Diagram:- Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.
Deployment diagrams are used for visualizing the deployment view of a system. This is generally used by the deployment team.
Note −Component diagrams are dependent upon the classes, interfaces, etc. which are part of class/object diagram. Again, the deployment diagram is dependent upon the components, which are used to make component diagrams.
2. Behavioral Diagram: Any system can have two aspects, static and dynamic. So, a model is considered as complete when both the aspects are fully covered.
Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system.
UML has the following five types of behavioral diagrams.
i.Use case diagram
ii. Statechart diagram
iii.Activity diagram
i. Use Case Diagram:-Use case diagrams are a set of use cases, actors, and their

relationships. They represent the use case view of a system.

A use case represents a particular functionality of a system. Hence, use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as **actors**.

ii.<u>Statechart Diagram:-</u> Any real-time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system.

Statechart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface, etc.

State chart diagram is used to visualize the reaction of a system by internal/external factors.

<u>iii.Activity Diagram:-</u> Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.

Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system.

Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.

<u>3.Interaction diagram:</u>

It is a subset of behavioral diagrams. It depicts the interaction between two objects and the data flow between them. Following are the several interaction diagrams in UML:

- Sequence diagram.
- Collaboration diagram.

<u>i.Sequence Diagram:-</u> A sequence diagram is an interaction diagram. From the name, it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another.

Interaction among the components of a system is very important from implementation and execution perspective. Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.

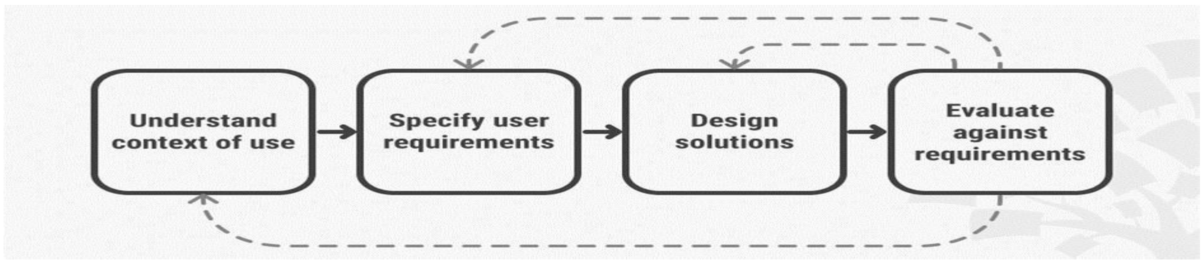
<u>ii.Collaboration Diagram:-</u> Collaboration diagram is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links.

The purpose of collaboration diagram is similar to sequence diagram. However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.

****

<u>USER-CENTERED DESIGN</u>:

User-centered design (UCD) is an iterative design process in which designers focus on the users and their needs in each phase of the design process. In UCD, design teams involve users throughout the design process via a variety of research and design techniques, to create highly usable and accessible products for them. In user- centered design, designers use a mixture of *investigative* methods and tools (e.g., surveys and interviews) and *generative* ones (e.g., brainstorming) to develop an understanding of user needs. Generally, each iteration of the UCD approach involves four distinct phases. First, as designers working in teams, we try to understand the *context* in which users may use a system. Then, we identify and specify the users' *requirements*. A *design* phase follows, in which the design team develops solutions. The team then proceeds to an *evaluation* phase. Here, you assess the outcomes of the evaluation against the users' context and requirements, to check how well a design is performing. More specifically, you see how close it is to a level that matches the users' specific context and satisfies all of their relevant needs. From here, your team makes further iterations of these four phases, andyou continue until the evaluation results are satisfactory.


Understand context of use
Specify user requirements
Design solutions
Evaluate against requirements

****

CHARACTERISTICS OF USERS

Software engineers must develop an understanding of the users

- Goals for using the system
- Potential patterns of use
- Demographics
- Knowledge of the domain and of computers
- Physical ability
- Psychological traits and emotional feelings

****

Developing Use Case Models of Systems:

The Use-case model is defined as a model which is used to show how users interact with the system in order to solve a problem. As such, the use case model defines the user's objective, the interactions between the system and the user, and the system's behavior required to meet these objectives.

Various model elements are contained in use-case model, such as actors, use cases, and the association between them.

We use a use-case diagram to graphically portray a subset of the model in order to make the communication simpler. There will regularly be a numerous use-case diagram which is related to the given model, each demonstrating a subset of the model components related to a specific purpose. A similar model component might be appearing on a few use-case diagrams; however, each use-case should be consistent. If, in order to handle the use-case model, tools are used then this consistency restriction is automated so that any variations to the component of the model (changing the name, for instance) will be reflected automatically on each use-case diagram, which shows that component.

Packages may include a use-case model, which is used to organize the model to simplify the analysis, planning, navigation, communication, development and maintenance.

Various use-case models are textual and the text captured in the use-case specifications, which are linked with the element of every use-case model. The flow of events of the use case is described with the help of these specifications.

The use-case model acts as an integrated thread in the development of the entire system. The use-case model is used like the main specification of the system functional requirements as the basis for design and analysis, as the basis for user documentation, as the basis of defining test cases, and as an input to iteration planning.

Origin of Use-Case:-

Nowadays, use case modeling is frequently connected with UML, in spite of the factthat it has been presented before UML existed. Its short history is:

- Ivar Jacobson, in the year 1986, originally formulated textual and visual modeling methods to specify use cases.
- And in the year 1992, his co-authored book named Object-Oriented Software Engineering - A Use Case Driven Approach, assisted with promoting the strategy for catching functional requirements, particularly in software development.

Components of Basic Model:-

There are various components of the basic model:

1. Actor
2. Use Case
3. Associations

1.Actor:- Usually, actors are people involved with the system defined on the basis of their roles. An actor can be anything such as human or another external system.

2.Use Case:- The use case defines how actors use a system to accomplish a specific objective. The use cases are generally introduced by the user to meet the objectives of the activities and variants involved in the achievement of the goal.

3.**Associations:-** Associations are another component of the basic model. It is used to define the associations among actors and use cases they contribute in. This association is called communicates-association.

Advanced Model Components:-

There are various components of the advanced model:

1. Subject
2. Use-Case Package
3. Generalizations
4. Dependencies

1.Subject:- The subject component is used to represent the boundary of the system of interest.

2.**Use-Case Package:-** We use the model component in order to structure the use case model to make simpler the analysis, planning, navigation, and communication. Suppose there are various actors or use cases. In that case, we can also use use-case packages inorder to further structure the use-case model in much the similar way we use directories or folders to organize the information on our hard-disk.

For various reasons, we divide the use-case model into the use-case packages,containing:

- To help parallel development by partitioning the problem into bite-sized parts.
- To improve communication with various stakeholders by making packaging containing actors, use cases and related to the specific stakeholder.

3.Generalizations:- Generalizations mean the association between the actors in orderto help re-use of common properties.

4.Dependencies:- In UML, various types of dependencies are defined between use cases. In particular, <<include>> and <<extend>>.

We use <<include>> dependency to comprise shared behavior from an included usecase into a base use case to use common behavior.

We use <<extend>> dependency to include optional behavior from an extended use-case into an extended use case.

****

USE CASE DIAGRAM

A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. It depicts the high-level functionality of a system and also tells how the user handles a system.

Purpose of Use Case Diagrams:

The main purpose of a use case diagram is to portray the dynamic aspect of a system. It accumulates the system's requirement, which includes both internal as well as external influences. It invokes persons, use cases, and several things that invoke the actors and elements accountable for the implementation of use case diagrams. It represents how an entity from the external environment can interact with a part of the system.

Following are the purposes of a use case diagram given below:

1. It gathers the system's needs.
2. It depicts the external view of the system.
3. It recognizes the internal as well as external factors that influence the system.
4. It represents the interaction between the actors.

How to draw a Use Case diagram?

It is essential to analyze the whole system before starting with drawing a use case diagram, and then the system's functionalities are found. And once every single functionality is identified, they are then transformed into the use cases to be used in the use case diagram.

After that, we will enlist the actors that will interact with the system. The actors are the person or a thing that invokes the functionality of a system. It may be a system or a private entity, such that it requires an entity to be pertinent to the functionalities of the system to which it is going to interact.

Once both the actors and use cases are enlisted, the relation between the actor and use case/ system is inspected. It identifies the no of times an actor communicates with the system. Basically, an actor can interact multiple times with a use case or system at a particular instance of time.

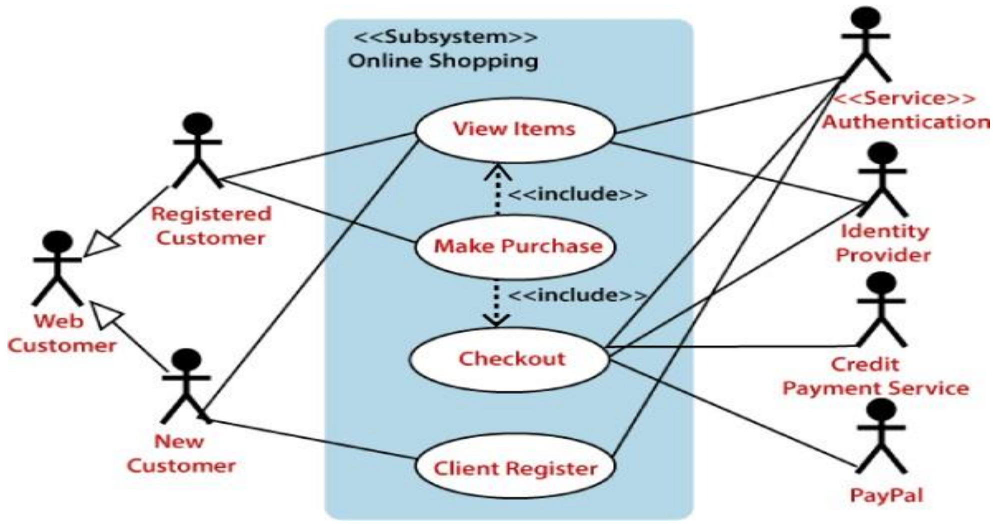
Following are some rules that must be followed while drawing a use case diagram:

1. A pertinent and meaningful name should be assigned to the actor or a use case of asystem.
2. The communication of an actor with a use case must be defined in anunderstandable way.
3. Specified notations to be used as and when required.
4. The most significant interactions should be represented among the multiple no of interactions between the use case and actors.

Example of a Use Case Diagram

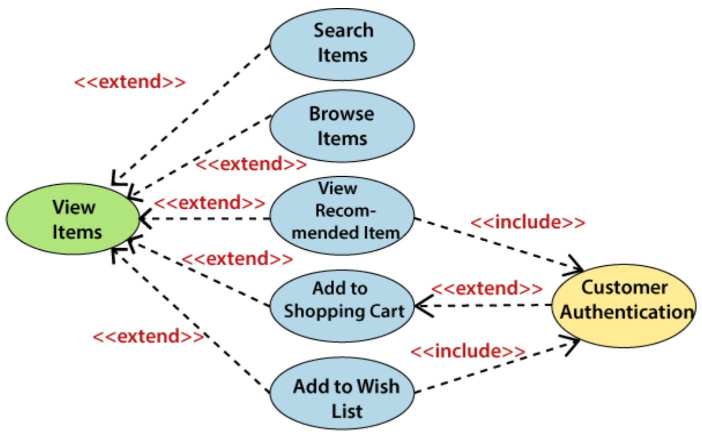
A use case diagram depicting the Online Shopping website is given below.

Here the Web Customer actor makes use of any online shopping website to purchase online. The top-level uses are as follows; View Items, Make Purchase, Checkout, Client Register. The **View Items** use case is utilized by the customer who searches and view products. The **Client Register** use case allows the customer to register itself with the website for availing gift vouchers, coupons, or getting a private sale invitation. It is to be noted that the **Checkout** is an included use case, which is part of **Making Purchase,** and itis not available by itself.



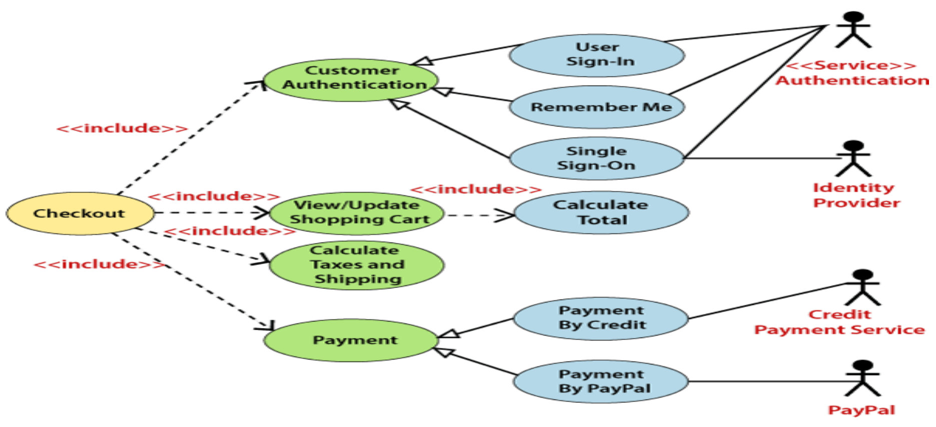
View Items is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item. The View Items is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item.

Both **View Recommended Item** and **Add to Wish List** include the Customer Authentication use case, as they necessitate authenticated customers, and simultaneously item can be added to the shopping cart without any user authentication.



Similarly, the **Checkout** use case also includes the following use cases, as shown below. It requires an authenticated Web Customer, which can be done by login page, user authentication cookie ("Remember me"), or Single Sign-On (SSO). SSO needs an external identity provider's participation, while Web site authentication service is utilized in all these use cases.

The Checkout use case involves Payment use case that can be done either by the credit card and external credit payment services or with PayPal.



THE BASICS OF USER INTERFACE DESIGN

The visual part of a computer application or operating system through which a client interacts with a computer or software. It determines how commands are given to the computer or the program and how data is displayed on the screen.

Types of User Interface

There are two main types of User Interface:

- Text-Based User Interface or Command Line Interface
- Graphical User Interface (GUI)

Text-Based User Interface: This method relies primarily on the keyboard. A typical example of this is UNIX.

Advantages:

- Many and easier to customizations options.
- Typically capable of more important tasks.

Disadvantages:

- Relies heavily on recall rather than recognition.
- Navigation is often more difficult.

Graphical User Interface (GUI): GUI relies much more heavily on the mouse. A typical example of this type of interface is any versions of the Windows operating systems.

Advantages:

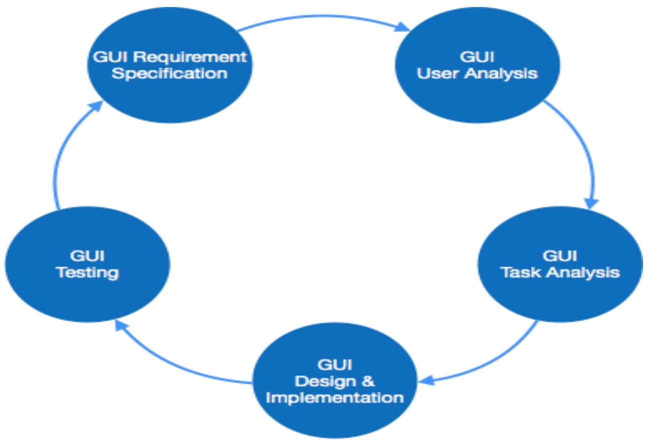
- Less expert knowledge is required to use it.
- Easier to Navigate and can look through folders quickly in a guess and check manner.
- The user may switch quickly from one task to another and can interact with several different applications.

Disadvantages:

- Typically decreased options.
- Usually less customizable. Not easy to use one button for tons of different variations.

User Interface Design Activities:-

There are a number of activities performed for designing user interface. The process of GUI design and implementation is alike SDLC. Any model can be used for GUI implementation among Waterfall, Iterative or Spiral Model.
A model used for GUI design and development should fulfill these GUI specific steps.



- GUI Requirement Gathering.
- User Analysis.
- Task Analysis
- GUI Design & implementation.
- Testing .

****

USABILITY PRINCIPLES

A well designed user *interface* is comprehensible and controllable, helping users to complete their work successfully and efficiently, and to feel competent and satisfied. *Effective* user interfaces are designed based on principles of human interface design. The principles listed below are consolidated from a wide range of published sources (Constantine & Lockwood, 1999; Cooper & Reimann, 2003; Gerhardt-Powals, 1996; Lidwell, Holden & Butler, 2003; Nielsen, 1994; Schneiderman, 1998; Tognazzini, 2003) and are based on a long history of human-computer interaction research, cognitive psychology, and design best practices.

Usefulness

Value: The system should provide necessary utilities and address the real needs of users. **Relevance:** The information and functions provided to the user should be relevant to the user's *task* and context.

Consistency

Consistency and standards: Follow appropriate standards/conventions for the platform and the suite of products. Within an application (or a suite of applications), make sure that actions, terminology, and commands are used consistently.

Real-world conventions: Use commonly understood concepts, terms and metaphors, follow real-world conventions (when appropriate), and present information in a natural and logical order.

Simplicity

Simplicity: Reduce clutter and eliminate any unnecessary or irrelevant elements.

Visibility: Keep the most commonly used options for a task visible (and the other options easily accessible).

Self-evidency: Design a system to be usable without instruction by the appropriate target user of the system: if appropriate, by a member of the general public or by a user who has the appropriate subject-matter knowledge but no prior experience with the system. Display data in a manner that is clear and obvious to the appropriate user.

Communication

Feedback: Provide appropriate, clear, and timely feedback to the user so that he sees the results of his actions and knows what is going on with the system.

Structure: Use organization to reinforce meaning. Put related things together, and keep unrelated things separate.

Sequencing: Organize groups of actions with a beginning, middle, and end, so that users know where they are, when they are done, and have the satisfaction of accomplishment.

Help and documentation: Ensure that any instructions are concise and focused on supporting the user's task.

Error Prevention and Handling

Forgiveness: Allow reasonable variations in input. Prevent the user from making serious errors whenever possible, and ask for user confirmation before allowing a potentially destructive action.

Error recovery: Provide clear, plain-language messages to describe the problem and suggest a solution to help users recover from any errors.

Undo and redo: Provide "emergency exits" to allow users to abandon an unwanted action. The ability to reverse actions relieves anxiety and encourages user exploration of unfamiliar options.

Efficiency

Efficacy: (For frequent use) Accommodate a user’s continuous advancement in knowledge and skill. Do not impede efficient use by a skilled, experienced user.

Shortcuts: (For frequent use) Allow experienced users to work more quickly by providing abbreviations, function keys, macros, or other accelerators, and allowing customization or tailoring of frequent actions.

User control: (For experienced users) Make users the initiators of actions rather than the responders to increase the users' sense that they are in charge of the system.

Workload Reduction

Supportive automation: Make the user's work easier, simpler, faster, or more fun. Automate unwanted workload.

Reduce memory load: Keep displays brief and simple, consolidate and summarize data, and present new information with meaningful aids to interpretation. Do not require the user to remember information. Allow recognition rather than recall.

Free cognitive resources for high-level tasks: Eliminate mental calculations, estimations, comparisons, and unnecessary thinking. Reduce uncertainty.

Usability Judgment

It depends: There will often be tradeoffs involved in design, and the situation, sound judgment, experience should guide how those tradeoffs are weighed.

A foolish consistency...: There are times when it makes sense to bend or violate some of the principles or guidelines, but make sure that the violation is intentional and appropriate.

Class Design and Class Diagrams

A Class is a blueprint that is used to create Object. The Class defines what object can do.

What is Class Diagram?

UML CLASS DIAGRAM gives an overview of a software system by displaying classes, attributes, operations, and their relationships. This Diagram includes the class name, attributes, and operation in separate designated compartments.

Class Diagram defines the types of objects in the system and the different types of relationships that exist among them. It gives a high-level view of an application. This modeling method can run with almost all Object-Oriented Methods. A class can refer to another class. A class can haveits objects or may inherit from other classes.

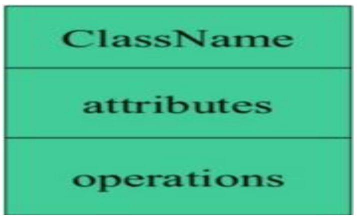
Class Diagram helps construct the code for the software application development.

ESSENTIAL ELEMENTS OF A UML CLASS DIAGRAM

Essential elements of UML class diagram are:

1. Class Name
2. Attributes
3. Operations

Class Name



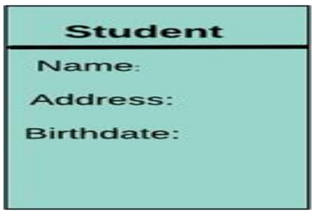
The name of the class is only needed in the graphical representation of the class. It appears in the topmost compartment. A class is the blueprint of an object which can share the same relationships, attributes, operations, & semantics. The class is rendered as a rectangle, including its name, attributes, and operations in seperate compartments.

Following rules must be taken care of while representing a class:

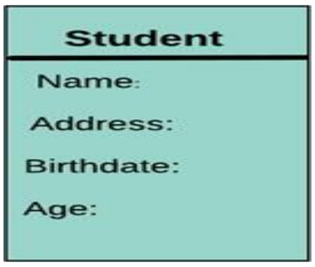
1. A class name should always start with a capital letter.
2. A class name should always be in the center of the first compartment.
3. A class name should always be written in **bold** format.
4. An abstract class name should be written in italics format.

Attrbutes:

An attribute is named property of a class which describes the object being modeled. In the class diagram, this component is placed just below the name-compartment.



A derived attribute is computed from other attributes. For example, an age of the student can be easily computed from his/her birth date.



Attributes characteristics

- The attributes are generally written along with the visibility factor.
- Public, private, protected and package are the four visibilities which are denoted by +, -,#, or ~ signs respectively.
- Visibility describes the accessibility of an attribute of aclass.
- Attributes must have a meaningful name that describes the use of it in a class.

Relationships

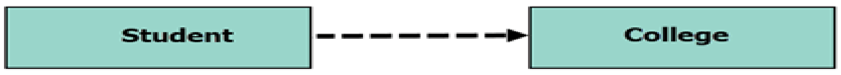
There are mainly three kinds of relationships in UML:

1. Dependencies
2. Generalizations
3. Associations

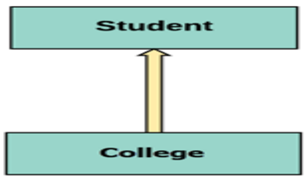
Dependency

A dependency means the relation between two or more classes in which a change in one may force changes in the other. However, it will always create a weaker relationship. Dependency indicates that one class depends on another.

In the following example, Student has a dependency on College



GENERALIZATION:



A generalization helps to connect a subclass to its superclass. A sub-class is inherited from its superclass. Generalization relationship can't be used to model interface implementation. Class diagram allows inheriting from multiple superclasses.

In this example, the class Student is generalized from Person Class.

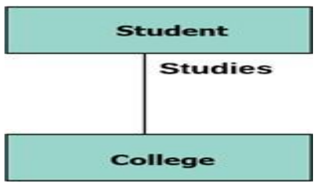
ASSOCIATION:

This kind of relationship represents static relationships between classes A and B. For example;an employee works for an organization.

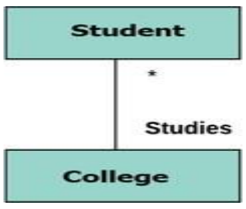
Here are some rules for Association:

- Association is mostly verb or a verb phrase or noun or noun phrase.
- It should be named to indicate the role played by the class attached at the end of theassociation path.
- Mandatory for reflexive associations

In this example, the relationship between student and college is shown which are studies.



MULTIPLICITY

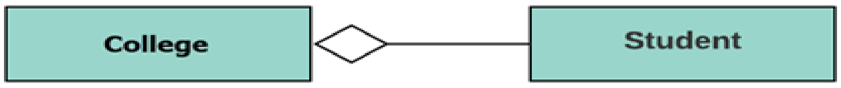


A multiplicity is a factor associated with an attribute. It specifies how many instances of attributes are created when a class is initialized. If a multiplicity is not specified, by default oneis considered as a default multiplicity.

Let's say that that there are 100 students in one college. The college can have multiple students.

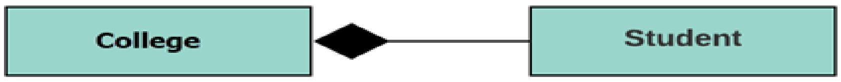
Aggregation

Aggregation is a special type of association that models a whole- part relationship between aggregate and its parts.



For example, the class college is made up of one or more student. In aggregation, the contained classes are never totally dependent on the lifecycle of the container. Here, the college class will remain even if the student is not available.

Composition:



The composition is a special type of aggregation which denotes strong ownership between two classes when one class is a part of another class.

For example, if college is composed of classes student. The college could contain many students, while each student belongs to only one college. So, if college is not functioning all the students also removed.

INTERACTION DIAGRAMS: From the term Interaction, it is clear that the diagram is used to describe some type of interactions among the different elements in the model. This interaction is a part of dynamic behavior of the system.

This interactive behaviour is represented in UML by two diagrams known as **Sequence diagram** and **Collaboration diagram**. The basic purpose of both the diagrams are similar.

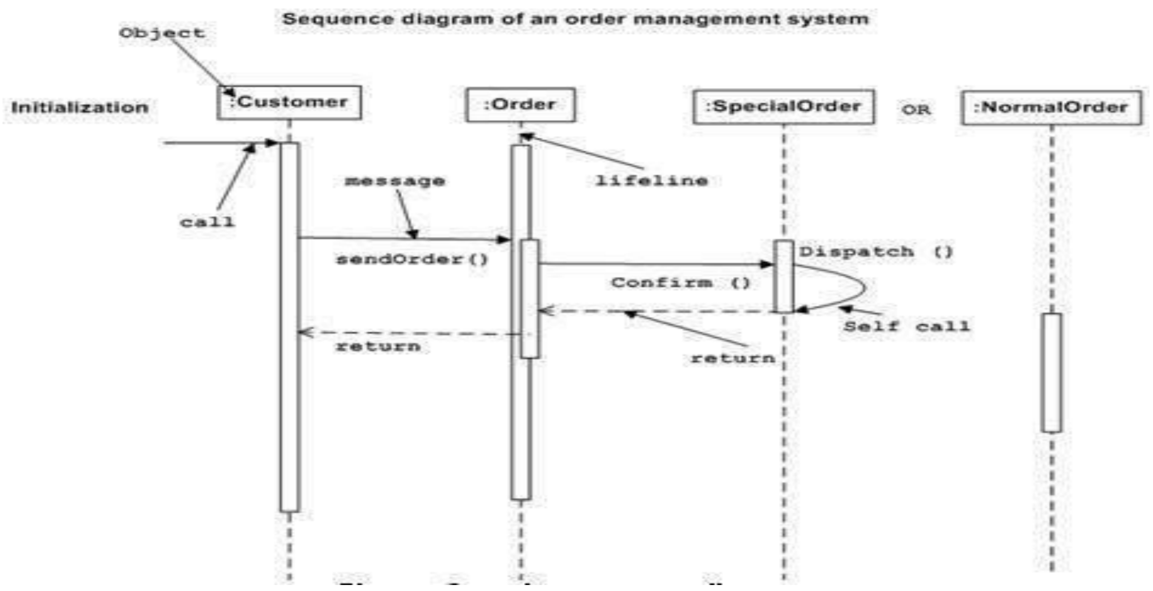
Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

Following are two interaction diagrams modeling the order management system. Thefirst diagram is a sequence diagram and the second is a collaboration diagram

1. The Sequence Diagram:- The sequence diagram has four objects (Customer, Order,SpecialOrder and NormalOrder).

The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

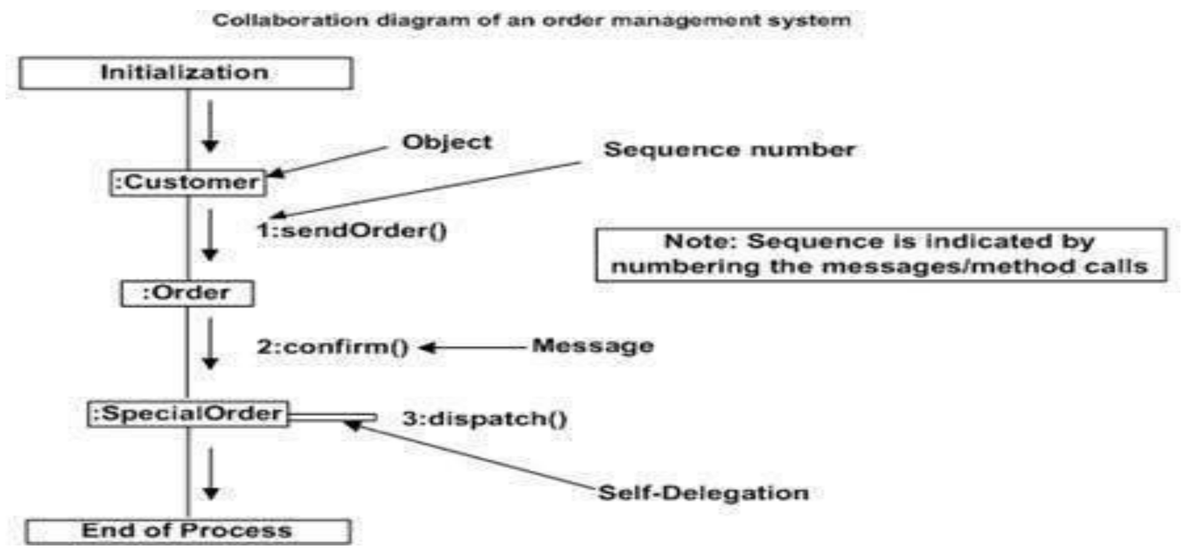
The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.



2. The Collaboration Diagram:- The second interaction diagram is the collaboration diagram. It shows the object organization as seen in the following diagram. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. We havetaken the same order management system to describe the collaboration diagram.

Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization.

To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used.



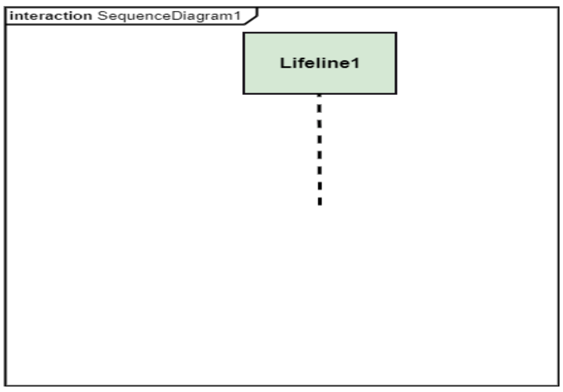
Purpose of Interaction Diagrams:

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

The purpose of interaction diagram is −

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.



UML STATE MACHINE DIAGRAM

The state machine diagram is also called the Statechart or State Transition diagram, which shows the order of states underwent by an object within the system. It captures the software system's behavior. It models the behavior of a class, a subsystem, a package, and a complete system.

It tends out to be an efficient way of modeling the interactions and collaborations in the external entities and the system. It models event-based systems to handle the state of an object. It also defines several distinct states of a component within the system. Each object/component has a specific state.

Following are the types of a state machine diagram that are given below:

1. **Behavioral state machine**

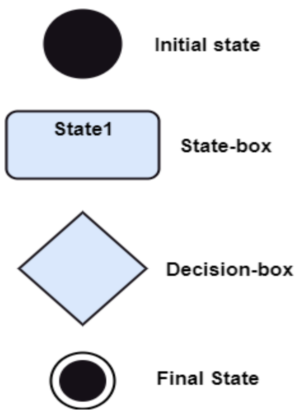
The behavioral state machine diagram records the behavior of an object within thesystem. It depicts an implementation of a particular entity. It models the behavior of the system.

2. **Protocol state machine**

It captures the behavior of the protocol. The protocol state machine depicts the change in the state of the protocol and parallel changes within the system. But itdoes not portray the implementation of a particular component.

Notation of a State Machine Diagram

Following are the notations of a state machine diagram enlisted below:



a. **Initial state:** It defines the initial state (beginning) of a system, and it isrepresented by a black filled circle.

b. **Final state:** It represents the final state (end) of a system. It is denoted by a filledcircle present within a circle.

c. **Decision box:** It is of diamond shape that represents the decisions to be made onthe basis of an evaluated guard.

d. **Transition:** A change of control from one state to another due to the occurrence of some event is termed as a transition. It is represented by an arrow labeled with an event due to which the change has ensued.

e. **State box:** It depicts the conditions or circumstances of a particular object of a class at a specific point of time. A rectangle with round corners is used to represent the state box.

<u>**ACTIVITY DIAGRAMS**</u>

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc.

Purpose of Activity Diagrams

The basic purposes of activity diagrams is similar to other four diagrams. It captures the dynamic behavior of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is sometimes considered as the flowchart. Although the diagrams look like a flowchart, they are not. It shows different flows such as parallel, branched, concurrent, and single.

The purpose of an activity diagram can be described as −

- 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 Following are the rules

that are to be followed for drawing an activity diagram:

1. A meaningful name should be given to each and every activity.
2. Identify all of the constraints.
3. Acknowledge the activity associations.

Notation of an Activity diagram

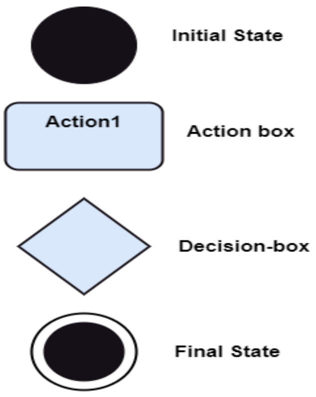
Activity diagram constitutes following notations:

Initial State: It depicts the initial stage or beginning of the set of actions.

Final State: It is the stage where all the control flows and object flows end.

Decision Box: It makes sure that the control flow or object flow will follow only onepath.

Action Box: It represents the set of actions that are to be performed.



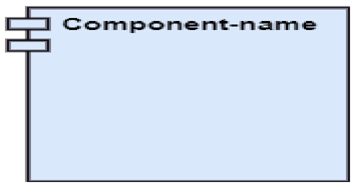
UML COMPONENT DIAGRAM

A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable. It models the physical view of a system such as executables, files, libraries, etc. that resides within the node.

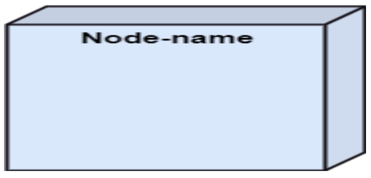
It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

Notation of a Component Diagram

a) A component



b) A node



Purpose of a Component Diagram

Since it is a special kind of a UML diagram, it holds distinct purposes. It describes all the individual components that are used to make the functionalities, but not the functionalities of the system. It visualizes the physical components inside the system. The components can be a library, packages, files, etc.

The component diagram also describes the static view of a system, which includes the organization of components at a particular instant. The collection of component diagrams represents a whole system.

The main purpose of the component diagram are enlisted below:

1. It envisions each component of a system.
2. It constructs the executable by incorporating forward and reverse engineering.
3. It depicts the relationships and organization of components.

When to use a Component Diagram?

It represents various physical components of a system at runtime. It is helpful in visualizing the structure and the organization of a system. It describes how individual components can together form a single system. Following are some reasons, which tells when to use component diagram:

1. To divide a single system into multiple components according to the functionality.
2. To represent the component organization of the system.

<u>DEPLOYMENT DIAGRAM</u>

Deployment diagrams are used to visualize the topology of the physical components of a system, where the software components are deployed.

Deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationship

Purpose of Deployment Diagram

The main purpose of the deployment diagram is to represent how software is installed on the hardware component. It depicts in what manner a software interacts with hardware to perform its execution.

Both the deployment diagram and the component diagram are closely interrelated to each other as they focus on software and hardware components. The component diagram represents the components of a system, whereas the deployment diagram describes how they are actually deployed on the hardware.

The deployment diagram does not focus on the logical components of the system, but it put its attention on the hardware topology.

Following are the purposes of deployment diagram enlisted below:

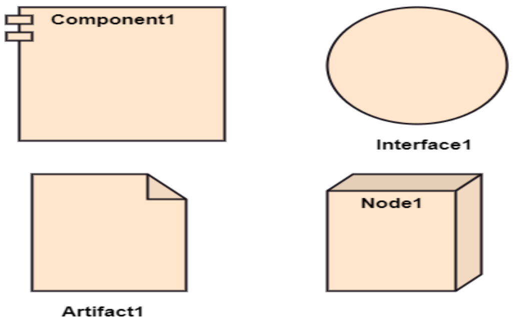
1. To envision the hardware topology of the system.
2. To represent the hardware components on which the software components are installed.
3. To describe the processing of nodes at the runtime.

Symbol and notation of Deployment diagram

The deployment diagram consists of the following notations:

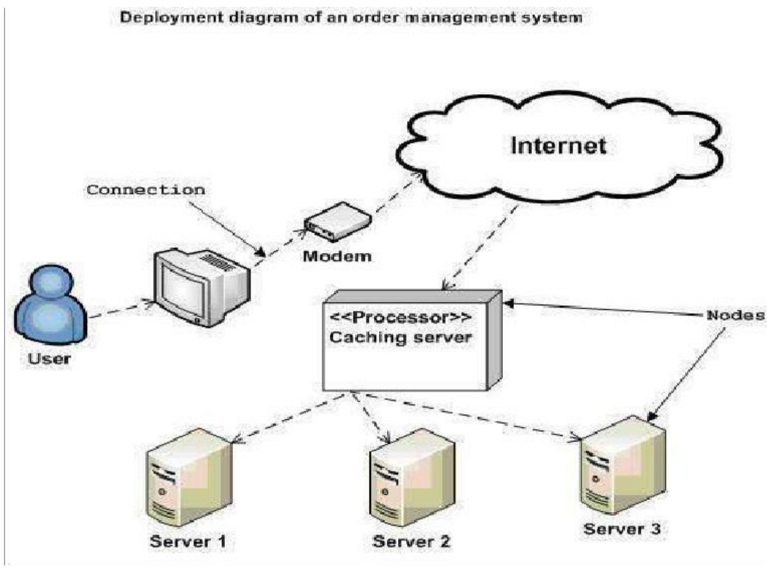
1. A component

2. An artifact
3. An interface
4. A node



Deployment diagrams can be used for the followings:

1. To model the network and hardware topology of a system.
2. To model the distributed networks and systems.
3. Implement forwarding and reverse engineering processes.
4. To model the hardware details for a client/server system.
5. For modeling the embedded system.



UNIT-3

Software Design and Architecture

Software Design (or) Design process:

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programminglanguage.

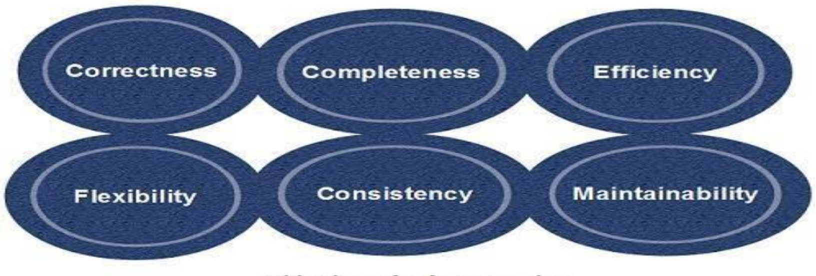
The software design phase is the first step in **SDLC (Software Design Life Cycle)**, which moves the concentration from the problem domain to the solution domain. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.

Software Design Levels:- Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design-** The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Objectives of Software Design

Following are the purposes of Software design:



Objectives of Software Design

1. **Correctness:** Software design should be correct as per requirement.
2. **Completeness:** The design should have all components like data structures,modules, and external interfaces, etc.
3. **Efficiency:** Resources should be used efficiently by the program.

4. **Flexibility:** Able to modify on changing needs.
5. **Consistency:** There should not be any inconsistency in the design.
6. **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

****

<u>Principles Leading to Good Design:</u>

Many principles are employed to organize, coordinate, classify, and set up software design's structural components. Software Designs become some of the most convenient designs when the following principles are applied. They help to generate remarkable User Experiences and customer loyalty.

The principles of a good software design are:

1. Modularity
2. Coupling
3. Abstraction
4. Anticipation of change
5. Simplicity
6. Sufficiency and completeness

<u>Modularity:-</u>Dividing a large software project into smaller portions/modules is known as modularity. It is the key to scalable and maintainable software design. The project is divided into various components and work on one component is done at once. It becomes easy to test each component due to modularity. It also makes integrating new features more accessible.

<u>Advantages of Modularity:</u>

- It allows large programs to be written by several or different people.
- It provides more checkpoints to measure progress.
- It provides a framework for complete testing, more accessible to test.
- It produced the well designed and more readable program.

<u>Disadvantages of Modularity:</u>

- Execution time maybe, but not certainly, longer.
- Storage size perhaps, but is not certainly, increased.
- Compilation and loading time may be longer.
- Inter-module communication problems may be increased.

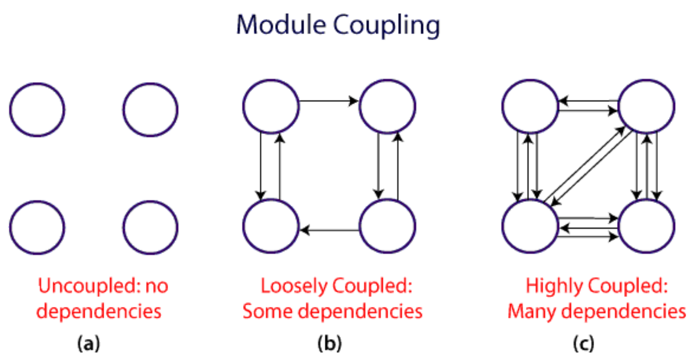
<u>1. Coupling:-</u> Coupling refers to the extent of interdependence between software modules and how closely two modules are connected. Low coupling is a feature of good design. With low coupling, changes can be made in each module individually, without changing the other modules.

Coupling: It measures the relative interdependence among modules.

Module Coupling:

In software engineering, the coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

The various types of coupling techniques are shown in fig:


Module Coupling
Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

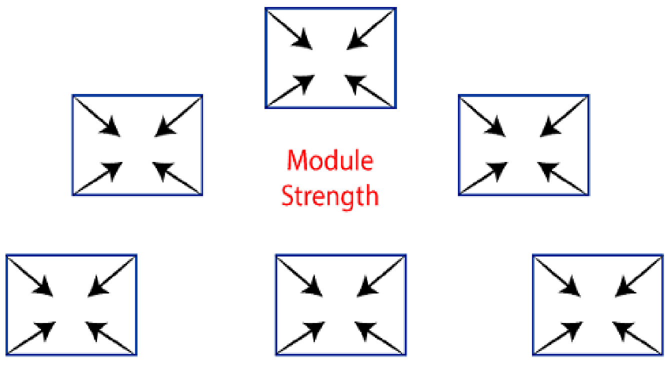
A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Cohesion: It measures the relative function strength of a module.

Module Cohesion:

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."



Cohesion= Strength of relations within Modules

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Modu inding.
Coupling shows the relationships betweenmodules.	Cohesion shows the relationship withinthe odule.
Coupling shows the lative **independence** between the modules.	Cohesion shows the module's lative **functional** strength.
While creating, you should aim for low coupling, ependency among modules should be less.	While creating you should aim for high cohesi e., a cohesive component/ module focuses o ngle function (i.e., single-mindedness) with li teraction with other modules of the system.
In coupling, modules are linked to the othermodules.	In cohesion, the module focuses on asingle thir

2. Abstraction:- The process of identifying the essential behavior by separating it from its implementation and removing irrelevant details is known as Abstraction. The inability to separate essential behavior from its implementation will lead to unnecessary coupling.

3. Anticipation of Change:- The demands of software keep on changing, resulting in continuous changes in requirements as well. Building a good software design consists of its ability to accommodate and adjust to change comfortably.

4. Simplicity:- The aim of good software design is simplicity. Each task has its own module, which can be utilized and modified independently. It makes the code easy to use and minimizes the number of setbacks.

5. Sufficiency and Completeness:- A good software design ensures the sufficiency and completeness of the software concerning the established requirements. It makes sure that the software has been adequately and wholly built.

****

TECHNIQUES FOR MAKING GOOD DESIGN DECISIONS

Using priorities and objectives to decide among alternatives

- Step 1: List and describe the alternatives for the design decision.
- Step 2: List the advantages and disadvantages of each alternative with respect toyour objectives and priorities.
- Step 3: Determine whether any of the alternatives prevents you from meeting oneor more of the objectives.

- Step 4 : Choose the alternative that helps you to best meet your objectives.
- Step 5: Adjust priorities for subsequent decision making.

Example priorities and objectives:

Imagine a system has the following objectives, starting with top priority:

- **Security**: Encryption must not be breakable within 100 hours of computing time ona 400Mhz Intel processor, using known cryptanalysis techniques.
- **Maintainability:** No specific objective.
- **CPU efficiency**: Must respond to the user within one second when running on a400 MHz intel processor.
- **Network bandwidth efficiency**: Must not require transmission of more than 8KBof data per transaction.
- **Memory efficiency**: Must not consume over 20MB of RAM.
- **Portability**: Must be able to run on Windows 98,NT 4 and ME as well as Linux.

Using cost-benefit analysis to choose among alternatives

- To estimate the costs add up:

  The incremental cost of doing the software engineering work, including ongoing maintenance.

  The incremental costs of any development technology required.

  The incremental costs that end-users and product support personnel will experience.

- To estimate the benefits, add up:

  The incremental software engineering time saved.

  The incremental benefits measured in terms of either increased sales or elsefinancial benefit to users.

****

<u>WRITING A GOOD DESIGN DOCUMENT</u>

Design document is an aid to make better designs.

- They force you to be explicit and consider the important issues before starting implementation.
- They allow a group of people to review the design and therefore to improve it.
- Design documents as a means of communication. To those who will be implementing the design.

  To those who will need, in the future, to modify the design.

  To those who need to create systems or subsystems that interface withthe system being designed.

Structure of design document:

A. Purpose: What system or part of the system this design document describes. Make reference to the requirements that are being implemented by this design.

B. **General priorities**: Describe the priorities used to guide the design process.

C. **Outline of the design**: Give a high level description of the design that allows the reader to quickly get a general feeling for it.

D. **Major design issues**: Discuss the important issues that had to be resolved.

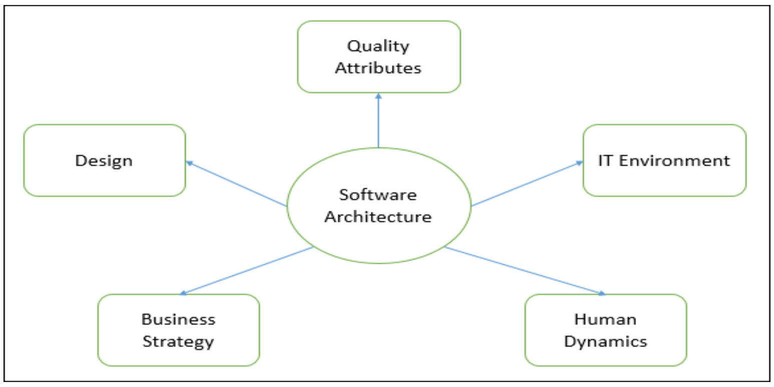
E. **Other details of the design**: Give any other details the reader may want to know that have not yet been mentioned.

****

SOFTWARE ARCHITECTURE

Software Architecture typically refers to the bigger structures of a software system, and it deals with how multiple software processes cooperate to carry out their tasks. **Software Design** refers to the smaller structures and it deals with the internal design of a single software process.

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In **Architecture**, nonfunctional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

Software Architecture:

Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

1. It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

2. Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of –

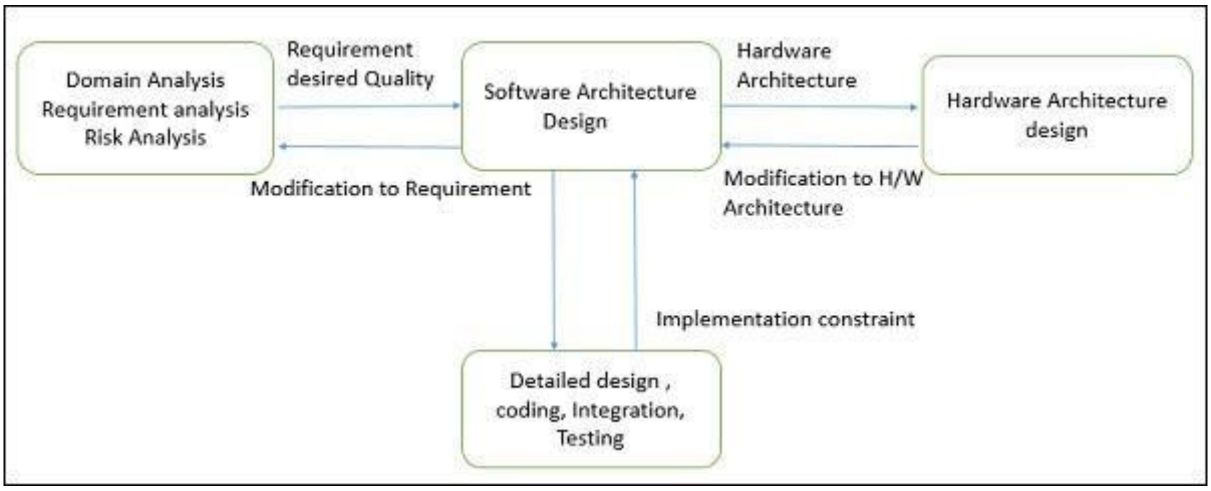
1. Selection of structural elements and their interfaces by which the system is composed.

2. Behavior as specified in collaborations among those elements.
3. Composition of these structural and behavioral elements into large subsystem.
4. Architectural decisions align with business objectives.
5. Architectural styles guide the organization.

Software Design:

Software design provides a **design plan** that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows −

1. To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
2. Act as a blueprint during the development process.
3. Guide the implementation tasks, including detailed design, coding, integration, andtesting. It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.



Goals of Architecture

The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with
building a technical solution and builds a bridge between business and technical requirements.
Some of the other goals are as follows −

1. Expose the structure of the system, but hide its implementation details.
2. Realize all the use-cases and scenarios.
3. Try to address the requirements of various stakeholders.
4. Handle both functional and quality requirements.
5. Reduce the goal of ownership and improve the organization's market position.
6. Improve quality and functionality offered by the system.
7. Improve external confidence in either the organization or system.

Limitations:

1. Lack of tools and standardized ways to represent architecture.
2. Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
3. Lack of awareness of the importance of architectural design to software development.
4. Lack of understanding of the role of software architect and poor communication among stakeholders.
5. Lack of understanding of the design process, design experience and evaluation of design.

****

ARCHITECTURAL PATTERNS

The architectural pattern shows how a solution can be used to solve a reoccurring problem. In another word, it reflects how a code or components interact with each other. Moreover, the architectural pattern is describing the architectural style of our system and provides solutions for the issues in our architectural style. Personally, I prefer to define architectural patterns as a way to implement our architectural style.

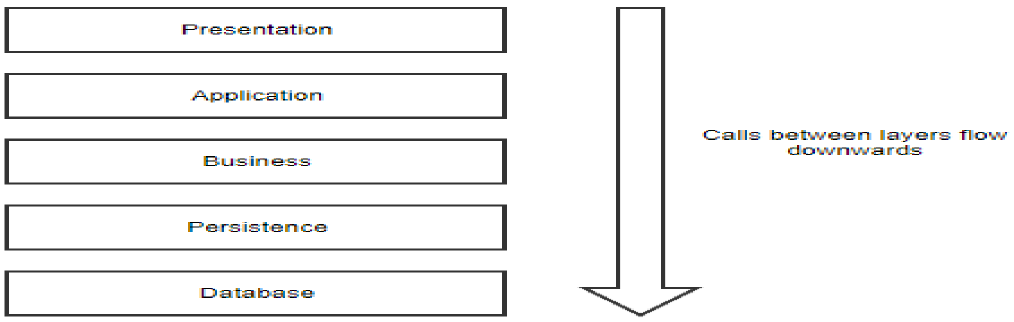
1. MULTI-LAYER ARCHITECTURAL PATTERNS:

The layered pattern is probably one of the most well-known software architecture patterns.Many developers use it, without really knowing its name. The idea is to split up your code into “layers”, where each layer has a certain responsibility and provides a service to a higher layer.

There isn’t a predefined number of layers, but these are the ones you see most often:

1. Presentation or UI layer
2. Application layer
3. Business or domain layer
4. Persistence or data access layer
5. Database layer

The idea is that the user initiates a piece of code in the presentation layer by performing some action (e.g. clicking a button). The presentation layer then calls the underlying layer, i.e. the application layer. Then we go into the business layer and finally, the persistence layer stores everything in the database. So higher layers are dependent upon and make calls to the lower layers.



You will see variations of this, depending on the complexity of the applications. Some applications might omit the application layer, while others add a caching layer. It's even possible to merge two layers into one.

Advantages

- Most developers are familiar with this pattern.
- It provides an easy way of writing a well-organized and testable application.

Disadvantages

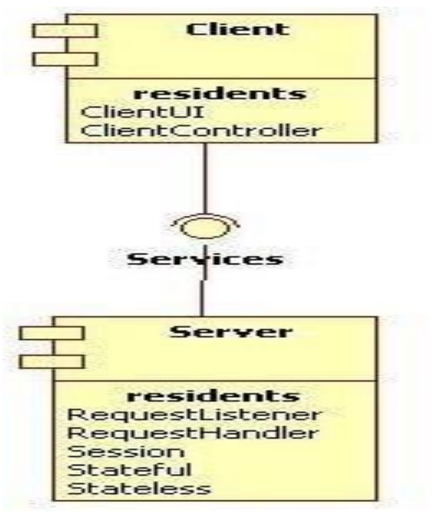
- It tends to lead to monolithic applications that are hard to split up afterward.
- Developers often find themselves writing a lot of code to pass through the different layers, without adding any value in these layers. If all you are doing is writing a simple CRUD application, the layered pattern might be overkill for you.

2.THE CLIENT-SERVER PATTERN:

A Client-Server Architecture consists of two types of components: clients and servers. A server component perpetually listens for requests from client components. When a request is received, the server processes the request, and then sends a response back to the client. Servers may be further classified as stateless or stateful. Clients of a stateful server may make composite requests that consist of multiple atomic requests. This enables a more conversational or transactional interactions between client and server. To accomplish this, a stateful server keeps a record of the requests from each current client. This record is called a session.

In order to simultaneously process requests from multiple clients, a server often uses the Master-Slave Pattern. In this case the Master perpetually listens for client requests. When a request is received, the master creates a slave to processes the request, and then resumes listening. Meanwhile, the slave performs all subsequent communication with the client.

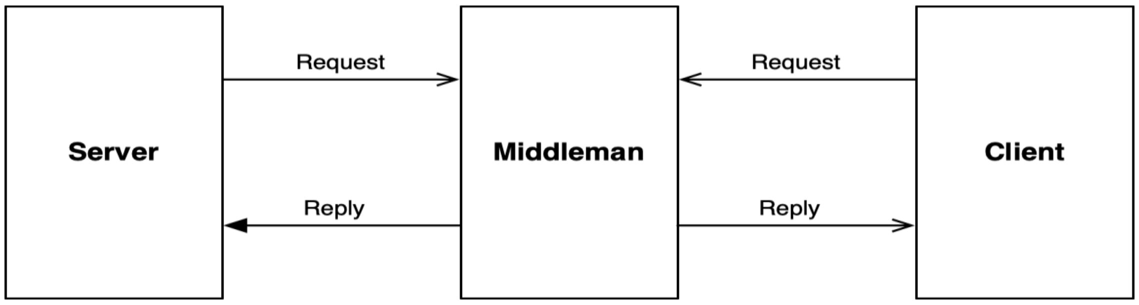
Here is a simple component diagram showing a server component that implements operations specified in a Services interface, and a client component that depends on these services.



3. **THE BROKER ARCHITECTURAL PATTERN:**

Transparently distribute aspects of the software system to different nodes.

1. An object can call method of another object without knowing that thisobject is remotely located.
2. COBRA is a well known open standard that allows you to build this kind ofarchitecture.

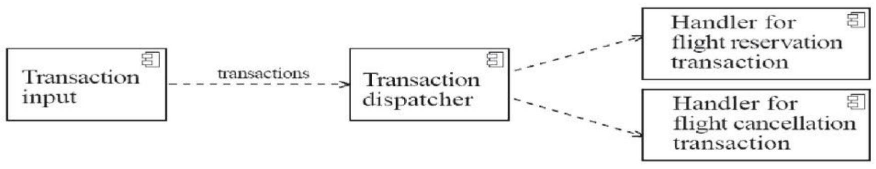


4. **THE TRANSACTION- PROCESSING ARCHITECTURAL PATTERN:**

A process reads a series of inputs one by one.

1. Each input describes a transaction- a command that typically some changeto the data stored by the system.
2. There is a transaction dispatcher component that decides what to do witheach transaction.
3. This dispatches a procedure call or message to one of a series of componentthat will handle the transaction.

Example of a transaction-processing system


Transaction input
transactions
Transaction dispatcher
Handler for flight reservation transaction
Handler for flight cancellation transaction

THE PIPE AND FILTER ARCHITECTURAL PATTERN:

A stream of data , in a relatively simple format, is passed through a series ofprocesses

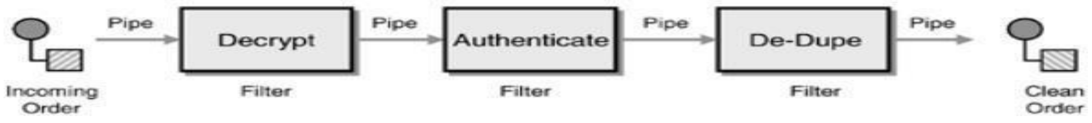
1.Each of which transforms it in some way.

2.Data is constantly fed into the pipeline.

3.The processes work concurrently.

4.The architecture is very flexible.

- Almost all the components could be removed.
- Components could be replaced.
- New components could be inserted.
- Certain components could be reordered.



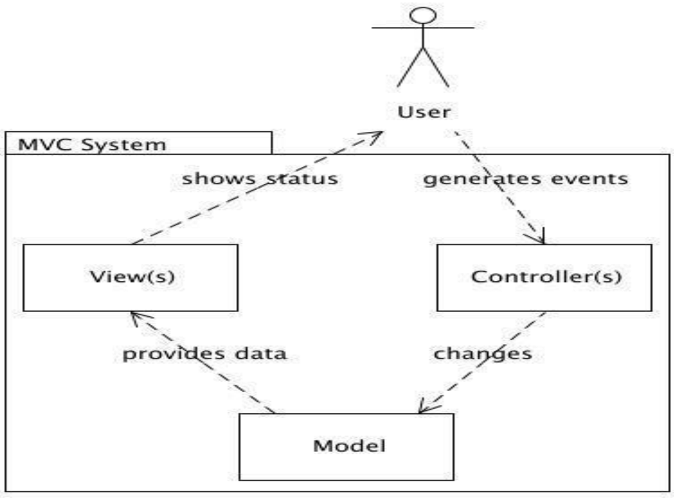
5. THE MODEL-VIEW-CONTROLLER(MVC) ARCHITECTURAL PATTERN:

An architectural pattern used to separate the user interface layer from other parts of thesystem.

1. The model contains the underlying classes whose instances are to be viewedand manipulated.
2. The view contains objects used to render the appearance of the data fromthe model in the user interface.
3. The controller contains the objects that control and handle the usersinteraction with the view and the model.
4. The observable design pattern is normally used to separate the model fromthe view.

Advantages

- Multiple developers can work simultaneously on the model, controller and views.
- MVC enables logical grouping of related actions on a controller together. The viewsfor a specific model are also grouped together.
- Models can have multiple views.



Design Patterns

Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples.

Usage of Design Pattern

Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

Types of Design Patterns

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software** , there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. We'll also discuss another category of design pattern: J2EE design patterns.

S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than stantiating objects directly using new operator. This gives program moreflexibility in deciding which ɔjects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance isused to compose terfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.
4	**J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identifiec / Sun Java Center.

1. THE ABSTRACTION-OCCURRENCE PATTERN

- **Context:**

1. Often in a domain model you find a set of related objects(*occurrences)*.
2. The members of such a set share common information
3. But also differ from each other in important ways.

- **Problem:**

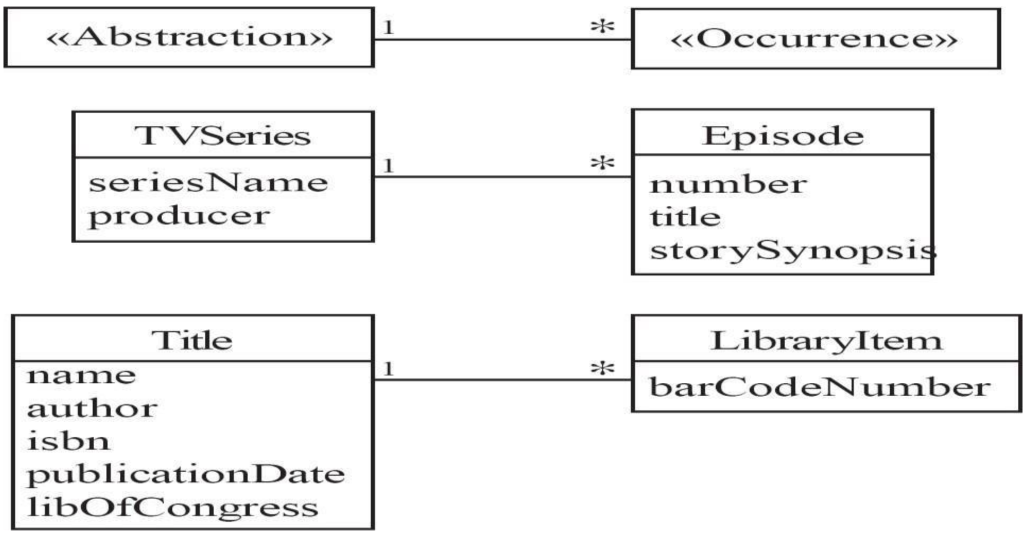
1.What is the best way to represent such sets of occurrences in a classdiagram?

- **Forces:**

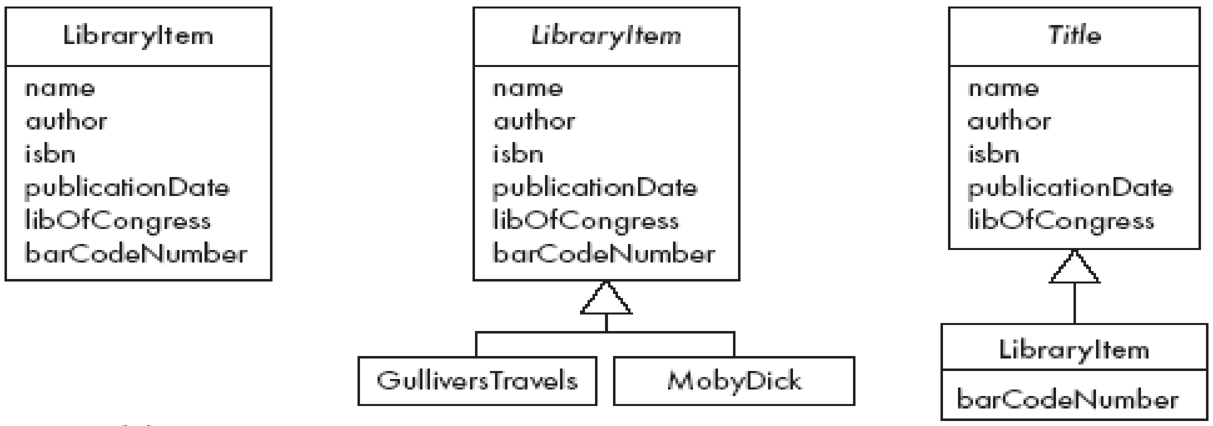
1. You want to represent the members of each set of occurrenceswithout duplicating the common information

- **Solution**:

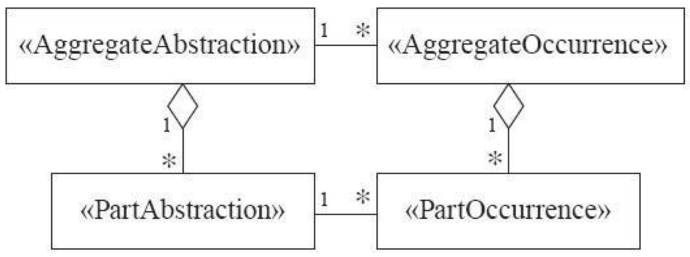
Create an <Abstraction> class that contains the data that iscommon to all the members of a set of occurrences. Then create an <<occurrence>> class representing the occurrences of this abstraction.Connect these classes with a one-to-many association.

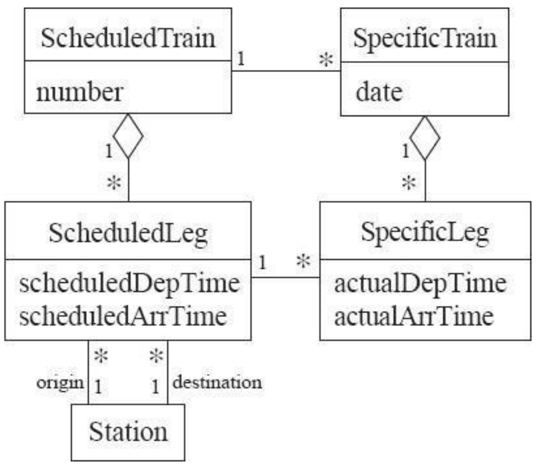


Antipatterns: Solutions that are inferior or do not work in this context.



Square variant or related Patterns: patterns that are similar to this pattern**.**





Refereneces: This pattern is a generalization of the Title-Item pattern of Eriksson andPenker.

2. THE GENERAL HIERARCHY PATTERN

- **Context:**

1.Objects in a hierarchy can have one or more objects above them(superiors), and one or more objects below them (subordinates).

2. Some objects cannot have any subordinates.

- **Problem:**

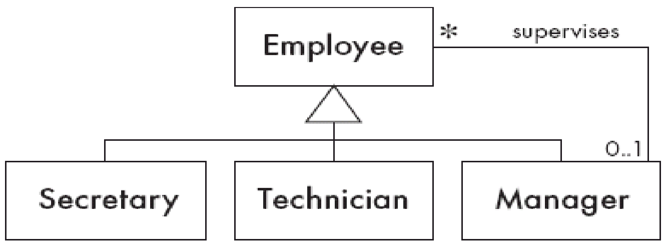
1.How do you represent a hierarchy of objects, in which some objectscannot have subordinates?

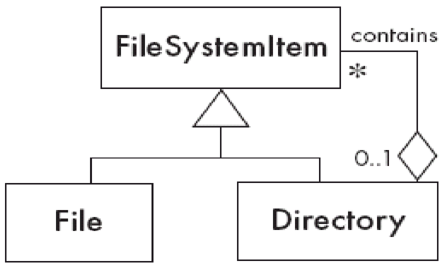
- **Forces:**

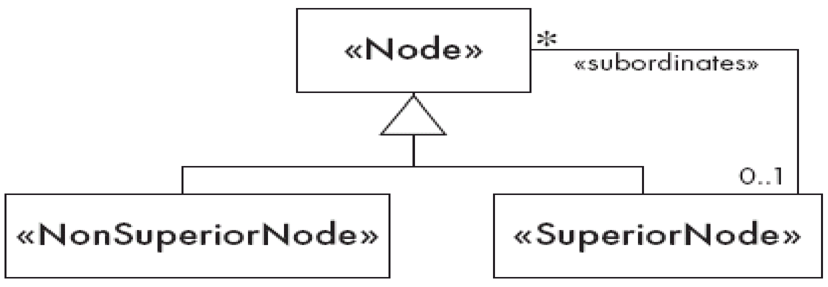
1.You want a flexible way of representing the hierarchy that prevents certain objects from having subordinates.

2.All the objects have many common properties and operations.

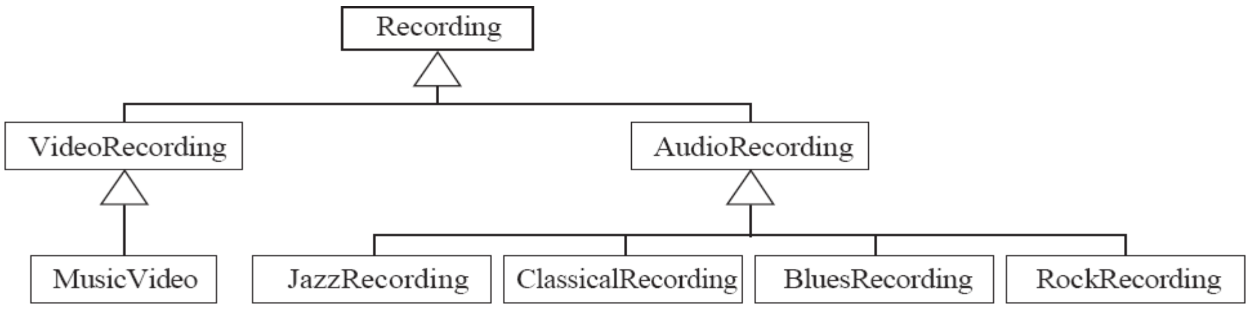
- **Solution:**







- **Antipattern:**



References: The composite pattern is one of the 'Gang of Four ' Patterns.

3. THE PLAYER-ROLE PATTERN

- ***Context***:

1.A *role* is a particular set of properties associated with an object in aparticular context.

2. An object may *play* different roles in different contexts.

- ***Problem***:

1.How do you best model players and roles so that a player canchange roles or possess multiple roles?

- ***Forces***:

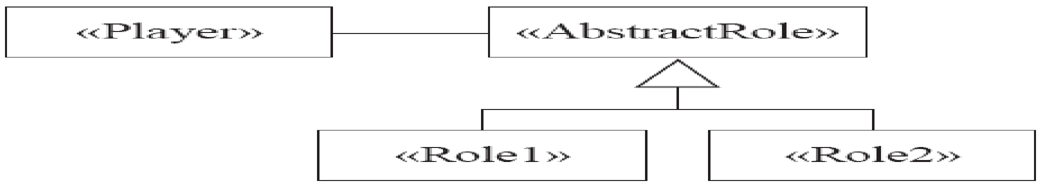
1.It is desirable to improve encapsulation by capturing the informationassociated with each separate role in a class.

2.You want to avoid multiple inheritance.

3. You cannot allow an instance to change class.

- ***Solution****:*

Example:



- **Antipatterns:**
- Merge all the properties and behaviours into a single «Player» class and nothave «Role» classes at all.
- Create roles as subclasses of the «Player» class.

4. THE SINGLETON PATTERN

- ***Context***:

— It is very common to find classes for which only one instance shouldexist (*singleton)*

- ***Problem***:

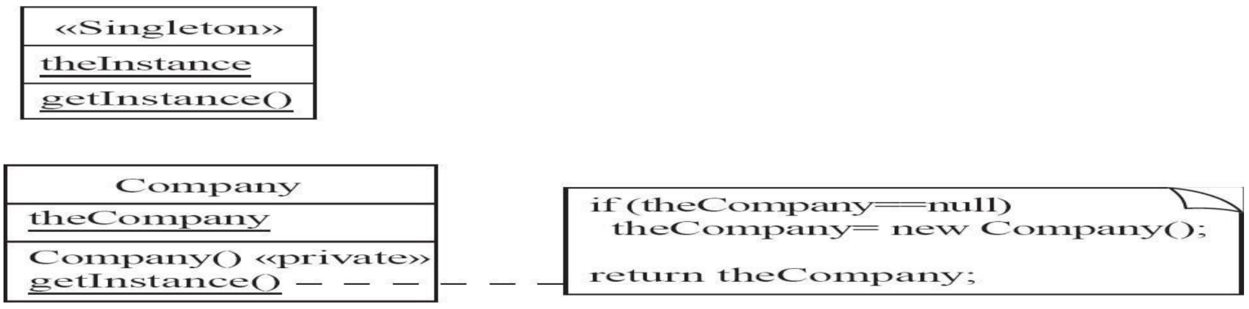
— How do you ensure that it is never possible to create more than oneinstance of a singleton class?

- ***Forces***:

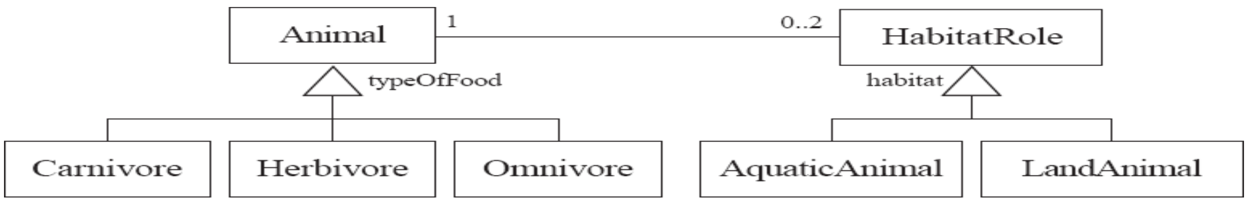
— The use of a public constructor cannot guarantee that no more thanone instance will be created.

— The singleton instance must also be accessible to all classes thatrequire it

- **Solution:**



5. THE OBSERVER PATTERN



- ***Context***:

1.When an association is created between two classes, the code for theclasses becomes inseparable.

2. If you want to reuse one class, then you also have to reuse the other.

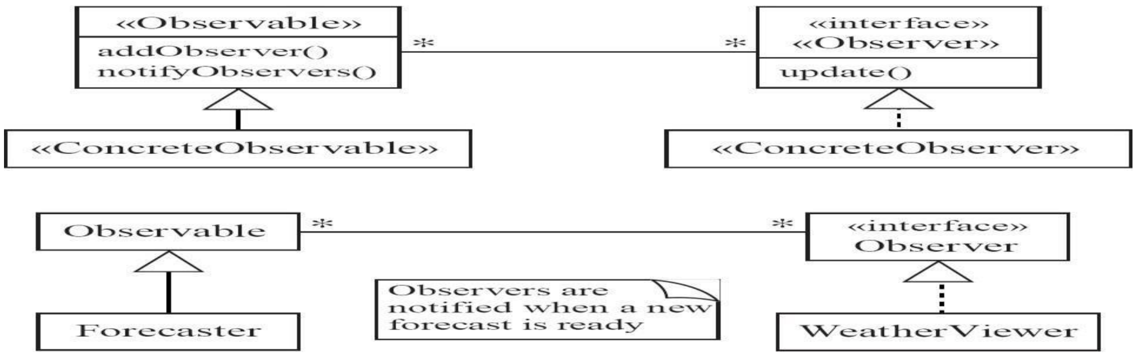
- ***Problem***:

1. How do you reduce the interconnection between classes, especiallybetween classes that belong to different modules or subsystems?

- ***Forces***:

1.You want to maximize the flexibility of the system to thegreatestextent possible

- ***Solution:***



Antipatterns:

- Connect an observer directly to an observable so that they both havereferences to each other.
- Make the observers *subclasses* of the observable.

6. THE DELEGATION PATTERN

- ***Context***:

1. You are designing a method in a class

2. You realize that another class has a method which provides therequired service

3.Inheritance is not appropriate

4.E.g. because the isa rule does not apply

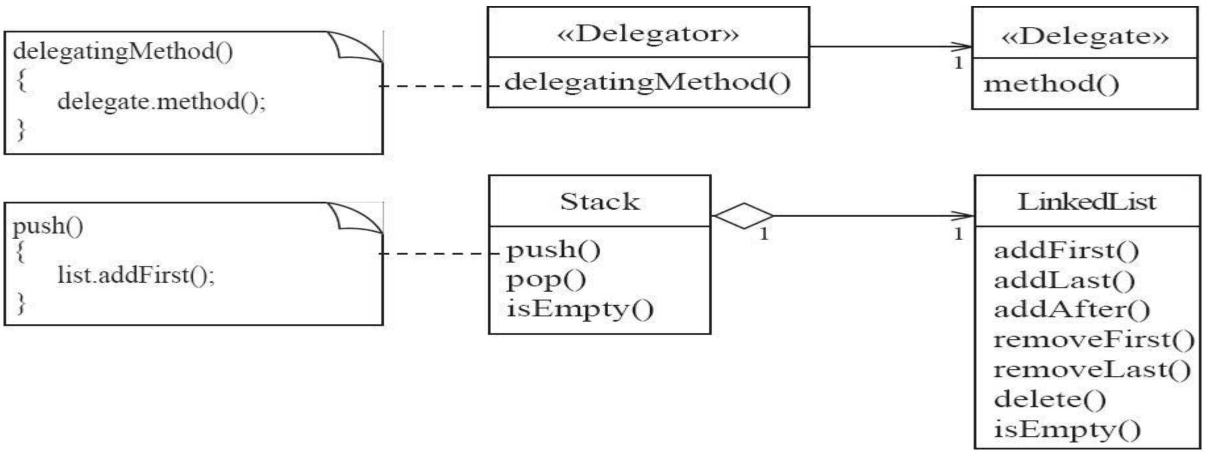
- ***Problem***:

1.How can you most effectively make use of a method that alreadyexists in the other class?

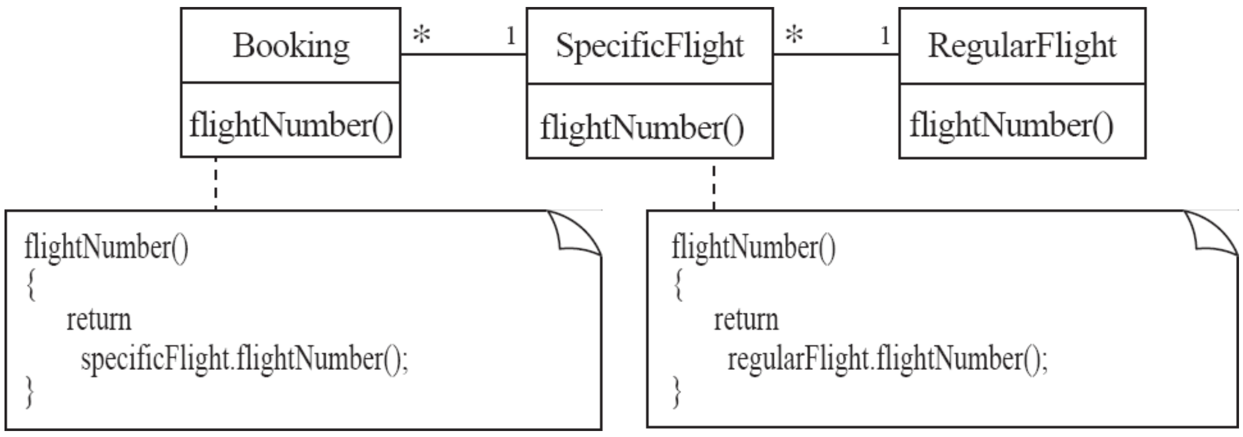
- ***Forces***:

1.You want to minimize development cost by reusing methods

- ***Solution:***



Example:



7. THE ADAPTER PATTERN

- ***Context***:

1. You are building an inheritance hierarchy and want to incorporate itinto an existing class.

2.The reused class is also often already part of its own inheritancehierarchy.

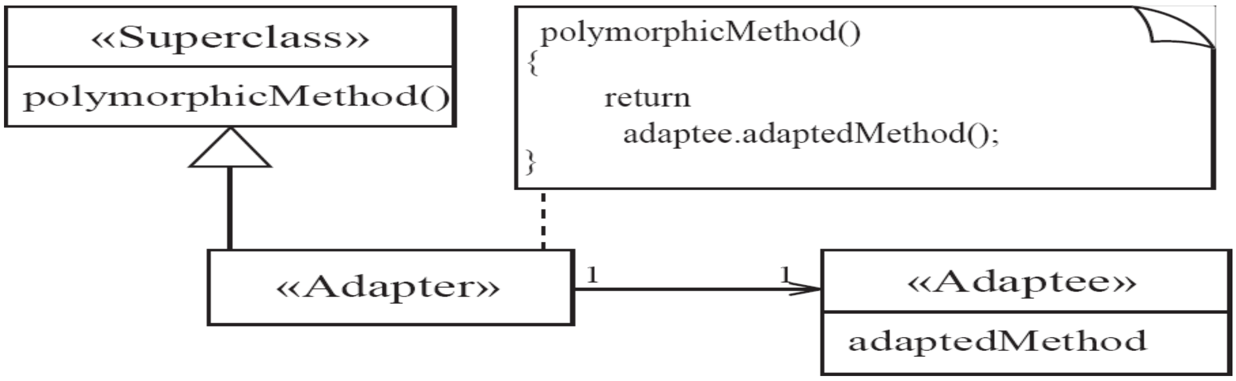
- ***Problem***:

1.How to obtain the power of polymorphism when reusing a classwhose methods have the same function but *not* the same signature as the other methods in the hierarchy?

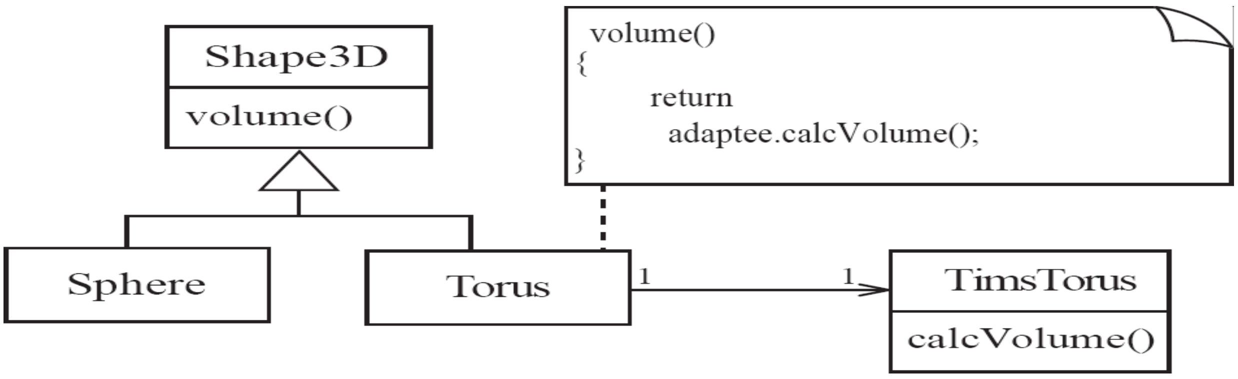
- ***Forces***:

1.You do not have access to multiple inheritance or you do not wantto use it.

- ***Solution:***



Example:



8. <u>THE FAÇADE PATTERN</u>

- ***Context***:

1. Often, an application contains several complex packages.

2.A programmer working with such packages has to manipulate manydifferent classes

- ***Problem***:

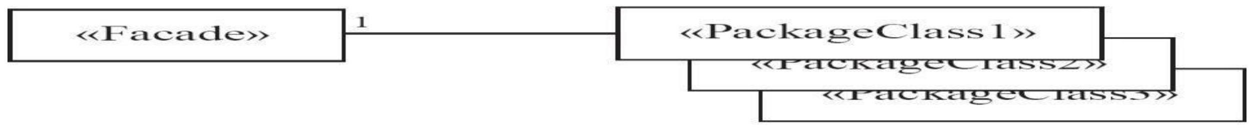
1.How do you simplify the view that programmers have of a complexpackage?

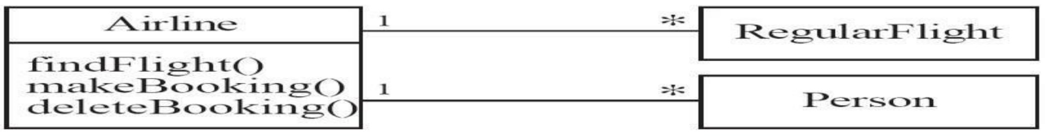
- ***Forces***:

1.It is hard for a programmer to understand and use an entiresubsystem

2.If several different application classes call methods of the complexpackage, then any modifications made to the package will necessitate a complete review of all these classes.

• ***Solution:***





9. THE IMMUTABLE PATTERN

• ***Context***:

1.An immutable object is an object that has a state that never changesafter creation

• ***Problem***:

1.How do you create a class whose instances are immutable?

• ***Forces***:

1.There must be no loopholes that would allow 'illegal' modificationof an immutable object

• ***Solution***:

1.Ensure that the constructor of the immutable class is the *only* placewhere the values of instance variables are set or modified.

2.Instance methods which access properties must not have sideeffects.

3.If a method that would otherwise modify an instance variable isrequired, then it has to return a *new* instance of the class.

10. THE READ-ONLY INTERFACE PATTERN

• ***Context***:

1.You sometimes want certain privileged classes to be able to modifyattributes of objects that are otherwise immutable

• ***Problem***:

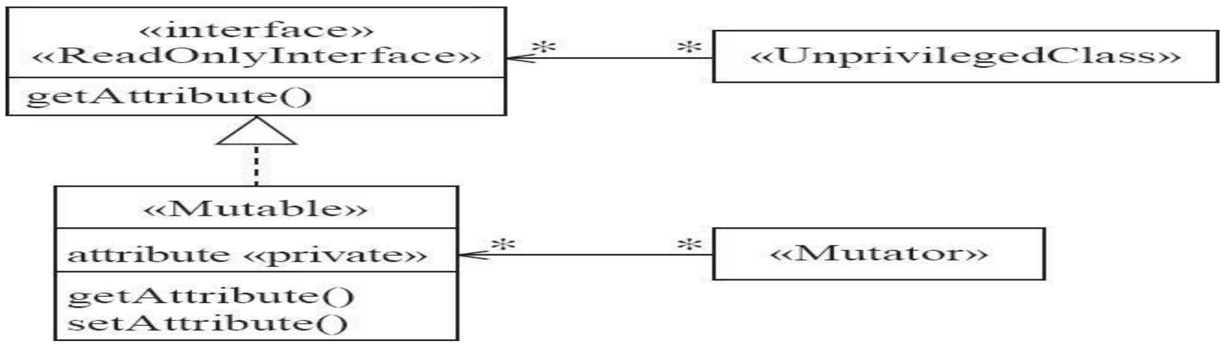
1.How do you create a situation where some classes see a class asread-only whereas others are able to make modifications?

• ***Forces:***

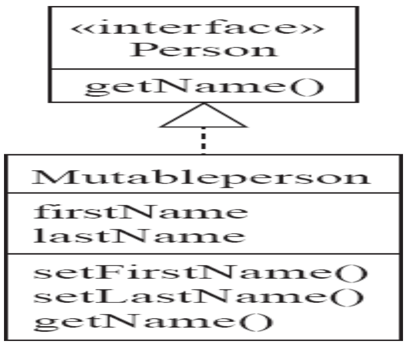
1.Restricting access by using the **public**, **protected** and **private** keywords is not adequately selective.

2.Making access **public** makes it public for both reading and writing

Solution:



Example:



<u>11.THE PROXY PATTERN</u>

Context:

- Often, it is time-consuming and complicated to create instances of aclass (*heavyweight* classes).
- There is a time delay and a complex mechanism involved in creatingthe object in memory

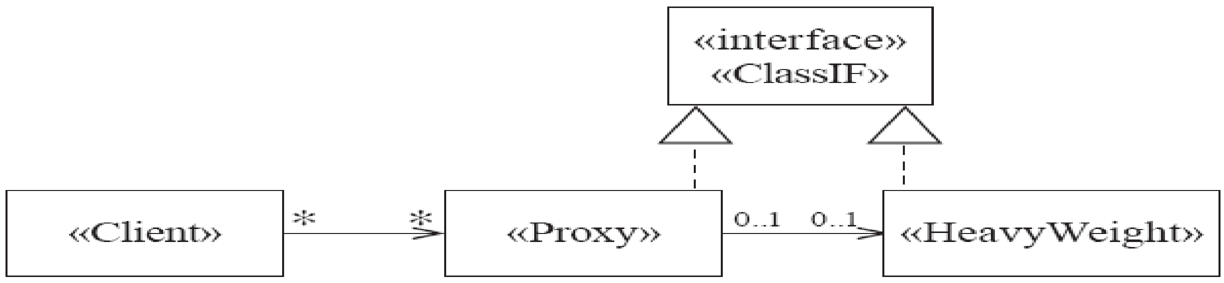
Problem:

- How to reduce the need to create instances of a heavyweight class?

Forces:

- We want all the objects in a domain model to be available forprograms to use when they execute a system’s various responsibilities.
- It is also important for many objects to persist from run to run of thesame program

Solution:



12.The Factory Pattern:

- ***Context***:

1. A reusable framework needs to create objecs; however the classofthe created objects depends on the application.

- ***Problem***:

1.How do you enable a programmer to add new application-specificclass into a system built on such a framework?

- ***Forces***:

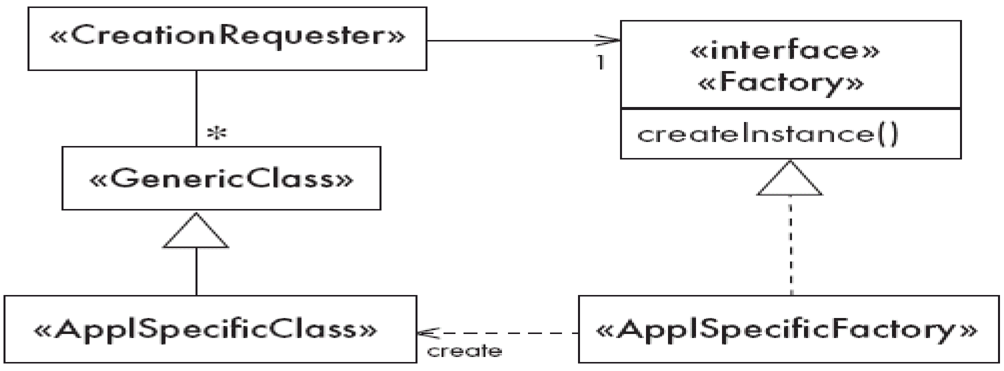
1.We want to have the framework create and work with application-specific classes that the framework does not yet know about.

- ***Solution***:

1.The framework delegates the creation of application-specific classesto a specialized class, the Factory.

2.The Factory is a generic interface defined in the framework.

3.The factory interface declares a method whose purpose is to createsome subclass of a generic class.



UNIT IV

Basic Definitions:

- A **Failure** is an unacceptable behaviour exhibited by a system.
- A **defect** is a flaw in any aspect of the system including the requirements, the design and the code, that contributes, or may potentially contribute, to the occurrence of one or more failures. A defect is also known as a fault.
- An **error** is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect into the system.

<u>EFFECTIVE AND EFFICIENT TESTING</u>:

- Software testing can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by it's design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.
- The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy and usability. It mainly aims at measuring specification, functionality and performance of a software program or application
- To test effectively, you must use a strategy that uncovers as many defects possible.
- To test efficiently, you must find the largest possible number of defects using the fewest possible tests. An effective and efficient testing strategy is often called a high-yield strategy.

Software testing can be divided into two steps:

1. **Verification:** it refers to the set of tasks that ensure that software correctlyimplements a specific function.
2. **Validation:** it refers to a different set of tasks that ensure that the software thathas been built is traceable to customer requirements.

Different techniques of Software Testing:

Black-Box Testing

- The technique of testing without having any knowledge of the interior workings of the application is called black-box testing. The tester is oblivious to the system architecture and does not have access to the source code.
- Typically, while performing a black-box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon.The following table lists the advantages and disadvantages of black-box testing.

Advantages	**Disadvantages**
Well suited and efficient for large code segments.	Limited coverage, since only a selected number of test scenarios is actually performed.
Code access is not required.	Inefficient testing, due to the fact that the tester only has limited knowledge about an application.
Clearly separates user's perspective from thedeveloper's perspective through visibly defined roles.	Blind coverage, since the tester cannot target specific code segments or error prone areas.
Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language, or operating systems.	The test cases are difficult to design.

White-Box Testing

- White-box testing is the detailed investigation of internal logic and structure of the code. White-box testing is also called **glass testing** or **open-box testing**. In order to perform **white-box** testing on an application, a tester needs to know the internal workings of the code.
- The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

The following table lists the advantages and disadvantages of white-box testing.

Advantages	**Disadvantages**
As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively.	Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased.

It helps in optimizing the code.	Sometimes it is impossible to look into everynook and corner to find out hidden errors that
	may create problems, as many paths will gountested.
Extra lines of code can be removed which canbring in hidden defects.	It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.
Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.	

<u>DEFECTS IN ORDINARY ALGORITHMS:</u>

1. **Incorrect logical conditions:**

   Defect: The logical conditions that govern looping and if-then-else statements arewrongly formulated.

   Testing Strategy:

- Use equivalence class and boundary testing.
- Consider as an input each variable used in a rule or logicalcondition.

2. **Performing a calculation in the wrong part of a control construct**:

   Defect: The program performs an action when it should not, or does not perform anaction when it should. These are typically caused by inappropriately excluding the action from or including the action in, a loop or if-then-else construct.

   Testing Strategy:

- Design tests that execute each loop zero times, exactly once,and more than once.

3. **Non Terminating a loop or recursion:**

   Defect: A loop or a recursion does not always terminate, that is ,it is ‘infinite’.

   Testing Strategy:

- Although the programmer should have analyzed all loops orrecursions to ensure they reach a terminating case.

4. **Not setting up the correct preconditions for an algorithm:**

   Defect: When specifying an algorithm, one specifies preconditions that state whatmust be true before the algorithm should be executed. A defect would exist if a program proceeds to do its work, even when the preconditions are not satisfied.

Testing Strategy: Run test cases in which each precondition is not satisfied.Preferably its input values are just beyond what the algorithm can accept.

5. **Not handling null conditions**:

   Defect: A null condition is a situation where there normally exists one or more dataitems to process, but sometimes there are none. It is a defect when a program behaves abnormally when a null condition is encountered. In these situations, the program should ‘do nothing, gracefully’.

   Testing Strategy: Determine all possible null conditions and run test cases thatwould highlight any in appropriate behaviour.

6. **Not handling singleton or n0on-singleton conditions:**

   Defect: A singleton condition is like a null condition. It occurs when there is normally more than one of something, but sometimes there is only one. A non- singleton condition is the inverse there is almost always one of something, but occasionally there can be more than one. Defects occur when the unusual case isnot properly handled.

   Testing Strategy: Brainstorm to determine unusual conditions and run appropriatetests.

7. **Off-by-one errors**:

   Defect: A program inappropriately adds or subtracts 1, or inappropriately loops onetoo many times or one too few times. This is particularly common type of defect.

   Testing Strategy: Develop boundary tests in which you verify that the programcomputes the correct numerical answer, or performs the correct number of iterations.

8. **Operator precedence errors:**

   Defect: An operator precedence error occurs when a programmer omits needed parentheses, or puts parentheses in the wrong place. Operator precedence errors areoften extremely obvious, but can occasionally lie hidden until special conditions arise.

   Testing Strategy: In software that computes formulae, run tests that anticipatedefects.

DEFECTS IN NUMERICAL ALGORITHMS

Assuming a floating point value will be exactly equal to some other value

Defect: If you perform an arithmetic calculation on a floating point value, then the result will very rarely be computed exactly. To test equality, you should always test if it is within a small range around that value.

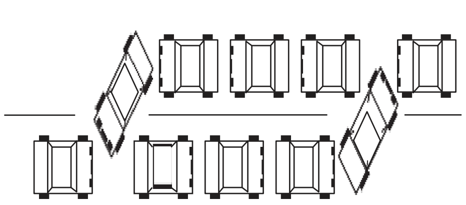
Testing strategies: Standard boundary testing should detect this type of defect.

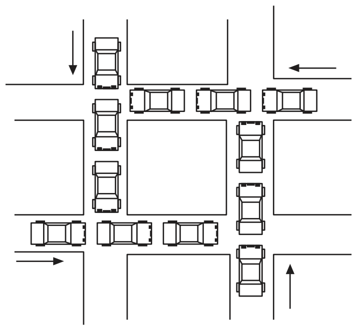
DEFECTS IN TIMING AND COORDINATION

Timing and co-ordination defects arise in situations involving some form of concurrency. They occur when several threads or processes interact in inappropriate ways. The three most important kinds of timing and co-ordination defects are deadlocks, livelocks and critical races.

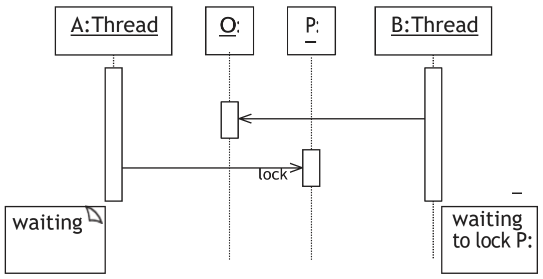
Deadlock and livelock

Defects: A deadlock is a situation where two or more threads or processes are stopped, waiting for each other to do something before either can proceed. Since neither can do anything, they permanently stop each other from proceeding. Vehicles waiting to move in one direction hold up vehicles waiting to travel in other directions; however, each set of vehicles is ultimately delaying itself. Everybody therefore waits indefinitely.



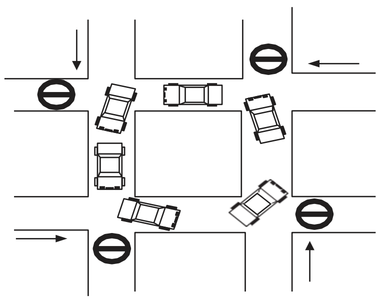


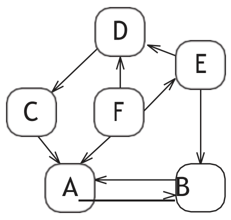
Twoexamples of gridlock: a form of deadlock



A deadlock situation in software

Livelock is similar to deadlock, in the sense that the system is stuck in a particular behavior that It cannot getout of.The difference is as follows : where as in deadlock the system is normally hung, with nothing going on, in livelock, the system can do some computations, but it can never get out of a limited set of states.





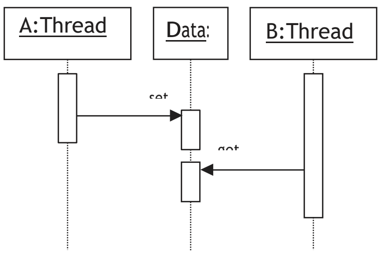
Livelock in traffic (left) andsoftware (right)

Testing Strategies: Deadlocks and livelocks tend to occur as a result of unusual combinations of conditions that are hard to anticipate or reproduce. It is often most effective to use *inspection* to detect such defects, rather than testing alone. Whether testing or inspecting, a person with a background in real-time systems should be employed – such a person can apply his or her knowledge and experience to best anticipatetiming and co-ordination defects.

If its cost is justifiable, glass-box testing is one of the best ways to uncover thesedefects, since you can actually study the progress of the various threads.

Critical races

Defect: A critical race is a defect in which one thread or process can sometimes experience a failure because another thread or process interferes with the 'normal' sequence of events. The defect is not that the other thread tries to do something, but that the system allows interference to occur. Critical races are often simply called 'race conditions', although the word 'critical' should be used to distinguish a defect from a race that has no bad consequences.



Testing Strategies: Testing for critical races is done using the same strategies as testing for deadlocks and livelocks. Once again, inspection is often a better solution.

It is particularlyhardtotestforcriticalracesusingblack-boxtestingalone, since you often do not know the extent of the concurrency going on inside the system, and you cannot always manipulate the various threads to cause race conditions.

<u>DEFECTS IN HANDLING STRESS AND UNUSAL SITUATIONS</u>

The defects discussed in this section are encountered only when a system is being heavily used, or forced to its limits in some other way. These defects represent a lack of robustness. To test for such defects you must run the system intensively, supply it with a very large amount of input, run many copies of it, run it on a computer or network that is busy running other systems, and run it in atypical environments.

- **Insufficient throughput or response time on minimal configurations**

Defect: A minimally configured platform is one that barely conforms to the environment specified in the requirements. It has exactly the minimum amount of memory and disk space specified, the slowest CPU, the oldest permitted release of the operating system and other support software, as well as the slowest allowed network connection.

It is a defect if, when the system is tested on a minimal configuration, its throughput or response time fails to meet requirements.

Testing Stragies: Perform testing using minimally configured platforms. For extra reliability, you could test the system in an environment that has a configuration that is worse than the minimum. In such conditions, it should report that it could not run properly.

- **Incompatibility with specific configurations of hardware or software**

  Defect: It is extremely common for a system to work properly with certain configurations of hardware, operating systems and external libraries, but fail if it is runusing other configurations.

  Testing Strategies: Extensively execute the system with a wide variety of configurations that might been countered by users. By a 'configuration' we mean a particular combination of hardware, operating system and other software.

- **Defects in handling peak loads or missing resources**

  Defect: If a computer becomes short of resources such as memory, disk space or network bandwidth, then programs cannot be expected to continue running normally. However, it is a defect if the system does not gracefully handle such situations. Other types of shortage of resources include missing files or data, lack of permission to perform certain operations, inability to start new threads, or running out of space in a fixed size datastructure.

  No matter what the cause, the program being tested should report the problem in such away that the user will understand. It should then either wait for the problem to be resolved, terminate gracefully or deal with the situation in some other sensible way.

  Testing Strategy: In any of these cases, the tester has to deny the program under test the resources it normally uses, by employing such methods as deleting files, concurrently running other resource-hungry programs, making less memory available, or disconnecting the system from the network. The tester can also run a very large number of copies of the program being tested, all at the same time. To do this effectively, you have to write a special program called a *load generator* that constantly provides input tothe system. Load generators are available commercially.

- **Inappropriate management of resources**

  Defect: A program does not manage resources effectively if it uses certain resources but does not make them available to other programs when it no longer needs them.

  Testing Strategy: Run the program intensively over a long period of time in such a way that it usesmany resources, relinquishes them and then uses them again repeatedly.

- **Defects in the process of recovering from a crash**

  Defect: Any system will undergo a sudden failure if its hardware fails, or if its power is turned off. When this occurs, It is a defect if the system is left in an unstable state and hence is unable to recover fully once the power is restored or the hardware repaired. It is also a defect if a system does not correctly deal with the crashes of related systems.

  Testing Strategy: An approach for testing for defects in the recovery process is to kill either your program or the programs with which it communicates, at various times during execution. You could also try turning the power off; however, operating systems themselves are often intolerant of doing that.

STRATEGIES FOR TESTING LARGE SYSTEMS

- **Integration testing: big bang versus incremental approaches**
- Testing how the parts of a system or subsystem work together is commonly called integration testing. It can be contrasted with unit testing, which is testing an individual module or component in isolation.
- The simplest approach to integration testing is big bang testing. In this approach, you take the entire integrated system and test it all at once. This strategy can be satisfactory when testing a small system; however, when a failure occurs while testing a larger system, it may be hard to tell in which subsystem a defect lies. A better integration testing strategy in most cases is incremental testing.
- In this approach, you first test each individual subsystem in isolation, and then continue testing as you integrate more and more sub systems. The big advantage of incremental testing is that when you encounter a failure, you can findthe defect more easily.
- Incremental testing can be performed horizontally or vertically, depending on the architecture of the system. Horizontal testing can be used when the system is divided into separate sub-applications, such as 'adding new products' and 'selling products'; you simply test each sub-application in isolation. However, for any sub- application that is complex, it is necessary to divide it up vertically into layers.

There are several strategies for vertical incremental testing: top-down, bottom-up and sandwich.

Top-down testing

- In top-down testing, you start by testing only the user interface, with the underlying functionality simulated by stubs. Stubs are pieces of code that have the same interface(i.e. API) as the lower-level layers, but which do not perform any real computations or manipulate any real data. Any call to a stub will typically immediately return with a fixed default value.
- If a defect is detected while performing top-down testing, the tester can be reasonably confident that the defect is in the layer that calls the stubs. It could also be in the stubs, but that is less likely since stubs are so simple. The big drawback to top-down testing is the cost of writing the stubs.

Bottom-up testing

- To perform bottom-up testing, you start by testing the very lowest levels of the software. This might include a database layer, a network layer, a layer that performs some algorithmic computation, or a set of utilities of some kind.
- You need drivers to test the lower layers of software. Drivers are simple programs designed

specifically for testing; they make calls to the lower layers. Driversinbottom-uptestinghaveasimilarroletostubsintop-downtesting, and are time-consuming to write.

Sandwich testing

- Sandwich testing is a hybrid between bottom-up and top-down testing – it is sometimesthereforecalledmixedtesting.Atypicalapproachtosandwichtestingis to test the user interface in isolation,using stubs, and also to testthe very lowest- levelfunctions, using drivers.
- Then, when the complete system is integrated, only the middle layer remains on which to perform the final set of tests. For many systems, sandwich testing will be the most effective form of integration testing.

The test–fix–test cycle and regression testing

- When a failure occurs during testing or after deployment, most organizations follow a carefully planned process. Each failure report is entered into a failure tracking system. It is then is screened and assigned a priority. It is often too expensive to fix every defect, therefore low-priority failures might be put on a*known bugs list* that is included with the software's *release notes*.
- When a decision is made to resolve a failure, somebody is assigned to investigate it, track down the defect or defects that caused it, and fix those defects. Finally a new version of the system is created, ready to be tested again.
- **THE RIPPLE EFFORT**: Unfortunately, there is a high probability that efforts to remove defects will actually add new ones – either because the maintainer tries to fix problems without fully understanding the ramifications of the changes, or because he or she makes ordinary human errors. This phenomenon is known as the *ripple effect*, because new defects caused by erroneous fixes of other defects tend to spread through a system like a ripple. The system regresses into a more and more failure-prone state.
- You can minimize the ripple effect by applying a very disciplined approach to the process of fixing defects.
- However, experience shows that the ripple effect will still persist if you do not have an appropriate testing strategy.
- After making any change, you must therefore not only re-run the test case that led to the detection of the defect, but you must also re-test the rest of the system's functionality. This latter process is called **regression testing.**
- It is normally far too expensive to re-run every single test case whenever a change is made to software, so regression testing involves running a subset of the original test cases. The regression tests are carefully selected to cover as muchof the system as possible.

Deciding when to stop testing

- You might imagine that you should go on re-testing software until all the test cases have passed.

Unfortunately, this is not a practical approach.

- However, it is also a poor strategy to stop testing merely because you have run out of time or money. This will result in a poor-quality system. You should, instead, establish a set of criteria like the following to decide when testing should beconsidered complete:
- All of the level 1 test cases must have been successfully executed.
- Certain predefined percentages of level 2 and level 3 test cases must have been executed successfully.
- The targets must have been achieved and then maintained for at least two cycles of ‘builds’. A *build* involves compiling and integrating all the components of the software, incorporating any changes since the last build.

The roles of people involved in testing

- Testing is an intellectual exercise that is just as challenging and creative as design. All software engineers should develop their testing skills, although some have a particular talent for testing and may specialize in it.
- In a software development organization, the first pass of unit and integration testing is often called *developer testing*. This is preliminary testing performed by the software developers who do the design and programming. Organizations should then employ an *independent testing group* that runs the complete set of test cases, seeking defects left by the developers.

Testing performed by users and clients: alpha, beta and acceptance

- **Alpha testing** is testing performed by users and clients, under the supervision of the software development team. The development team normally invites some users to work with the software and to watch for problems that they encounter.
- **Beta testing** is testing performed by the user or client in their normal work environment. It ncan be initiated either at the request of the software developers, or at the request of users who want to try the system out. Beta testers are selected from the potential user population and given a pre-release version of the software.
- **Acceptance testing,** like alpha and beta testing, is performed by users and customers. However, the customers do it on their own initiative – to decide whether software is of sufficient quality to purchase. Many large organizations also perform acceptance testing before they will pay a developer they have contracted to develop custom software for them. Other organizations use acceptance testing inorder to choose between several competing generic products.

Software Project Management

What is Project?

A project is a group of tasks that need to complete to reach a clear result. A project also defines as a set of inputs and outputs which are required to achieve a goal. Projects can vary from simple to difficult and can be operated by one person or a hundred.

Projects usually described and approved by a project manager or team executive. Theygo beyond their expectations and objects, and it's up to the team to handle logistics andcomplete the project on time. For good project development, some teams split the project into specific tasks so they can manage responsibility and utilize team strengths.

What is software project management?

Software project management is an art and discipline of planning and supervising software projects. It is a sub-discipline of software project management in which software projects planned, implemented, monitored and controlled.

It is a procedure of managing, allocating and timing resources to develop computer software that fulfills requirements.

In software Project Management, the client and the developers need to know the length, period and cost of the project.

Prerequisite of software project management?

There are three needs for software project management. These are:

1. Time
2. Cost
3. Quality

It is an essential part of the software organization to deliver a quality product, keeping the cost within the client?s budget and deliver the project as per schedule. There are various factors, both external and internal, which may impact this triple factor. Any of three-factor can severely affect the other two.

Project Manager

A project manager is a character who has the overall responsibility for the planning, design, execution, monitoring, controlling and closure of a project. A project manager represents an essential role in the achievement of the projects.

A project manager is a character who is responsible for giving decisions, both large and small projects. The project manager is used to manage the risk and minimize uncertainty. Every decision the project manager makes must directly profit their project.

Role of a Project Manager:

1. Leader: A project manager must lead his team and should provide them direction to make them understand what is expected from all of them.

2. Medium: The Project manager is a medium between his clients and his team. He must coordinate and transfer all the appropriate information from the clients to his team and report to the senior management.

3. **Mentor:**He should be there to guide his team at each step and make sure that the team has an attachment. He provides a recommendation to his team and points them in the right direction.

Responsibilities of a Project Manager:

1. Managing risks and issues.
2. Create the project team and assigns tasks to several team members.
3. Activity planning and sequencing.
4. Monitoring and reporting progress.
5. Modifies the project plan to deal with the situation.

<u>ACTIVITIES OF SOFTWARE PROJECT MANAGEMENT:</u>

Software Project Management consists of many activities, that includes planning of the project, deciding the scope of product, estimation of cost in different terms, scheduling of tasks, etc.

The list of activities are as follows:

1. Project planning and Tracking
2. Project Resource Management
3. Scope Management
4. Estimation Management
5. Project Risk Management
6. Scheduling Management
7. Project Communication Management
8. Configuration Management

Now we will discuss all these activities -

1. Project Planning: It is a set of multiple processes, or we can say that it a task that performed before the construction of the product starts.

2. Scope Management: It describes the scope of the project. Scope management is important because it clearly defines what would do and what would not. Scope Management create the project to contain restricted and quantitative tasks, which may merely be documented and successively avoids price and time overrun.

3. Estimation management: This is not only about cost estimation because whenever we start to develop software, but we also figure out their size(line of code), efforts, time aswell as cost.
If we talk about the size, then Line of code depends upon user or software requirement.
If we talk about effort, we should know about the size of the software, because based onthe size we can quickly estimate how big team required to produce the software.
If we talk about time, when size and efforts are estimated, the time required to develop the software can easily determine.

And if we talk about cost, it includes all the elements such as:

- Size of software
- Quality
- Hardware
- Communication
- Training
- Additional Software and tools
- Skilled manpower

4. Scheduling Management: Scheduling Management in software refers to all the activities to complete in the specified order and within time slotted to each activity. Project managers define multiple tasks and arrange them keeping various factors in mind.

For scheduling, it is compulsory -

- Find out multiple tasks and correlate them.
- Divide time into units.
- Assign the respective number of work-units for every job.
- Calculate the total time from start to finish.
- Break down the project into modules.

5. Project Resource Management: In software Development, all the elements are referred to as resources for the project. It can be a human resource, productive tools, and libraries.

Resource management includes:

- Create a project team and assign responsibilities to every team member
- Developing a resource plan is derived from the project plan.
- Adjustment of resources.

6. Project Risk Management: Risk management consists of all the activities like identification, analyzing and preparing the plan for predictable and unpredictable risk in the project.

Several points show the risks in the project:

- The Experienced team leaves the project, and the new team joins it.
- Changes in requirement.
- Change in technologies and the environment.
- Market competition.

7. Project Communication Management: Communication is an essential factor in the success of the project. It is a bridge between client, organization, team members and as well as other stakeholders of the project such as hardware suppliers.

From the planning to closure, communication plays a vital role. In all the phases, communication must be clear and understood. Miscommunication can create a big blunder in the project.

8. **Project Configuration Management:** Configuration management is about to control the changes in software like requirements, design, and development of the product.

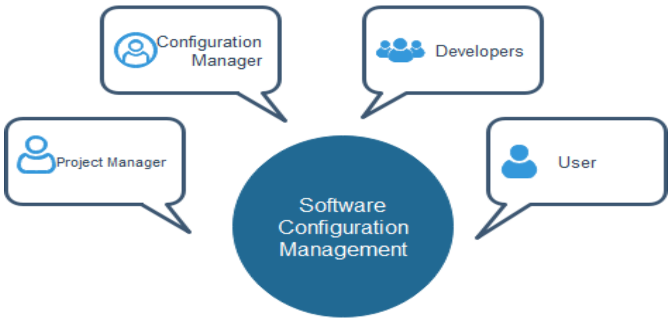
The Primary goal is to increase productivity with fewer errors.

Some reasons show the need for configuration management:

- Several people work on software that is continually update.
- Help to build coordination among suppliers.
- Changes in requirement, budget, schedule need to accommodate.
- Software should run on multiple systems.

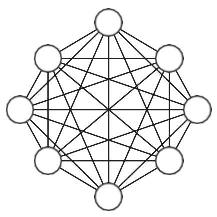
Tasks perform in Configuration management:

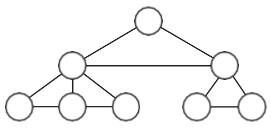
- Identification
- Baseline
- Change Control
- Configuration Status Accounting
- Configuration Audits and Reviews.
- **People involved in Configuration Management:**

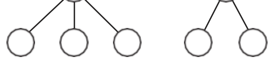


Software Engineering Teams

Software engineering is a human process.Choosing appropriate people for a team, and assigning roles and responsibilities to the team members, is therefore an important project management skill.







(a) Egoless (b)Chiefprogrammer (c) Strict hierachy

- In the **egoless approach**, decisions are made by consensus; this can lead to more creative solutions, since group members spontaneously get together to solve problems when they arise.Ingeneral, the egoless approach is mostsuited to difficult projects with many technical challenges. . However, it is dependent on the personalities of the individuals, and hence is more likely to run into trouble if personal conflicts arise. Also, the assumption that everybody is equal is often wrong when a team is composed of people whose levels of skill and knowledgevary widely.
- The **hierarchical approach** is reminiscent of the military, or large bureaucratic organizations. It is suitable for large projects with a strict schedule and where everybody is well trained and has a well-defined role. However, since everybody is responsible only for their own work, problems may go unnoticed.
- The **'chief programmer team'** is a model that is midway between egoless and hierarchical. It works very much like the surgical team in an operating room. The chief programmer leads and guides the project; however, he or she consults with, and relies on, individual specialists. As with the egoless approach, everybody's goal is the success of the project.

Choosing an effective size for a team

We have already discussed the fact that you should divide a system into subsystems. Each subsystem is then assigned to a team; the various teams have to co-ordinate their work. For a given estimated development effort in person-months there is an optimal team size. Doubling the size of a team will not have the development time. Making subsystems, and hence teams, too large or too small tends to make the development more complex and lead to designs that have higher coupling. Subsystems and teams should be sized such that the total amount of required knowledge and exchange of information is reduced – each team needs to understand only the overall software architecture, the details of its own subsystem, plus the interfaces to related subsystems.

Skills needed on a team

The following are some of the more common roles found on a development team:

■ **Architect**. This person is responsible for leading the decision making about the architecture, and maintaining the architectural model.

■ **Project manager**. Responsible for doing the project management tasks described in this chapter. Even in an egoless team, somebody has to be the custodian of the cost estimates and the schedule.

■ **Configuration management and build specialist**. This person ensures that, as changes are made, no new problems are introduced. Every one relies on builds as the baselines for quality assurance and subsequent development. This person also makes sure that documentation for each change is properly updated.

- **User interface specialist**. Although everybody should interact with users, this person has the particular responsibility to make sure that usability is kept at the fore frontof the design process.
- **Technology specialist**. Such a person has specialized knowledge and expertise in a technology such as databases, networking, operating systems etc.
- **Hardware and third-party software specialist**. This person makes sure that the development team has appropriate types of hardware and third-party software on which to develop and test the software. This person will install and perform acceptance testing on any reusable components the team plans to use.
- **User documentation specialist**. This person, who should have a technical writing background, ensures that online help and user manuals are well written.
- **Tester**. Even though there should be an independent test group, the developmentgroupmayhaveapersonwhoisresponsibleforthefirststageoftesting.

<u>**SOFTWARE COST ESTIMATION**</u>

For any new software project, it is necessary to know how much it will cost to develop andhow much development time will it take. These estimates are needed before development is initiated, but how is this done? Several estimation procedures have been developed and are having the following attributes in common.

1. Project scope must be established in advanced.
2. Software metrics are used as a support from which evaluation is made.
3. The project is broken into small PCs which are estimated individually.To achieve true cost & schedule estimate, several option arise.
4. Delay estimation
5. Used symbol decomposition techniques to generate project cost and scheduleestimates.
6. Acquire one or more automated estimation tools.

Uses of Cost Estimation

1. During the planning stage, one needs to choose how many engineers arerequiredfor the project and to develop a schedule.
2. In monitoring the project's progress, one needs to access whether the project is progressing according to the procedure and takes corrective action, if necessary.

Principles of effective cost estimation

Cost estimation is no difficult, as witnessed by the large number of projects that are completed behind schedule and over budget, or are not completed at all. There are several key principles that can help you to make better estimates.

Cost Estimation Principle 1: Divide and conquer

To make a better estimate, you should divide the project up into individual subsystems, and then divide each subsystem further into the activities that will be required to develop it. Next, you make a series of detailed estimates for each individual activity, and sum the results to arrive at the grand total estimate for the project.

Cost Estimation Principle 2: Include all activities when making estimates

when asked to estimate the cost of a new feature for Simple Chat, a beginner might focus primarily in the amount of time required to write the requirements document and the code. However, the time required for *all* development activities must be taken into account, including prototyping, design, inspecting, testing, debugging, writing userdocumentation and deployment.

Cost Estimation Principle 3:Base your estimates on past experience combined with what you can observe of the current project

There are two general strategies for using past experience. The first is to base your estimates purely on the person a judgment of experts with in your team. Such people will have worked on other projects and can extrapolate their experience to the current project. The second strategy is to use algorithmic models that have been developed in the software industry as a whole by analyzing a wide range of development projects. They take into account various aspects of a project's size and complexity, and provide formulas tocompute anticipated cost.

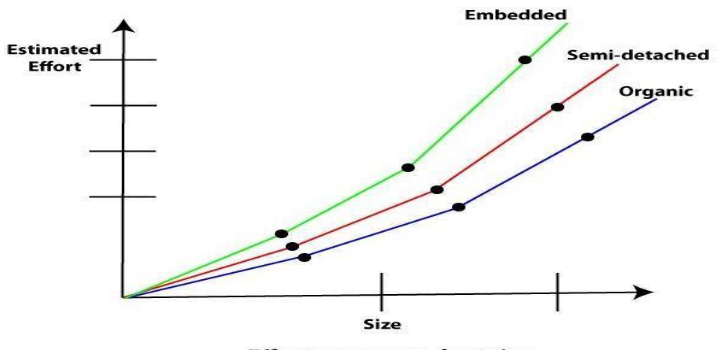
The algorithmic models allow you to systematically base your estimate of development effort on an estimate of some other factor that you can measure, or that is easier to estimate. Project managers base their estimates on factors such as the following:

- The number of use cases.
- The number of distinct requirements.
- The number of classes in the domain model.
- The number of widgets in the prototype user interface.
- An estimate of the number of lines of code.

A typical algorithmic model uses a formula such as the following:

$$E = a + bN^c$$

In this formula, E is the effort estimate and N is the estimate or measure being used as the basis for them effort estimate. The values a, b and $c$are obtained by extensive analysis of past projects, and determinations of the differences that will effect the current project.



Effort versus product size

Figure11.7showstheeffectoftypicalvaluesofthe *c* parameter used by one of the best-known algorithmic cost estimation models, COCOMO II (COnstructive COst Model, version II). COCOMO II computes effort, in person-months, from an estimate of size, measured in lines of code.

Another important algorithmic method is *Function Point Analysis*. In this approach, and several related approaches, you count features of the requirements and use these to compute an estimate of the system's size. You can then apply COCOMO to the sizeestimates.

The basic equation used by approaches like Function Point Analysis is:

$S = W1F1 + W2F2+W3F3 +\ldots$

The F_i represent counts of features of the requirements such as number of inputs, number of tables in the database, number of use cases etc. These counts are multiplied by weights (the W_i values) calculated using industry-wide experience. The results are summed to produce a system size, *S*.

Cost Estimation Principle4: Besure to account for differences when extrapolating from other projects

Although experience from other projects can be a good guide, there are often many subtle differences between projects. You have to consider carefully the effect of differences such as the following when making an estimate for a new project:

■ **Different software developers**. People differ dramatically in their skill and experience levels; as killed programmer can be up to ten times as productive as a less skilled programmer. In projects with only a few people, this can therefore significantlyinfluence the accuracy of estimates.

■ **Different development processes and maturity levels**. Teams that haveskilled management and a mature development methodology that includes such things as quality assurance, risk management and iterative development will be able to work more efficiently than organizations that follow an ad-hoc approach.

Different types of customers and users. Project teams that have good access to and rapport with both their customers and users can often proceed faster than organizations lacking these advantages. In the latter case, delays are caused by sloworpoor decision making. Projects that have a single user or customer will often alsoproceed faster than projects with many users and customers. Not only does it taketime to negotiate decisions when many people are involved, but the resulting compromises maybe difficult to develop efficiently.

■ **Different schedule demands**. Ironically, if a project team is under intense pressure to deliver software by a certain date they may cut corners and make mistakes that actually end up delaying them. On the other hand, a certain amount of deadline pressure can have a positive effect.

■ **Different technology**. The effort required can be affected by changes in hardware, other software systems with which your system must interact, database management systems, operating systems and programming languages. For example, if you change to a new programming language, that language can take considerable time to learn. Also, some languages are known to be less productive than others. Forexample, programming in assembler is unproductive because it takes many statements to express a given computation, and furthermore the complexity of assembler gives rise to more bugs. Programming in C or C++ is likely to result in faster development. Using Java or C# is likely to decrease development time stillfurther.

■ **Different technical complexity of the requirements**. Some systems are intrinsically more complex than others, because they require greater reliability, are distributed, or for many other reasons.

■ **Different domains**. A team embarking on a project in a domain in which it has not worked before will un doubted take more time than a team that has been working in the same domain for years.

■ **Different levels of requirement stability**. Some projects are easier to define in advance. On the other hand, some projects must proceed in an exploratory fashion since the requirements will not be clearly known until prototyping is complete. The total effort will thus vary substantially. Also, some domains are more prone to changes in requirements than others.

Cost Estimation Principle 5: Anticipate the worst case and plan for contingencies

you might have difficulty deciding on the requirements; there might be unexpected technical challenges in the design process; or somebody might quit or notperform up to expectations.

One way to plan for contingencies is to prioritize all the use cases according to their benefit to the customer. Another important thing to do is to build enough 'cushion' into yourcost estimate so as to account for typical delays. One way to do this is as follows. For every detailed estimate, make an 'optimistic' (O) estimate, a 'likely' (L) estimate and a 'pessimistic' (P) estimate. Your O estimate suggests the minimum time you reasonably think the activity might take. Your L estimate accounts for what you think would be a typical number of things going wrong. Your P estimate suggests what you think the activity would consume if many difficulties were experienced. You then add up the O estimates, the L estimates and the P estimates separately to arrive a global estimates of the best-case, typical-case and worst-case cost for the project.

Cost Estimation Principle 6: Combine multiple independent estimates

Use several different techniques and compare the results. If there are discrepancies, you can analyze your calculations to discover what factors are causing the differences. A well-respected approach to making multiple estimates is the Delphi technique. To use this technique, several individuals initially make cost estimates in private.

Principle 7: Revise and refine estimates as work progresses

- As you add detail
- As the requirements change
- As the risk management process uncovers problems

PROJECT SCHEDULING

Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and arrange them keeping various factors in mind. They look for tasks lie in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. Arrangement of tasks which lies out of critical path are less likely toimpact over all schedule of the project.

For scheduling a project, it is necessary to -

- Break down the project tasks into smaller, manageable form
- Find out various tasks and correlate them
- Estimate time frame required for each task
- Divide time into work-units
- Assign adequate number of work-units for each task
- Calculate total time required for the project from start to finish

PROJECT EXECUTION & MONITORING

In this phase, the tasks described in project plans are executed according to their schedules.

Execution needs monitoring in order to check whether everything is going according to the plan. Monitoring is observing to check the probability of risk and taking measures to address the risk or report the status of various tasks.

These measures include -

- **Activity Monitoring -** All activities scheduled within some task can be monitored on day-to-day basis. When all activities in a task are completed, it is considered ascomplete.
- **Status Reports -** The reports contain status of activities and tasks completed within a given time frame, generally a week. Status can be marked as finished,pending or work-in-progress etc.
- **Milestones Checklist -** Every project is divided into multiple phases where majortasks are performed (milestones) based on the phases of SDLC. This milestone checklist is prepared once every few weeks and reports the status of milestones.

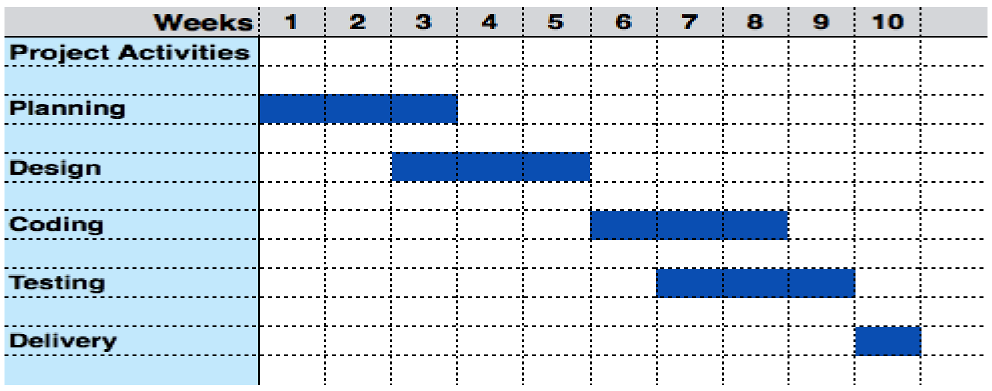
Project Management Tools

The risk and uncertainty rises multifold with respect to the size of the project, even when the project is developed according to set methodologies.

There are tools available, which aid for effective project management. A few are described -

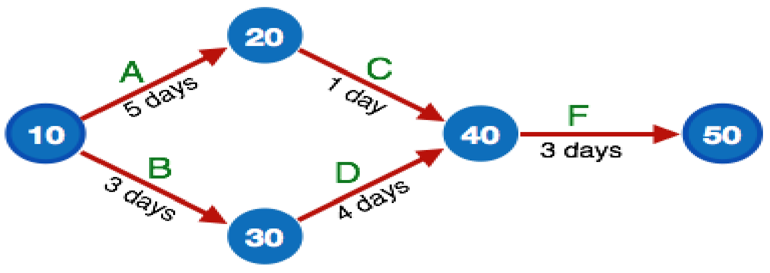
Gantt Chart

Gantt charts was devised by Henry Gantt (1917). It represents project schedule with respect to time periods. It is a horizontal bar chart with bars representing activities and time scheduled for the project activities.



PERT Chart

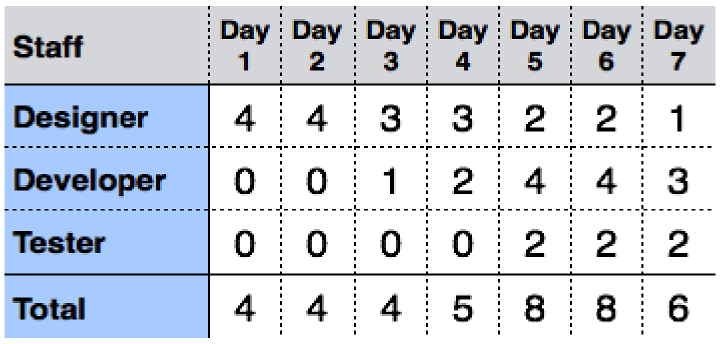
PERT (Program Evaluation & Review Technique) chart is a tool that depicts project as network diagram. It is capable of graphically representing main events of project in both parallel and consecutive way. Events, which occur one after another, show dependency of the later event over the previous one.



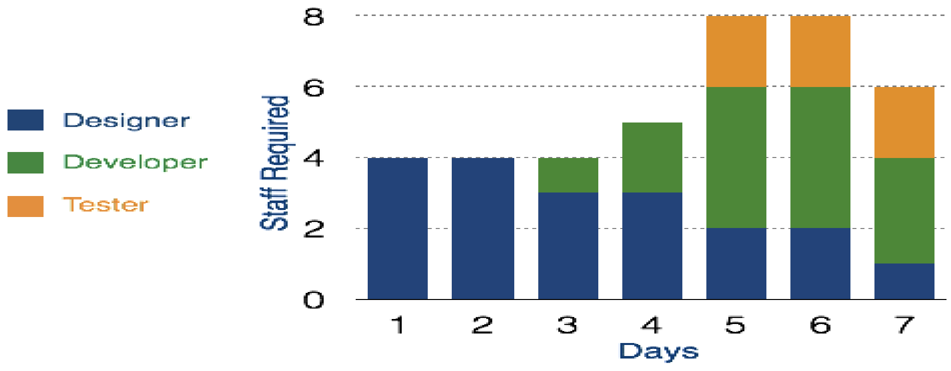
Events are shown as numbered nodes. They are connected by labeled arrows depicting sequence of tasks in the project.

Resource Histogram

This is a graphical tool that contains bar or chart representing number of resources (usually skilled staff) required over time for a project event (or phase). Resource Histogram is an effective tool for staff planning and coordination.



Staff	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Designer	4	4	3	3	2	2	1
Developer	0	0	1	2	4	4	3
Tester	0	0	0	0	2	2	2
Total	4	4	4	5	8	8	6



Critical Path Analysis

This tools is useful in recognizing interdependent tasks in the project. It also helps to find out the shortest path or critical path to complete the project successfully. Like PERT diagram, each event is allotted a specific time frame. This tool shows dependency of event assuming an event can proceed to next only if the previous one is completed.

Software Processes

A software process is the set of activities and associated outcome that produce a software product. Software engineers mostly carry out these activities. These are four key process activities, which are common to all software processes. These activities are:

1. **Software specifications:** The functionality of the software and constraints on itsoperation must be defined.
2. **Software development:** The software to meet the requirement must be produced.
3. **Software validation:** The software must be validated to ensure that it does whatthe customer wants.
4. **Software evolution:** The software must evolve to meet changing client needs.

The Software Process Model

A software process model is a specified definition of a software process, which is presented from a particular perspective. Models, by their nature, are a simplification, so a software process model is an abstraction of the actual process, which is being described. Process models may contain activities, which are part of the software process, software product, and the roles of people involved in software engineering. Some examples of the types of software process models that may be produced are:

1. **A workflow model:** This shows the series of activities in the process along with their inputs, outputs and dependencies. The activities in this model perform human actions.
2. **2. A dataflow or activity model:** This represents the process as a set of activities, each of which carries out some data transformations. It shows how the input to the process, such as a specification is converted to an output such as a design. The activities here may be at a lower

level than activities in a workflow model. They may perform transformations carried out by people or by computers.

3. **3. A role/action model:** This means the roles of the people involved in the software process and the activities for which they are responsible.

There are several various general models or paradigms of software development:

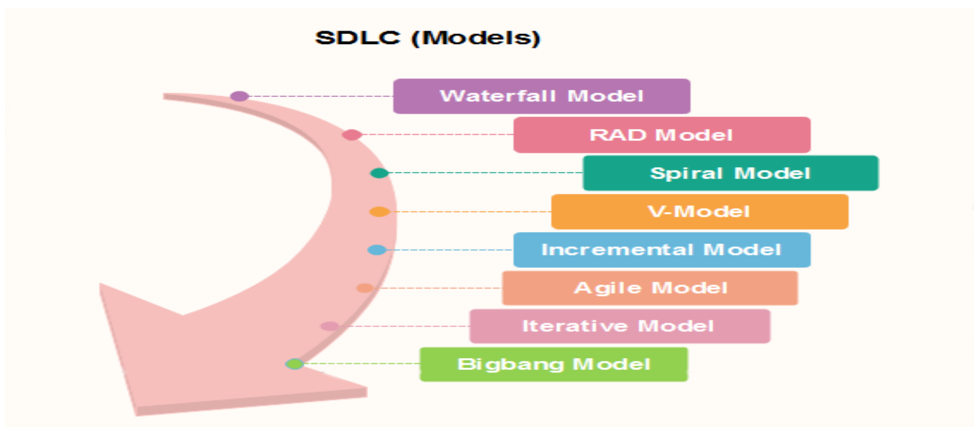
1. **The waterfall approach:** This takes the above activities and produces them as separate process phases such as requirements specification, software design, implementation, testing, and so on. After each stage is defined, it is "signed off" and development goes onto the following stage.
2. **Evolutionary development:** This method interleaves the activities of specification,development, and validation. An initial system is rapidly developed from a very abstract specification.
3. **Formal transformation:** This method is based on producing a formal mathematical system specification and transforming this specification, using mathematical methods to a program. These transformations are 'correctness preserving.' This means that you can be sure that the developed programs meet itsspecification.
4. **System assembly from reusable components:** This method assumes the parts ofthe system already exist. The system development process target on integrating these parts rather than developing them from scratch.

<u>SDLC MODELS</u>

Software Development life cycle (SDLC) is a spiritual model used in project managementthat defines the stages include in an information system development project, from an initial feasibility study to the maintenance of the completed application.

There are different software development life cycle models specify and design, which are followed during the software development phase. These models are also called "**Software Development Process Models**." Each process model follows a series of phase unique to its type to ensure success in the step of software development.

Here, are some important phases of SDLC life cycle:



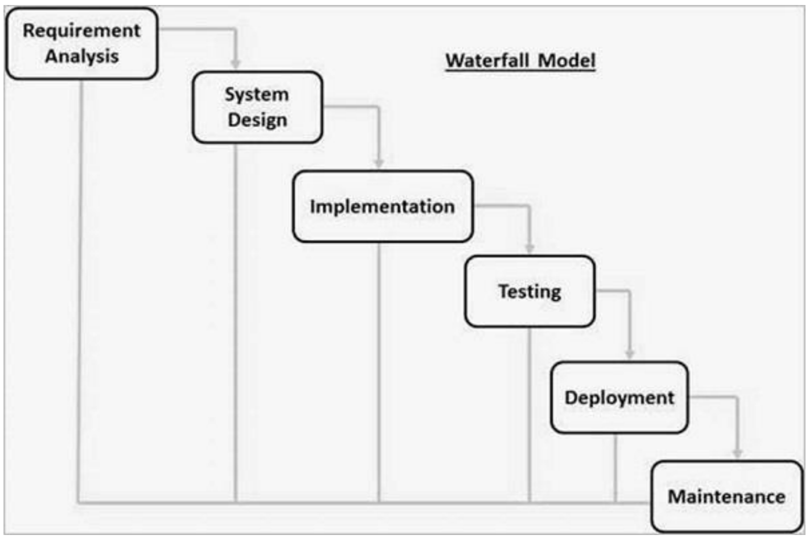
The Waterfall Model was the first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

The Waterfall model is the earliest SDLC approach that was used for software development.

WATERFALL MODEL - Design

Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are −

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.

- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

Waterfall Model - Application

- Requirements are very well documented, clear and fixed.
- Product definition is stable.
- Technology is understood and is not dynamic.
- There are no ambiguous requirements.
- Ample resources with required expertise are available to support the product.
- The project is short.

Some of the major advantages of the Waterfall Model are as follows −

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specificdeliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

The major disadvantages of the Waterfall Model are as follows −

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk ofchanging. So, risk and uncertainty is high with this process model.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- Adjusting scope during the life cycle can end a project.

SPIRAL MODEL

The spiral model, initially proposed by Boehm, is an evolutionary software process model that couples the iterative feature of prototyping with the controlled and systematic aspects of the linear sequential model. It implements the potential for rapid development of new versions of the software. Using the spiral model, the software is developed in a series of incremental releases.

During the early iterations, the additional release may be a paper model or prototype. During later iterations, more and more complete versions of theengineered system are produced.

The Spiral Model is shown in fig:

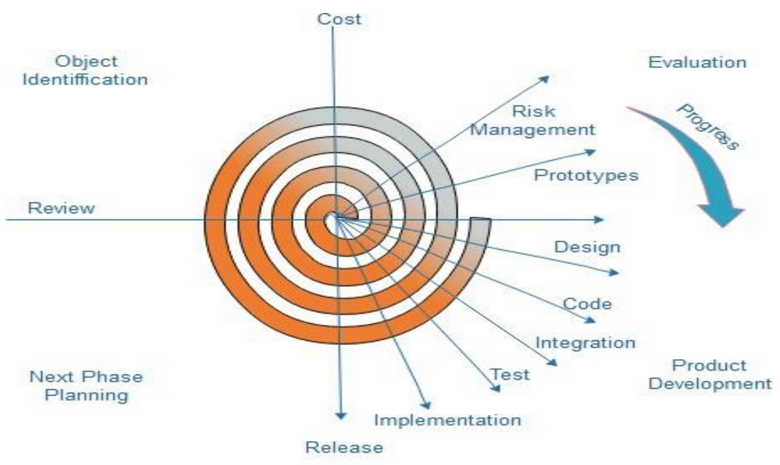


Fig. Spiral Model

Each cycle in the spiral is divided into four parts:

Objective setting: Each cycle in the spiral starts with the identification of purpose for that cycle, the various alternatives that are possible for achieving the targets, and the constraints that exists.

Risk Assessment and reduction: The next phase in the cycle is to calculate these various alternatives based on the goals and constraints. The focus of evaluation in this stage is located on the risk perception for the project.

Development and validation: The next phase is to develop strategies that resolve uncertainties and risks. This process may include activities such as benchmarking, simulation, and prototyping.

Planning: Finally, the next step is planned. The project is reviewed, and a choice made whether to continue with a further period of the spiral.

When to use Spiral Model?

- When deliverance is required to be frequent.
- When the project is large
- When requirements are unclear and complex
- When changes may require at any time
- Large and high budget projects

Advantages

- High amount of risk analysis
- Useful for large and mission-critical projects.

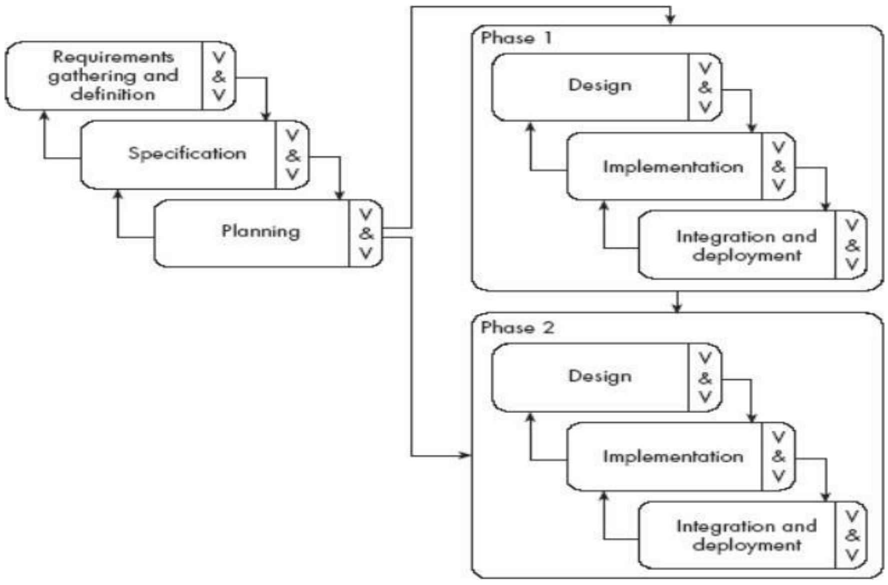
Disadvantages

- Can be a costly model to use.
- Risk analysis needed highly particular expertise
- Doesn't work well for smaller projects.

PHASE RELEASE MODEL

It introduces the notion of incremental development.

- After requirements gathering and planning, the project should be broken intoseparate subprojects, or phases.
- Each phase can be released to customers when ready.
- Parts of the system will be available earlier than when using a strict waterfallapproach.
- However, it continues to suggest that all requirements be finalized at the start ofdevelopment.

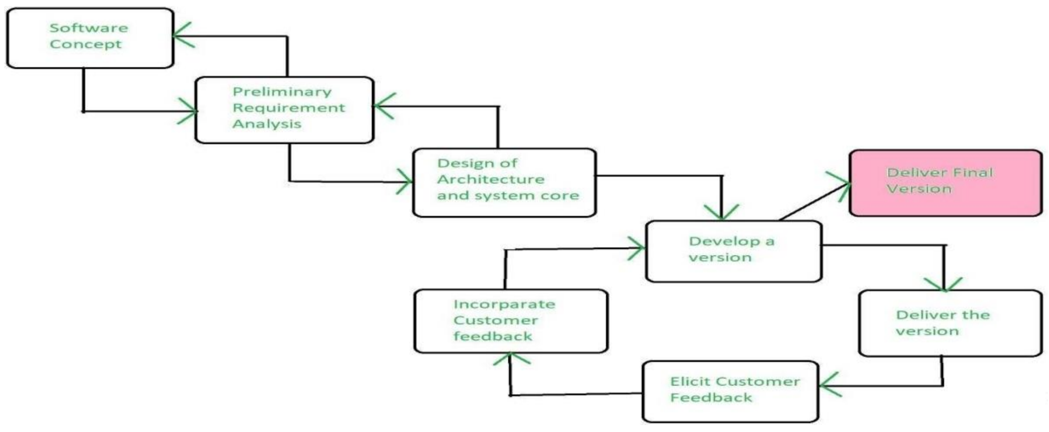

Requirements gathering and definition
V & V
Specification
Planning
Phase 1
Design
Implementation
Integration and deployment
Phase 2
Design
Implementation
Integration and deployment

EVOLUTIONARY MODEL

Evolutionary model is a combination of Iterative and Incremental model of software development life cycle. t is better for software products that have their feature sets redefined during development because of user feedback and other factors. The Evolutionary development model divides the development cycle into smaller, incremental waterfallmodels in which users are able to get access to the product at the end of each cycle.

Feedback is provided by the users on the product for the planning stage of the next cycle andthe development team responds, often by changing the product, plan or process. Therefore,the software product evolves with time.

All the models have the disadvantage that the duration of time from start of the project to the delivery time of a solution is very high. Evolutionary model solves this problem in a different approach.


Software Concept
Preliminary Requirement Analysis
Design of Architecture and system core
Develop a version
Deliver Final Version
Deliver the version
Elicit Customer Feedback
Incorparate Customer feedback

Evolutionary model suggests breaking down of work into smaller chunks, prioritizing themand then delivering those chunks to the customer one by one.

Application of Evolutionary Model:

1. It is used in large projects where you can easily find modules for incremental implementation. Evolutionary model is commonly used when the customer wants tostart using the core features instead of waiting for the full software.
2. Evolutionary model is also used in object oriented software development because thesystem can be easily portioned into units in terms of objects.

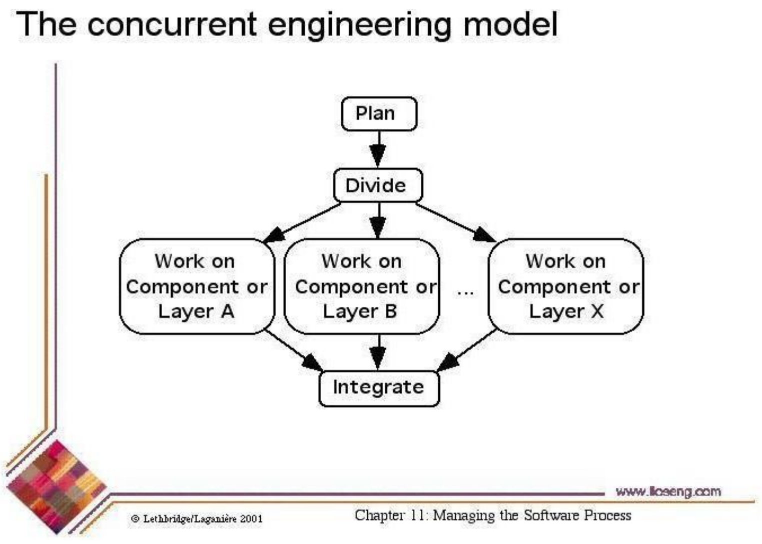
Advantages:

- In evolutionary model, a user gets a chance to experiment partially developed system.
- It reduces the error because the core modules get tested thoroughly.

Disadvantages:

- Sometimes it is hard to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented and delivered.

<u>CONCURRENT ENGINEERING MODEL</u>


The concurrent engineering model
Plan
Divide
Work on Component or Layer A
Work on Component or Layer B
...
Work on Component or Layer X
Integrate
www.lloseng.com
© Lethbridge/Laganière 2001
Chapter 11: Managing the Software Process

It explicitly accounts for the divide and conquer principle.

- Each team works on its own component, typically following a spiral orevolutionary approach.
- There has to be some initial planning, and periodic integration.

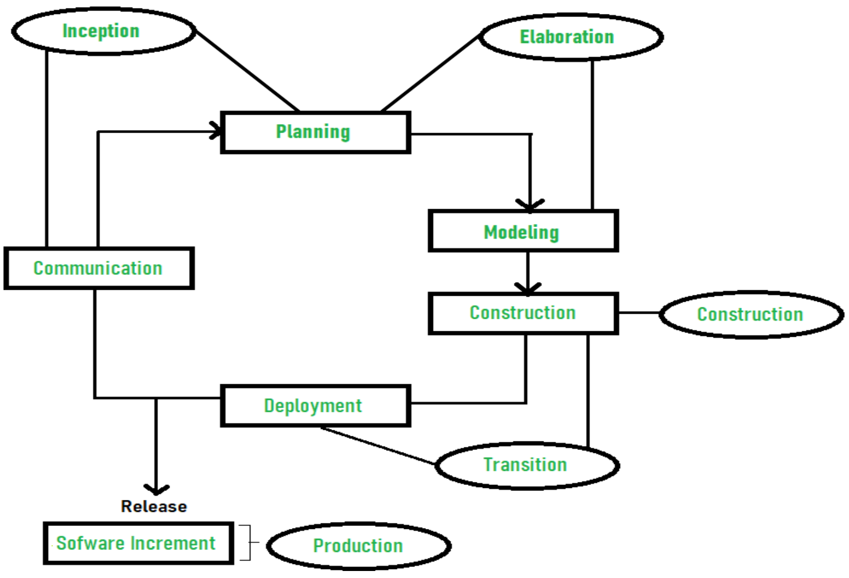
RUP AND ITS PHASES

Rational Unified Process (RUP) is a software development process for object-oriented models. It is also known as the Unified Process Model. It is created by Rational corporation and is designed and documented using UML (Unified Modeling Language). This process is included in IBM Rational Method Composer (RMC) product. IBM (International Business Machine Corporation) allows us to customize, design, and personalize the unified process.

RUP is proposed by Ivar Jacobson, Grady Bootch, and James Rambaugh. Some characteristics of RUP include use-case driven, Iterative (repetition of the process), and Incremental (increase in value) by nature, delivered online using web technology, can be customized or tailored in modular and electronic form, etc. RUP reduces unexpected development costs and prevents wastage of resources.

Phases of RUP :

There are total five phases of life cycle of RUP:



1. **Inception –**

- Communication and planning are main.
- Identifies Scope of the project using use-case model allowing managers toestimate costs and time required.
- Customers requirements are identified and then it becomes easy to make a planof the project.
- Project plan, Project goal, risks, use-case model, Project description, are made.
- Project is checked against the milestone criteria and if it couldn't pass thesecriteria then project can be either cancelled or redesigned.

2. **Elaboration –**

- Planning and modeling are main.
- Detailed evaluation, development plan is carried out and diminish the risks.
- Revise or redefine use-case model (approx. 80%), business case, risks.
- Again, checked against milestone criteria and if it couldn’t pass these criteriathen again project can be cancelled or redesigned.
- Executable architecture baseline.

3. **Construction –**

- Project is developed and completed.
- System or source code is created and then testing is done.
- Coding takes place.

4. **Transition –**

- Final project is released to public.
- Transit the project from development into production.
- Update project documentation.
- Beta testing is conducted.
- Defects are removed from project based on feedback from public.

5. **Production –**

- Final phase of the model.
- Project is maintained and updated accordingly.