**OBJECT ORIENTED SOFTWARE ENGINEERING**

**NOTES**

**MCA III SEMESTER**

**S.Jaswanthi**

**Assistant Professor**

**K.G.R.L P.G College**

**Bhimavaram**

**MCA-20303 OBJECT ORIENTED SOFTWARE ENGINEERING**

**UNIT I**

**Introduction to Object Oriented Software Engineering:** Nature of the Software, Types of Software, Software Engineering Activities, and 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-Centered 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 Behavioural 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.

**OBJECT ORIENTED SOFTWARE ENGINEERING**

**UNIT I**

**Introduction to Object Oriented Software Engineering:** Nature of the Software, Types of Software, Software Engineering Activities, and 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.

**Introduction to Object Oriented Software Engineering**

**Nature of the Software:**

The term software engineering is the product of two words, software, and engineering.Find time to learn programming, Stay 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.

**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.

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:** The necessity of software engineering appears because of a higher rate of progress in user requirements and the environment on which the program is working.

* **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 engineering principles.
* 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 hardwork 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.

\*\*\*\*\*

**Types of Software:**

The software is used extensively in several domains including hospitals, banks, schools, defence, finance, stock markets, and so on. It can be categorized into different types:

**On the basis of application:**

1. **System Software –** System Software is necessary to manage the computer resources and support the execution of application programs. Software like operating systems, compilers, editors and drivers, etc., come under this category. A computer cannot function without the presence of these. Operating systems are needed to link the machine-dependent needs of a program with the capabilities of the machine on which it runs. Compilers translate programs from high-level language to machine language.
2. **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.
3. **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.
4. **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.

1. **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.
2. **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.
3. **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.

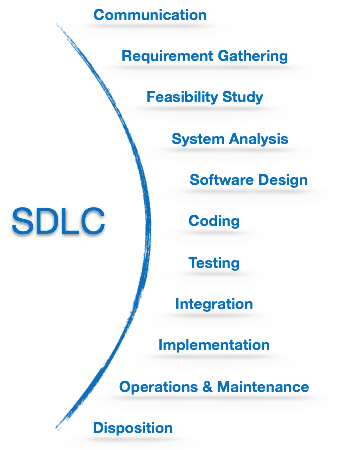
1. **Scientific Software –** Scientific and engineering software satisfies the needs of a scientific or engineering user to perform enterprise-specific tasks. Such software is written for specific applications using principles, techniques, and formulae specific to that field. Examples are software like MATLAB, AUTOCAD, PSPICE, ORCAD, etc.
2. **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.
3. **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.

\*\*\*\*\*

**Software Engineering Activities:**

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:



**Communication:** 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.

**Requirement Gathering:** 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.

**Feasibility Study:** After requirement gathering, the team comes up with a rough plan of software process. At this step the team analyzes if a 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.

**System Analysis:** 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 upgradation. 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.

\*\*\*\*\*

**Software Quality:**

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

**Example:** Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

**The modern view of a quality associated with a software product several quality methods such as the following:**

**Portability:** A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.

**Usability:** A software product has better usability if various categories of users can easily invoke the functions of the product.

**Reusability:** A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

**Correctness:** A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

**Maintainability:** A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

\*\*\*\*\*

**Introduction to Object Orientation**

**Introduction to Object Orientation:**

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 superclass.

**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 superclasses.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.

**\*\*\*\*\***

**Data Abstraction:**

Abstraction means to focus on the essential features of an element or object in OOP, ignoring its extraneous or accidental properties. The essential features are relative to the context in which the object is being used.

Grady Booch has defined abstraction as follows −

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

**Example** − When a class Student is designed, the attributes enrolment\_number, name, course, and address are included while characteristics like pulse\_rate and size\_of\_shoe are eliminated, since they are irrelevant in the perspective of the educational institution.

\*\*\*\*\*

**Inheritance & Polymorphism:**

**Inheritance:-**

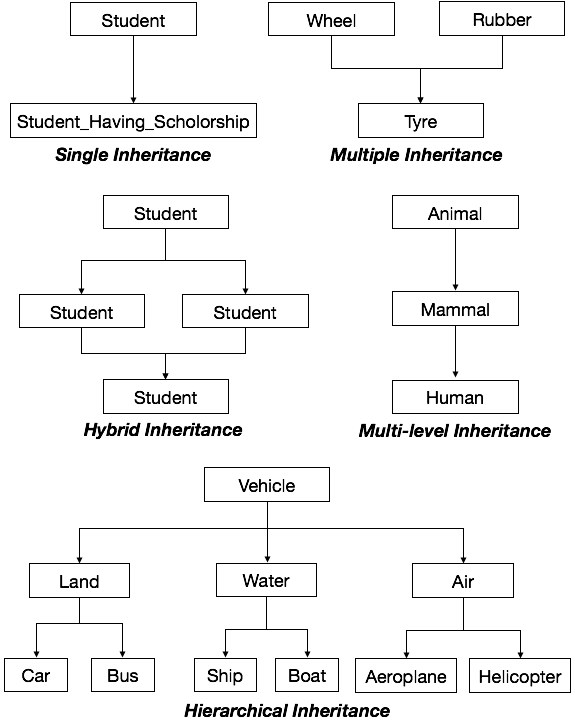
Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an “is – a” relationship.

**Example**:- From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow “is – a” mammal.

**Types of Inheritance:-**

* **Single Inheritance** − A subclass derives from a single super-class.
* **Multiple Inheritance** − A subclass derives from more than one super-classes.
* **Multilevel Inheritance** − A subclass derives from a super-class which in turn is derived from another class and so on.
* **Hierarchical Inheritance** − A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
* **Hybrid Inheritance** − A combination of multiple and multilevel inheritance so as to form a lattice structure.

The following figure depicts the examples of different types of inheritance.



**Polymorphism:-**

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

**Example:-** Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

\*\*\*\*\*

**Reusability in Software Engineering:**

Reusability is a major contributor to the development goal of achieving high speed, low cost and quality. Usually, it is better and faster to employ proven off-the-shelf designs rather than specially-crafted designs that might have problems. Because reusable components are used many times, they are more likely to be adjusted and perfected on the basis of field experience. When this is done, reusable components have a high level of quality. It may be difficult for a development team to achieve this level of quality on a one time basis. Thus, reusability helps achieve quality systems as well as fast development.

In practice, when reusability is employed appropriately, it enables applications to be customized quickly out of components that can be adapted to individual needs. One of the characteristics of reusability is that the facilities for reusability take time to build up, but they eventually enable an enterprise to create most applications rapidly.

**Benefits**  
- It is quicker to assemble applications from existing designs than to design them from scratch.  
- Because it is quicker and easier, design from reusable components is likely to cost less.  
- As reusable designs are thoroughly tested, perfected and comprehensive, the resulting product is less likely to have errors, instability, omissions and misunderstanding. It is more likely to incorporate the most elegant designs and well-thought-out ideas.  
- The reusable design may have a high order of shared expertise built into it.  
- Reusable designs can employ the best techniques. Persons who work with them learn these techniques.  
- Algorithms, structures or knowledge of great complexity may be built into the reusable modules, facilitating the use of powerful techniques. The user or analyst need not be aware of the underlying complexity.  
- Development from reusable components can be done by one person or by a small team.  
- Reusable components are well-documented.  
- Reduction in maintenance work is linked to reduction in development effort. It is quick to create a variation of an existing design.  
- Reusable components in an I-CASE environment are designed so that they are as easy to change as possible with a tool that checks that complete integrity and consistency are maintained.  
- Reusable designs help enforce standards.  
- Consistent design makes systems easier to use and components easier to assemble.  
- Reusable data models, process models and design help achieve integration of systems vertically in the enterprise and horizontally across the value chain.  
- Major business benefits are derived from the standards, integration and ease of change. An IS organization is better positioned to adapt its systems quickly to help the enterprise seize new opportunities quickly.  
- A major advantage of reusable designs is that developers can learn from the designs that are used. The reusable designs ought to employ the best design techniques. Working with reusable designs and modifying them is a good way to learn and gain experience with the techniques.

\*\*\*\*\*

**Examples:**

**Postal Codes:**

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.

element is the postal test class that allows the user to enter postal codes and test the facilities of the postal code hierarchy.

**The postal code hierarchy:**

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 getcountry, are abstract, meaning that they must be 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 postalcode exception if the format of the code is invalid.

• All the instance variables are declared private. All other classes, including

subclasses, can only access them using methods. This helps to improve

encapsulation.

• There is a to string method, as should be provided in most java classes. There are three examples of subclasses of postalcode. Each of these implements the two

abstract operations. For example, the validate method of one subclass,

Canadian postalcode, 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.

**The postalcode Exception class:**

PostalCode 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 it wishes.

**The user interface class PostalTest**:

The user interface class, PostalTest, has only a static main method and one private static helper method called getInput. The code prompts the user for input and then attempts to create an instance of one of the subclasses of PostalCode. If a PostalCode 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 PostalTest into PostalCode- the main method in PostalCode 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.**

**PointCPTest**

getX().

Main()

getY()

getRho()

getTheta()

ConvertStorageToCartesian()

ConvertStorageToPolar()

ToString()

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 PostalTest, 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 either 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:**

Domain analysis is the process by which a software engineer learns background information. He or she has to learn sufficient information so as to be able to understand the problem and make good decisions during requirements analysis and other stages of the software engineering process. The word ‘domain’ in this case means the general field of business or technology in which the customers expect to be using the software.

Some domains might be very broad, such as ‘airline reservations’, ‘medical diagnosis’, and ‘financial analysis’. Others are narrower, such as ‘the manufacturing of paint’ or ‘scheduling meetings’. People who work in a domain and who have a deep knowledge of it (or part of it), are called domain experts. Many of these people may become customers or users.

To perform domain analysis, you gather information from whatever sources of information are available: these include the domain experts; any books about the domain; any existing software and its documentation, and any other documents he or she can find. The interviewing, brainstorming and use case analysis techniques discussed later in this chapter can help with domain analysis. Object oriented modelling, discussed in the next chapter, can also be of assistance.

As a software engineer, you are not expected to become an expert in the domain; nevertheless, domain analysis can involve considerable work. The following benefits will make this work worthwhile:

* **Faster development:** You will be able to communicate with the stakeholders more effectively, hence you will be able to establish requirements more rapidly. Having performed domain analysis will help you to focus on the most important issues.
* **Better system:** Knowing the subtleties of the domain will help ensure that the solutions you adopt will more effectively solve the customer’s problem. You will make fewer mistakes, and will know which procedures and standards to follow. The analysis will give you a global picture of the domain of application; this will lead to better abstractions and hence improved designs.
* **Anticipation of extensions:** Armed with domain knowledge, you will obtain insights into emerging trends and you will notice opportunities for future development. This will allow you  to build a more adaptable system.

\*\*\*\*\*

**Problem Definition and Scope:**

A typical software development process follows these steps −

* Requirement gathering
* Problem definition
* System design
* Implementation
* Testing
* Documentation
* Training and support
* Maintenance

The first two steps assist the team in understanding the problem, the most crucial first step towards getting a solution. Person responsible for gathering requirement, defining the problem and designing the system is called **system analyst**.

**Requirement Gathering**

Usually, clients or users are not able to clearly define their problems or requirements. They have a vague idea of what they want. So system developers need to gather client requirements to understand the problem that needs to be resolved, or what needs to be delivered. Detailed understanding of the problem is possible only by first understanding the business area for which the solution is being developed. Some key questions that help in understanding a business include −

* What is being done?
* How is it being done?
* What is the frequency of a task?
* What is the volume of decisions or transactions?
* What are the problems being encountered?

Some techniques that help in gathering this information are −

* Interviews
* Questionnaires
* Studying existing system documents
* Analyzing business data

System analysts needs to create clear and concise but thorough requirements document in order to identify SMART – specific, measurable, agreed upon, realistic and time-based – requirements. A failure to do so results in −

* Incomplete problem definition
* Incorrect program goals
* Re-work to deliver required outcome to client
* Increased costs
* Delayed delivery

Due to the depth of information required, requirement gathering is also known as **detailed investigation**.

**Problem Definition:-** After gathering requirements and analyzing them, problem statement must be stated clearly. Problem definition should unambiguously state what problem or problems need to be solved. Having a clear problem statement is necessary to −

* Define project scope
* Keep the team focused
* Keep the project on track
* Validate that desired outcome was achieved at the end of project

\*\*\*\*\*

**Types of Requirements:**

### Functional Requirements:-

Requirements, which are related to functional aspect of software fall into this category.

They define functions and functionality within and from the software system.

#### Examples -

* Search option given to user to search from various invoices.
* User should be able to mail any report to management.
* Users can be divided into groups and groups can be given separate rights.
* Should comply business rules and administrative functions.
* Software is developed keeping downward compatibility intact.

### Non-Functional Requirements:-

Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software, which users make assumption of.

Non-functional requirements include -

* Security
* Logging
* Storage
* Configuration
* Performance
* Cost
* Interoperability
* Flexibility
* Disaster recovery
* Accessibility

Requirements are categorized logically as

* **Must Have** : Software cannot be said operational without them.
* **Should have** : Enhancing the functionality of software.
* **Could have** : Software can still properly function with these requirements.
* **Wish list** : These requirements do not map to any objectives of software.

While developing software, ‘Must have’ must be implemented, ‘Should have’ is a matter of debate with stakeholders and negation, whereas ‘could have’ and ‘wish list’ can be kept for software updates.

## User Interface requirements:-

UI is an important part of any software or hardware or hybrid system. A software is widely accepted if it is -

* easy to operate
* quick in response
* effectively handling operational errors
* providing simple yet consistent user interface

User acceptance majorly depends upon how user can use the software. UI is the only way for users to perceive the system. A well performing software system must also be equipped with attractive, clear, consistent and responsive user interface. Otherwise the functionalities of software system can not be used in convenient way. A system is said be good if it provides means to use it efficiently. User interface requirements are briefly mentioned below -

* Content presentation
* Easy Navigation
* Simple interface
* Responsive
* Consistent UI elements
* Feedback mechanism
* Default settings
* Purposeful layout
* Strategical use of color and texture.
* Provide help information
* User centric approach
* Group based view settings.

\*\*\*\*\*

**Techniques for Gathering and Analyzing Requirements:**

Techniques describe how tasks are performed under specific circumstances. A task may have none or one or more related techniques. A technique should be related to at least one task.

The following are some of the well-known requirements gathering techniques

**Brainstorming:** Brainstorming is used in requirement gathering to get as many ideas as possible from group of people. Generally used to identify possible solutions to problems, and clarify details of opportunities.

**Document Analysis:** Reviewing the documentation of an existing system can help when creating AS–IS process document, as well as driving gap analysis for scoping of migration projects. In an ideal world, we would even be reviewing the requirements that drove creation of the existing system – a starting point for documenting current requirements. Nuggets of information are often buried in existing documents that help us ask questions as part of validating requirement completeness.

**Focus Group:** A focus group is a gathering of people who are representative of the users or customers of a product to get feedback. The feedback can be gathered about needs/opportunities/ problems to identify requirements, or can be gathered to validate and refine already elicited requirements. This form of market research is distinct from brainstorming in that it is a managed process with specific participants.

**Interface analysis:** Interfaces for a software product can be human or machine. Integration with external systems and devices is just another interface. User centric design approaches are very effective at making sure that we create usable software. Interface analysis – reviewing the touch points with other external systems is important to make sure we don’t overlook requirements that aren’t immediately visible to users.

**Interview:** Interviews of stakeholders and users are critical to creating the great software. Without understanding the goals and expectations of the users and stakeholders, we are very unlikely to satisfy them. We also have to recognize the perspective of each interviewee, so that, we can properly weigh and address their inputs. Listening is the skill that helps a great analyst to get more value from an interview than an average analyst.

**Observation:** By observing users, an analyst can identify a process flow, steps, pain points and opportunities for improvement. Observations can be passive or active (asking questions while observing). Passive observation is better for getting feedback on a prototype (to refine requirements), where active observation is more effective at getting an understanding of an existing business process. Either approach can be used.

**Prototyping:** Prototyping is a relatively modern technique for gathering requirements. In this approach, you gather preliminary requirements that you use to build an initial version of the solution - a prototype. You show this to the client, who then gives you additional requirements. You change the application and cycle around with the client again. This repetitive process continues until the product meets the critical mass of business needs or for an agreed number of iterations.

**Requirement Workshops:** Workshops can be very effective for gathering requirements. More structured than a brainstorming session, involved parties collaborate to document requirements. One way to capture the collaboration is with creation of domain-model artifacts (like static diagrams, activity diagrams). A workshop will be more effective with two analysts than with one.

**Reverse Engineering:** When a migration project does not have access to sufficient documentation of the existing system, reverse engineering will identify what the system does. It will not identify what the system should do, and will not identify when the system does the wrong thing.

**Survey/Questionnaire:** When collecting information from many people – too many to interview with budget and time constraints – a survey or questionnaire can be used. The survey can force users to select from choices, rate something (“Agree Strongly, agree…”), or have open ended questions allowing free-form responses. Survey design is hard – questions can bias the respondents.

\*\*\*\*\*

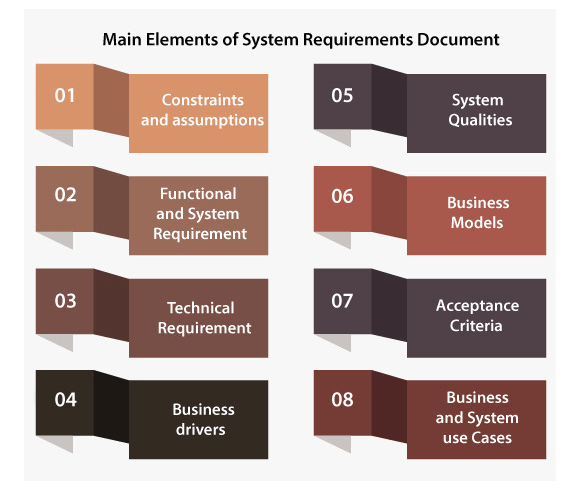
**Requirement Documents:**

System Requirements Document is also known as System Requirements Specifications*.* System requirements document is a set of documentation that describes the behavior and features of a software or system. It comprises of various elements that attempt to characterize the functionality needed by the client to satisfy their users. In other words, the system requirements document (SRD) describes the system-level performance and functional requirements for a system.

System Requirements Document or System Requirements Specification is defined as a document which defines what the software will do and how it will be required to perform, and it also defines the functionality the software needs to satisfy all stakeholders (users, business) requirements.

**Main Elements of System Requirements Document:-**

Based on the procedure employed (Waterfall vs. agile), the degree of convention and detail in the SRS will differ, however in general, system requirement document or system requirement specification contains a description of the system requirements, functional requirements, technical requirements, acceptance criteria, assumptions and constraints. We have explained each of these in more detail below:



**Constraints and Assumption:-**

The constraints and assumption section will underline the lack of any design imposed by the client on the system design, resulting in removing some options from being considered by the developers. This section also comprises assumptions which are made by the team of requirement engineering at the time of requirements gathering and analysis. If there is an incorrect assumption, it is needed to re-evaluate the system requirements specification in order to ensure that the documented requirements are still valid.

**Functional and System Requirements:-**

This segment normally comprises a hierarchical arrangement of requirements, with the functional/business requirements at the uppermost level and the detailed system requirements are listed as their child items.

Mostly in the form of statements, the requirements are written like "System requires the capability to do x," with supporting information comprised as required.

**Technical Requirements:-**

This part is utilized to list any of the non-functional requirements, which basically personifies the technical environment that the product requires to work in and incorporate the technical limitations that it requires to work under. These technical requirements are important in deciding how high-level functional requirements can decompose into more definite system requirements.

**Business Drivers:-**

This part depicts the reasons why the client is hoping to build the system. The new system's rationale is significant because it will manage the choices made by business experts, engineers, and system architects. Another convincing purpose behind recording the business logic and behind the system is that during the project, client can change the staff. Documentation that absolutely recognizes the business explanations for the system will help to continue support the project if the original sponsor proceeds onward.

The drivers may comprise issues as well as opportunities. Generally, a mix of issues and opportunities are required to give inspiration to a new system.

**System Qualities:-**

The system qualities are the non-functional requirements which are used to define the system's quality. These qualities are frequently known as "ilities," and the reason behind that is many of these qualities end with "ility". They are comprised of items like availability, maintainability, security, reliability, and scalability.

Unlike functional requirements, the quality of the system generally contains tables of particular metrics that the system should meet to be acknowledged.

**Business Models:-**

This part portrays the fundamental business model of the client that the system will require to support. This contains current-state, future-state and organizational context state diagrams, key business functions, business context and process flow diagrams. This part is generally made during the functional analysis phase.

**Acceptance Criteria:-**

This part will portray the measures by which the client will "sign-off" the final system. Based on the procedure, this may occur toward the end of the testing and quality assurance stage, or in agile methodology, toward the finish of every iteration.

Usually, this measure refers to the requirement in order to complete all user acceptance tests and fix all bugs/defects which meet a pre-defined priority or severity limits.

**Business and System Use Cases:-**

This segment ordinarily comprises a UML use case diagram, which shows the main external entities that will collaborate with the system along with diverse use cases that should be met. There will be a formal definition of the steps for every use case which is required to fulfill the purpose of the business, along with the essential pre-conditions and post-conditions.

Generally, the system use cases are derived from the system requirements and the business use case is derived from the functional requirements. By using the flowchart, the steps of use cases are represented.

\*\*\*\*\*

**Reviewing Requirements:**

A requirements review is a manual process that involves people from both client and contractor organisations. They check the requirements document for anomalies and omissions. The review process may be managed in the same way as program inspections (see Chapter 22). Alternatively, it may be organised as a broader activity with different people checking different parts of the document.

Requirements reviews can be informal or formal. Informal reviews simply involve contractors discussing requirements with as many system stakeholders as possible. It is surprising how often communication between system developers and stakeholders ends after elicitation and there is no confirmation that the documented requirements are what the stakeholders really said they wanted. Many problems can be detected simply by talking about the system to stakeholders before making a commitment to a formal review.

In a formal requirements review, the development team should ‘walk’ the client through the system requirements explaining the implications of each requirement. The review team should check each requirement for consistency and should check the requirements as a whole for completeness. Reviewers may also check for:

* Verifiability Is the requirement as stated realistically testable?
* Comprehensibility Do the procurers or end-users of the system properly understand the requirement?
* Traceability Is the origin of the requirement clearly stated?  You may have to go back to the source of the requirement to assess the impact of a change. Traceability is important as it allows the impact of change on the rest of the system to be assessed.
* Adaptability Is the requirement adaptable? That is, can the requirement be changed without large-scale effects on other system requirements.

\*\*\*\*\*

**Case Studies:**

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 that must meet stringent assurance and evaluation requirements. 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-physical threats. We describe our team-based method for developing a requirements document and relate that process to techniques in requirements engineering. The system requirements document presents a calibration point for future security requirements engineering techniques intended to meet both functional and assurance goals.

\*\*\*\*\*

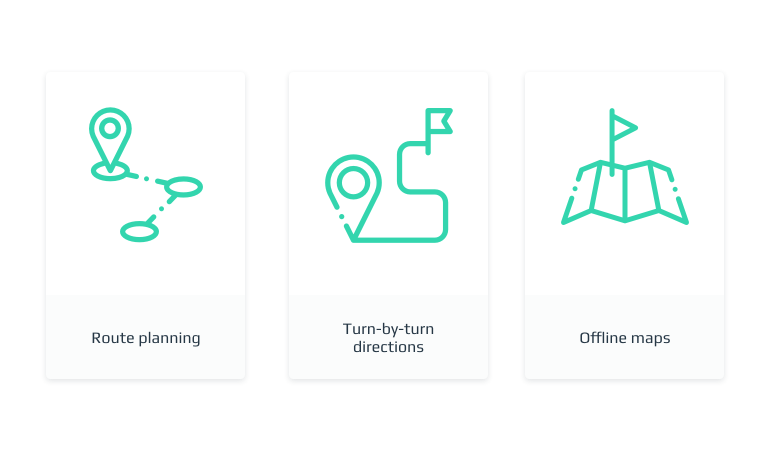
**GPS based Automobile Navigation System:**

Quite simply, a navigation system helps drivers get from point A to point B. What’s not that simple, however, is the software itself, which is actually quite intricate. Embedded navigation systems are becoming more advanced as voice guidance, live traffic reports, over-the-air updates, and point of interest (POI) search optimize the driving experience.

Everyone who has a smartphone or a car has probably used some navigation system. With [79 million](https://www.statista.com/statistics/200002/international-car-sales-since-1990/) vehicles expected to be sold globally in 2019, it’s not surprising that the automotive navigation system market is expected to increase at a [compound annual growth rate of 5.2%](https://www.technavio.com/report/global-automotive-navigation-systems-market-analysis-share-2018) from 2019 to 2022.

Essential features in automotive navigation

Along with additional options, there are fundamental in-vehicle navigation systems features that make driving efficient and safe. They include:



Let’s dig deeper into each of these features.

**Route planning and waypoints:-** Planning a route is the easiest and most basic thing a navigation system should be able to do. Usually, a navigation system offers alternative ways based on your preference for the fastest, the cheapest, or the most convenient route. You can also choose to avoid borders, ferries, tolls, highways, etc. The system will guide you to the necessary lane as well as automatically reroute you if you deviate from the initial course. Some embedded navigation systems allow adding up to 50 waypoints. Plus, you can usually see the history of your trips and choose to repeat them or backtrack.

**Turn-by-turn directions:-** Having a navigation system voice your every turn might be somewhat annoying, but it’s immensely useful. Turn-by-turn directions include visual data on the screen along with voice instructions, so your attention can remain on the road. The system voices the direction of the turn, street names, and the distance to the next turn. It can also warn you about traffic congestion and toll roads ahead.

**Offline maps:-** Unfortunately, you can’t always rely on a cell connection, especially when you travel to remote areas. So an offline map is among the most essential features in automotive navigation. Map updates ensure that the street you’re turning down still exists. And while offline maps don’t provide POIs or traffic reports, they can navigate you by showing the current position of your car according to the GPS tracker.

\*\*\*\*\*

**Simple Chat Instant Messaging System:**

**A. Problem.** Sometimes a user wants to prevents message received by a givenclient from appearing on his or her screen. This might be because the clients user is being deliberately annoying, or because the clients user is sending lots of public messages that are useful to other users but are not useful to the current user.

We therefore wish to add a facility to simple chat that will allow a given user to

block messages coming from another specified user.

**B. Background information**. See the requirements for simple chat phase2 for the system on which these requirements are based. The features described in these requirements are part of phase 3.

Issues considered:

Issue 1: can the user block more than one other user at a time?

Decision: yes; however, he or she will have to issue a sequence of block commands.

Issue 2 : it would be useful for a user to know if anyone is blocking messages that

come from him or her.

Decision: add a command called # whoblocksme that will do this.

Issue 3: should the server be able to block messages?

Option 3.1: do not allow the server to block messages.

Advantage: this would be simpler.

Option 3.2 : the server should be able to block ordinary messages from clients, but not

administrative messages such as # forward ect.

Advantage: this would prevent a malicious user from overwhelming the servers

display.

Decision: choose option 3.2

Issue 4: what types of messages should the user be able to block?

Option 4.1: block only private messages,but not public or channel messages.

Advantage: the user can always avoid public or channel messages by changing

channel.

Disadvantage: forcing the user to change channel is not really satisfactory- what if the

user is interested in other messages on that channel?

Option 4.2: block private, public and channel messages.

Decision: choose option 4.2

Issue 5: how should this feature interact with forwarding? The problem: imagine we

have clients A,B and C with set to forward all messages to B.

Issue 6 : Does it make sense to block messages from myself?

Decision: No.

Issue 7: can a user block messages from the server?

Decision: yes.

Issue 8 : how does a user unblock messages?

Decision: the # unblock command with no arguments will cancel all blocking that a

user has set up. The # unblock command with an argument will cancel blocking for

messages from that user only.

Issue 9: any user could circumvent blocking by logging in using a different loginID.

Option 9.1: stop allowing users to create their own loginIDs.

Advantage: would solve this problem.

Disadvantage: would make the system less useful in a chat environment.

Issue 10: should it be possible to block users who are not even logged on; and should

a block persist even if the blocked user logs off and logs on again?

Decision: yes, to both questions, because otherwise a user could circumvent the

blocking by logging off and on.

**C. Environment and system models**: this feature is to be added to phase 3 of simple

chat, at the same time as the forwarding and channels features are added. There are no

additional environmental considerations for these requirements.

**D. Functional requirements**

1. General

• Commands. Each of these can be issued from the user interface of the client or

server. If issued from the client UI they will be transmitted to the server

unchanged.

• #block<user>

• Initiate blocking for any user named<user>,except self.

• Works whether <user> is connected or not.

• Displays a message on the originating UI that states:

• Messages from <user> will be blocked.

• An attempt to block messages to self will cause the following message to be

displayed:

• You cannot block the sending of messages to yourself.

**E. Other requirements**. There are no additional qualities, platform or process

requirements for this feature. All such requirements of the base system still apply.

\*\*\*\*\*

**UNIT II**

**Unified Modeling Language & Use Case Modeling:** Introduction to UML, Modeling Concepts, Types of UML Diagrams with Examples; User-Centered 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 Behavioural Diagrams: Interaction Diagrams, State Diagrams, Activity Diagrams, Component and Deployment Diagrams.

**Unified Modeling Language & Use Case Modeling**

**Introduction to UML:**

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

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

OMG is continuously making efforts to create a truly industry standard.

* UML stands for **Unified Modeling Language**.
* UML is different from the other common programming languages such as C++, Java, COBOL, etc.
* UML is a pictorial language used to make software blueprints.
* UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
* Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization, UML has become an OMG standard.

**Goals of UML:-** 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.

**A Conceptual Model of UML:-**

To understand the conceptual model of UML, first we need to clarify what is a conceptual model? and why a conceptual model is required?

* A conceptual model can be defined as a model which is made of concepts and their relationships.
* A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other.

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

* UML building blocks
* Rules to connect the building blocks
* Common mechanisms of UML

**Role of UML in OO Design:-**

* UML is a modeling language used to model software and non-software systems. Although UML is used for non-software systems, the emphasis is on modeling OO software applications. Most of the UML diagrams discussed so far are used to model different aspects such as static, dynamic, etc. Now whatever be the aspect, the artifacts are nothing but objects.
* If we look into class diagram, object diagram, collaboration diagram, interaction diagrams all would basically be designed based on the objects.
* Hence, the relation between OO design and UML is very important to understand. The OO design is transformed into UML diagrams according to the requirement. Before understanding the UML in detail, the OO concept should be learned properly. Once the OO analysis and design is done, the next step is very easy. The input from OO analysis and design is the input to UML diagrams.

\*\*\*\*\*

**Modeling Concepts:**

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

\*\*\*\*\*

**Types of UML Diagrams with Examples:**

The elements are like components which can be associated in different ways to make a complete UML picture, which is known as diagram. Thus, it is very important to understand the different diagrams to implement the knowledge in real-life systems.

Any complex system is best understood by making some kind of diagrams or pictures. These diagrams have a better impact on our understanding. If we look around, we will realize that the diagrams are not a new concept but it is used widely in different forms in different industries.

We prepare UML diagrams to understand the system in a better and simple way. A single diagram is not enough to cover all the aspects of the system. UML defines various kinds of diagrams to cover most of the aspects of a system.

You can also create your own set of diagrams to meet your requirements. Diagrams are generally made in an incremental and iterative way.

There are two broad categories of diagrams and they are again divided into subcategories −

1.Structural Diagrams

2.Behavioral Diagrams

**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, and nodes. 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** − If the above descriptions and usages are observed carefully then it is very clear that all the diagrams have some relationship with one another. 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 Diagrams:-** 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.Sequence diagram

iii.Collaboration diagram

iv.Statechart diagram

v.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.Sequence Diagram:-** 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.

**iii.Collaboration Diagram:-** 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.

**iv.Statechart Diagram:-** 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.

**v.Activity Diagram:-** 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.

\*\*\*\*\*

**User-Centered Design:**

**User-centred design** (UCD) is a collection of processes which focus on putting users at the center of product design and development. You develop your digital product taking into account your user’s requirements, objectives and feedback. A more formal definition is the one provided by *the* Interaction Design Foundation:

User-centered design (UCD) is an iterative design process in which designers and other stakeholders focus on the users and their needs in each phase of the design process. UCD calls for involving users throughout the design process via a variety of research and design techniques so as to create highly usable and accessible products for them.

**User-centered design vs. Human-centered design:**- User-centered design is very often used interchangeably with human-centered design, but there is a difference in that it is a subset of it. Simply put, all users are humans, but not all humans will be your users (you wish!). Thus, user-centered design requires deeper analysis of users – your target audience. It is not only about general characteristics of a person; it is about particular habits and preferences of target users to come up with right solutions for specific problems.

User-centered design takes into account age, gender, social status, education and professional background, influential factors, product usage expectations and demands and many other important things that may vary for different segments. What is critical for some may be irrelevant for others. User-centered design is about deep research on users’ habits, from their interactions with the product to their vision of how the product should look like and behave.

**User-centered design and UX:-** User-centered design improves the user experience. While it can be applied to almost any product, in this article, we will focus on website or mobile app development. It helps to understand users’ needs and preferences regarding features of a product, task, goals, [user flows](https://www.interaction-design.org/literature/topics/user-flows?ep=ug0), etc. At the end of the day, it has become one of the most important user experience requirements – that of being user-centered. It should be implemented throughout the entire [customer experience](https://www.interaction-design.org/literature/topics/customer-experience), no guessing, no personal opinion. What matters is what your users say and do. Every “touchpoint” that the customer has with the product should be analysed, well design and developed.

**User-Centered Design Principles:-**There are five major UCD principles

1. A clear understanding of user and task requirements.
2. Incorporating user feedback to define requirements and design.
3. Early and active involvement of the user to evaluate the design of the product.
4. Integrating user-centred design with other development activities.
5. Iterative design process

It is quite simple – if you change the design late in the process, then it will typically cost ten times more than if you changed it during the requirements stage. Analysis and feedback are critical. User-centered design makes sure that you design and develop products right from the beginning, exactly what your clients want.

**The Essential Elements of User-Centered Design:-**

* **Visibility**: Users should be able to see from the beginning what they can do with the product, what is it about, how they can use it.
* **Accessibility:** Users should be able to find information easily and quickly. They should be offered various ways to find information for example call to action buttons, search option, menu, etc.
* **Legibility:** Text should be easy to read. As simple as that.
* **Language:** Short sentences are preferred here. The easier the phrase and the words, the better.

\*\*\*\*\*

**Characteristics of Users:**

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.

\*\*\*\*\*

**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 behaviour 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 fact that 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 in order 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 order to 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 use case 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:**

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

* Used to gather the requirements of a system.
* Used to get an outside view of a system.
* Identify the external and internal factors influencing the system.
* Show the interaction among the requirements are actors.

**How to Draw a Use Case Diagram?**

Use case diagrams are considered for high level requirement analysis of a system. When the requirements of a system are analyzed, the functionalities are captured in use cases.

We can say that use cases are nothing but the system functionalities written in an organized manner. The second thing which is relevant to use cases are the actors. Actors can be defined as something that interacts with the system.

Actors can be a human user, some internal applications, or may be some external applications. When we are planning to draw a use case diagram, we should have the following items identified.

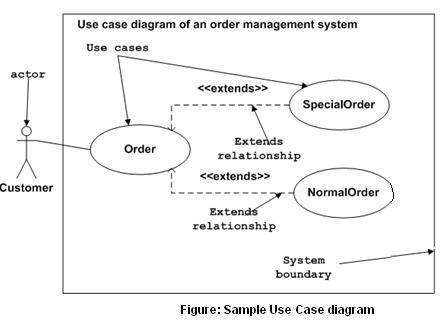
* Functionalities to be represented as use case
* Actors
* Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram

* The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
* Give a suitable name for actors.
* Show relationships and dependencies clearly in the diagram.
* Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
* Use notes whenever required to clarify some important points.

Following is a sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases **(Order, SpecialOrder, and NormalOrder)** and one actor which is the customer.

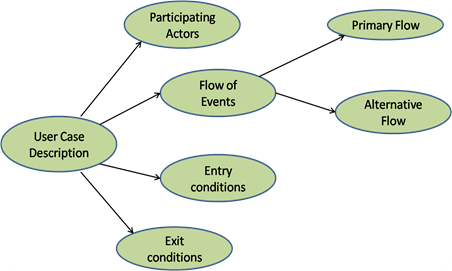
The SpecialOrder and NormalOrder use cases are extended from *Order* use case. Hence, they have extended relationship. Another important point is to identify the system boundary, which is shown in the picture. The actor Customer lies outside the system as it is an external user of the system.

****

\*\*\*\*\*

**Use Case Descriptions:**

A Use Case description is a detailed description of for each of the classes of interaction within a given Use Case Diagram:



So as a quick example, a Use Case Description for say Article Search would be something like:

|  |  |
| --- | --- |
| **Use Case Name** | Searching for an Article |
| **Participating Actors** | Library User |
| **Flow of Events** | 1. User enters search terms 2. System searches’ articles from file 3. System displays all articles matching terms 4. User searches for article from given list and clicks on the one they need 5. System displays that article to the user |
| **Alternative Flows** | 2.a   System has no articles to search 2.b   System informs user and exits 3.a   System finds no articles matching users terms 3.b   System informs user and exits 4.a   User can’t find article in the list 4.b   User exits and tries more search terms |
| **Entry Conditions** | User has correct user authentication and a valid search term |
| **Exit Conditions** | User finds correct article, or user exits searching system due to article not being found |

REPORT THIS AD

\*\*\*\*\*

**The Basics of User Interface Design:**

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction.

UI can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

* Attractive
* Simple to use
* Responsive in short time
* Clear to understand
* Consistent on all interfacing screens

UI is broadly divided into two categories:

1.Command Line Interface

2.Graphical User Interface

**1.Command Line Interface (CLI):-**

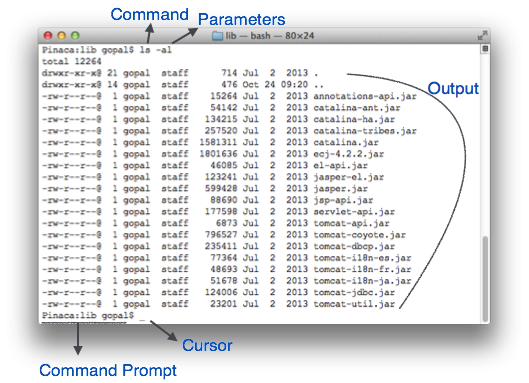
CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate.

CLI uses less amount of computer resource as compared to GUI.

**CLI Elements:-**



A text-based command line interface can have the following elements:

* **Command Prompt** - It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.
* **Cursor** - It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.
* **Command** - A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

**2.Graphical User Interface:-**

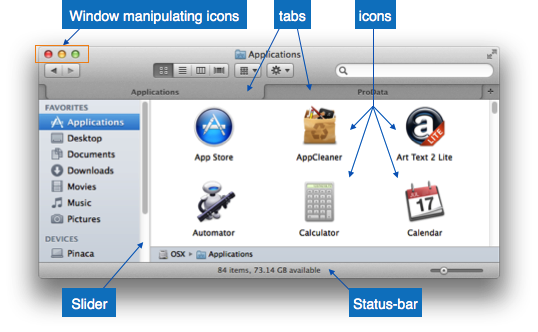
Graphical User Interface provides the user graphical means to interact with the system. GUI can be combination of both hardware and software. Using GUI, user interprets the software.

Typically, GUI is more resource consuming than that of CLI. With advancing technology, the programmers and designers create complex GUI designs that work with more efficiency, accuracy and speed.

**GUI Elements:-**

GUI provides a set of components to interact with software or hardware.

Every graphical component provides a way to work with the system. A GUI system has following elements such as:

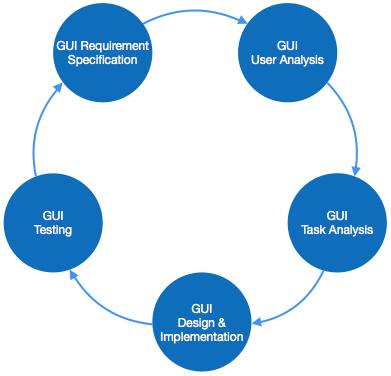


* **Window** - An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure. It is easier for a user to navigate in the file system in an exploring window. Windows can be minimized, resized or maximized to the size of screen. They can be moved anywhere on the screen. A window may contain another window of the same application, called child window.
* **Tabs** - If an application allows executing multiple instances of itself, they appear on the screen as separate windows.**Tabbed Document Interface** has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers use this feature.
* **Menu** - Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window. The menu can be programmed to appear or hide on mouse clicks.
* **Icon** - An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened. Icon displays application and programs installed on a system in the form of small pictures.
* **Cursor** - Interacting devices such as mouse, touch pad, digital pen are represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.

**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** - The designers may like to have list of all functional and non-functional requirements of GUI. This can be taken from user and their existing software solution.
* **User Analysis** - The designer studies who is going to use the software GUI. The target audience matters as the design details change according to the knowledge and competency level of the user. If user is technical savvy, advanced and complex GUI can be incorporated. For a novice user, more information is included on how-to of software.
* **Task Analysis** - Designers have to analyze what task is to be done by the software solution. Here in GUI, it does not matter how it will be done. Tasks can be represented in hierarchical manner taking one major task and dividing it further into smaller sub-tasks. Tasks provide goals for GUI presentation. Flow of information among sub-tasks determines the flow of GUI contents in the software.
* **GUI Design & implementation** - Designers after having information about requirements, tasks and user environment, design the GUI and implements into code and embed the GUI with working or dummy software in the background. It is then self-tested by the developers.
* **Testing** - GUI testing can be done in various ways. Organization can have in-house inspection, direct involvement of users and release of beta version are few of them. Testing may include usability, compatibility, user acceptance etc.

\*\*\*\*\*

**Usability Principles:**

## Effectivity:Reach the goal. (Principle 1):-

The first and therefore most important rule is the observance of effectivity. An application should enable the user to work effectively by providing a suitable functionality for the completion of his task. The user is not overloaded with information and functions that are not necessary for his use case. Accordingly, dialogs should only contain relevant or frequently used information. Every additional information distracts the user from the important content and brings the risk of reducing relative visibility.

It is therefore necessary to answer the central question: What is the task to be supported? An in-depth analysis of workflows and their comprehension is the key to understand the tasks and goals of the users to be able to evaluate the efficiency of the implementation.

**Efficiency: Reach the goal quickly. (Principle 2):-**

Particular attention also needs to be paid to the optimisation of efficiency. The user should be able to accomplish a task in minimal time. At the same time, the application requires only minimal time to react and execute an action. So, this principal refers to the speed of operation of both users and the system. Indications of users not being able to work efficiently are for example unsuitably used interface widgets, missing default values, misleading search results and high-resolution pictures requiring a fast internet connection.

Signs of low system efficiency, on the other hand, are delayed reactions to input in the form of a “shaking” cursor, slow interface widgets and missing feedback.

**Controllability: The user has the power! (Principle 3):-**

The user has, regarding his authority within the system, control over all actions. He may cancel, pause, and resume actions at a later point in time. It is possible for him to revert an action and, as the case may be, start a new attempt if the result was not satisfactory. To control the application, the user still has the option of using alternative input devices to the mouse, such as keyboard, screen reader and Braille display, according to his personal requirements.

Not satisfying these points will lead to increased frustration for the user.

**Customisability: Everything suits the user. (Principle 4):-**

The application should be flexible enough to support different approaches for the completion of a task. Additionally, the user can customize the system according to his previous experiences, culture and needs e.g. by changing font size, changing the default settings, and setting his own shortcuts. A good example for the consideration of customisability is the online platform <http://www.elster.de/>, which is used for web based registration and submission of tax returns. After logging in, the user is asked to which user group he belongs to, to better adapt to his needs afterwards.

\*\*\*\*\*

**Class Design and Class Diagrams**

**Essentials of UML Class Diagrams:**

The class diagram depicts a static view of an application. It represents the types of objects residing in the system and the relationships between them. A class consists of its objects, and also it may inherit from other classes. A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.

It shows the attributes, classes, functions, and relationships to give an overview of the software system. It constitutes class names, attributes, and functions in a separate compartment that helps in software development. Since it is a collection of classes, interfaces, associations, collaborations, and constraints, it is termed as a structural diagram.

**Purpose of Class Diagrams:-**

The main purpose of class diagrams is to build a static view of an application. It is the only diagram that is widely used for construction, and it can be mapped with object-oriented languages. It is one of the most popular UML diagrams. Following are the purpose of class diagrams given below:

1. It analyses and designs a static view of an application.
2. It describes the major responsibilities of a system.
3. It is a base for component and deployment diagrams.
4. It incorporates forward and reverse engineering.

**Benefits of Class Diagrams:-**

1. It can represent the object model for complex systems.
2. It reduces the maintenance time by providing an overview of how an application is structured before coding.
3. It provides a general schematic of an application for better understanding.
4. It represents a detailed chart by highlighting the desired code, which is to be programmed.
5. It is helpful for the stakeholders and the developers.

**Vital components of a Class Diagram**

The class diagram is made up of three sections:

**1.Upper Section:** The upper section encompasses the name of the class. A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics. Some of the following rules that should be taken into account while representing a class are given below:

* 1. Capitalize the initial letter of the class name.
  2. Place the class name in the center of the upper section.
  3. A class name must be written in bold format.
  4. The name of the abstract class should be written in italics format.

**2.Middle Section:** The middle section constitutes the attributes, which describe the quality of the class. The attributes have the following characteristics:

* 1. The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).

f. The accessibility of an attribute class is illustrated by the visibility factors.

g. A meaningful name should be assigned to the attribute, which

will explain its usage inside the class.

**3.Lower Section:** The lower section contain methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.



\*\*\*\*\*

**Associations and Multiplicity:**

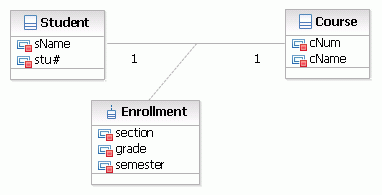
**Associations:-**

In UML diagrams, an association class is a class that is part of an association relationship between two other classes.

You can attach an association class to an association relationship to provide additional information about the relationship. An association class is identical to other classes and can contain operations, attributes, as well as other associations.

For example, a class called Student represents a student and has an association with a class called Course, which represents an educational course. The Student class can enroll in a course. An association class called Enrollment further defines the relationship between the Student and Course classes by providing section, grade, and semester information related to the association relationship.

As the following figure illustrates, an association class is connected to an association by a dotted line.



**Multiplicity:-**

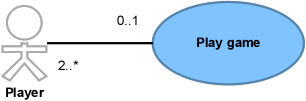
Multiplicity can be set for attributes, operations, and associations in a UML class diagram, and for associations in a use case diagram. The multiplicity is an indication of how many objects may participate in the given relationship or the allowable number of instances of the element.

In a use case diagram, multiplicity indicates how many actors can take part in how many occurrences of a use case. Multiplicity on a use case could mean that an actor interacts with multiple use cases, multiplicity on an actor could mean that one or more actors interact with a particular use case.

**Example**

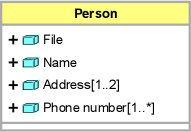
The relationship between a player and the play game use case. A game may be played by many (two or more) players, but a player may not always participate in the game.

To show this relationship, the following multiplicity is set: on the side of the actor **2.., on the side of the use case \*0..1**. This is shown in the diagram as follows:



For a complete registration of a person's details, the person must at least have one address but can have two addresses. Also, the person must have at least one telephone number but may have many.

To show this, the following multiplicity is set: for the address attribute the multiplicity is **1..2**, for the phone number attribute the multiplicity is set to **1..**{\*}. This is shown in the diagram as follows:



\*\*\*\*\*

**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.

Generalization relationship is utilized in class, component, deployment, and use case diagrams to specify that the child inherits actions, characteristics, and relationships from its parent.

To meet UML's standard, it necessitates usage of the same types of model elements in the generalization relationship, i.e., generalization relation can either be used between actors or between use cases, but not between an actor and a use case.

The generalization relationship is incorporated to record attributes, operations, and relationships in a parent model element so that it can be inherited in one or more child model elements.

The parent model element can have as many children, and also, the child can have one or more parents. But most commonly, it can be seen that there is one parent model element and multiple child model elements. The generalization relationship does not consist of names. The generalization relationship is represented by a solid line with a hollow arrowhead pointing towards the parent model element from the child model element.

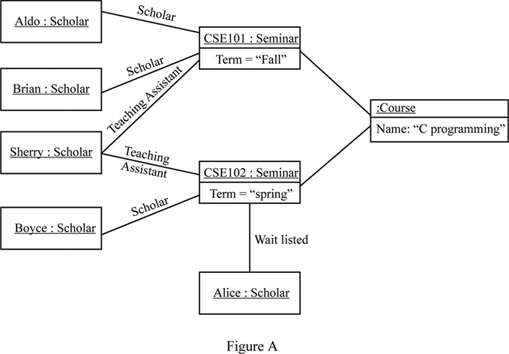


\*\*\*\*\*

**Instance Diagrams:**

Instance diagrams, are occasionally referred to as Object diagrams in UML. Instance diagrams are helpful in exploring the real world instances of objects and also the relationships among them.

Though the class diagrams are good in depicting this information, a few see them to be more abstract. The UML instance diagram [will](https://www.chegg.com/learn/business/business-law/will-in-introduction-to-political-science) be a great choice to depict the highly difficult relationships among the classes.



For instance, the Figure - A depicts the instance diagram. A scholar can go to a “seminar”, can be “Wait listed” for attending it, or can be a “TA” for it.

In the instance diagram, Aldo and Brian are students in CSC101, whereas Sherry is a teaching assistant for CSC101. Sherry is also the “TA” for CSC102, Boyce is a scholar, and Alice is “Wait listed”. The instance diagram also depicts that both the seminars are part of the book “C programming”. This diagram makes the three relationships between seminars and scholars, and two relationships between the course and seminars.

The representations used in UML instance diagrams are simple and easy to understand. They illustrate the objects and the relationships between them. When we illustrate an object we need to incorporate adequate information to make it an identifiable object. Here, we employ one of two formats: A single rectangular box to show the instances, like “Alice: Student” or a single rectangular box separated into two, one showing the instance and other showing the attribute values like “name = C programming”. Link between the objects are illustrated as lines with the roles like “scholar”.

\*\*\*\*\*

**Advanced Features of Class Diagrams:**

Class diagrams offer a number of benefits for any organization. Use UML class diagrams to:

* Illustrate data models for information systems, no matter how simple or complex.
* Better understand the general overview of the schematics of an application.
* Visually express any specific needs of a system and disseminate that information throughout the business.
* Create detailed charts that highlight any specific code needed to be programmed and implemented to the described structure.
* Provide an implementation-independent description of types used in a system that are later passed between its components.

\*\*\*\*\*

**Process of Developing Class Diagrams:**

Class diagrams go hand in hand with [object-oriented design](https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_design.htm). So knowing its basics is a key part of being able to draw good class diagrams.

When required to describe the static view of a system or its functionalities, you’d be required to [draw a class diagram](https://creately.com/diagram-type/class-diagram). Here are the steps you need to follow to create a class diagram.

#### Step 1: Identify the class names

The first step is to identify the primary objects of the system.

#### Step 2: Distinguish relationships

Next step is to determine how each of the classes or objects are related to one another. Look out for commonalities and abstractions among them; this will help you when grouping them when drawing the class diagram.

#### Step 3: Create the Structure

First, add the class names and link them with the appropriate connectors. You can add attributes and functions/ methods/ operations later.

\*\*\*\*\*

**Interaction and Behavioural Diagrams:**

**Interaction Diagrams:-**

Interaction diagrams are a subclass of behavioral diagrams that give emphasis to object interactions and also depicts the flow between various use case elements of a system. In simple words, it shows how objects interact with each other and how the data flows within them. It consists of communication, interaction overview, sequence, and timing diagrams.

* **Sequence Diagram:** It shows the interactions between the objects in terms of messages exchanged over time. It delineates in what order and how the object functions are in a system.
* **Communication Diagram:** It shows the interchange of sequence messages between the objects. It focuses on objects and their relations. It describes the static and dynamic behavior of a system.
* **Timing Diagram:** It is a special kind of sequence diagram used to depict the object's behavior over a specific period of time. It governs the change in state and object behavior by showing the time and duration constraints.
* **Interaction Overview diagram:** It is a mixture of activity and sequence diagram that depicts a sequence of actions to simplify the complex interactions into simple interactions.

**Behavioural Diagrams:-**

Behavioral diagrams portray a dynamic view of a system or the behavior of a system, which describes the functioning of the system. It includes use case diagrams, state diagrams, and activity diagrams. It defines the interaction within the system.

* **State Machine Diagram:** It is a behavioral diagram. it portrays the system's behavior utilizing finite state transitions. It is also known as the **State-charts** diagram. It models the dynamic behavior of a class in response to external stimuli.
* **Activity Diagram:** It models the flow of control from one activity to the other. With the help of an activity diagram, we can model sequential and concurrent activities. It visually depicts the workflow as well as what causes an event to occur.
* **Use Case Diagram:** It represents the functionality of a system by utilizing actors and use cases. It encapsulates the functional requirement of a system and its association with actors. It portrays the use case view of a system.

\*\*\*\*\*

**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 behavior 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.

## 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.

## How to Draw an Interaction Diagram?

As we have already discussed, the purpose of interaction diagrams is to capture the dynamic aspect of a system. So to capture the dynamic aspect, we need to understand what a dynamic aspect is and how it is visualized. Dynamic aspect can be defined as the snapshot of the running system at a particular moment

We have two types of interaction diagrams in UML. One is the sequence diagram and the other is the collaboration diagram. The sequence diagram captures the time sequence of the message flow from one object to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.

Following things are to be identified clearly before drawing the interaction diagram

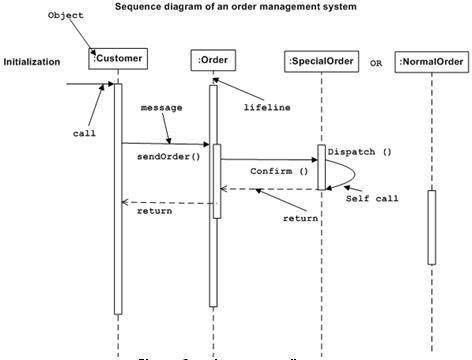
* Objects taking part in the interaction.
* Message flows among the objects.
* The sequence in which the messages are flowing.
* Object organization.

Following are two interaction diagrams modeling the order management system. The first 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.

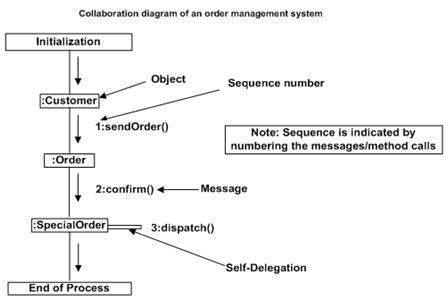
The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, 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 have taken 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.



\*\*\*\*\*

**State Diagrams:**

Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using Statechart diagrams −

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

**How to Draw a Statechart Diagram?**

Statechart diagram is used to describe the states of different objects in its life cycle. Emphasis is placed on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

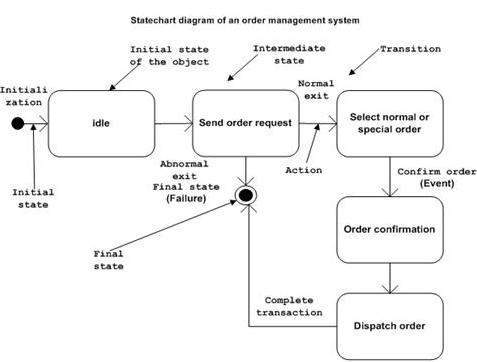
Before drawing a Statechart diagram we should clarify the following points −

* Identify the important objects to be analyzed.
* Identify the states.
* Identify the events.

Following is an example of a Statechart diagram where the state of Order object is analyzed

The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for the state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exits. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as a complete transaction as shown in the following figure. The initial and final state of an object is also shown in the following figure.



\*\*\*\*\*

**Activity Diagrams:**

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

**How to Draw an Activity Diagram?**

Activity diagrams are mainly used as a flowchart that consists of activities performed by the system. Activity diagrams are not exactly flowcharts as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane, etc.

Before drawing an activity diagram, we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities, we need to understand how they are associated with constraints and conditions.

Before drawing an activity diagram, we should identify the following elements

* Activities
* Association
* Conditions
* Constraints

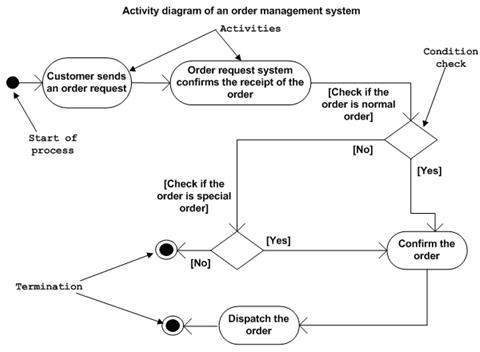
Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

Following is an example of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

* Send order by the customer
* Receipt of the order
* Confirm the order
* Dispatch the order

After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process.



\*\*\*\*\*

**Component and Deployment Diagrams:**

**Component Diagrams:-**

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

**Purpose of Component Diagrams:-**

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

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

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

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

**How to Draw a Component Diagram?**

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries, etc

The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

Initially, the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.

This diagram is very important as without it the application cannot be implemented efficiently. A well-prepared component diagram is also important for other aspects such as application performance, maintenance, etc.

Before drawing a component diagram, the following artifacts are to be identified clearly −

* Files used in the system.
* Libraries and other artifacts relevant to the application.
* Relationships among the artifacts.

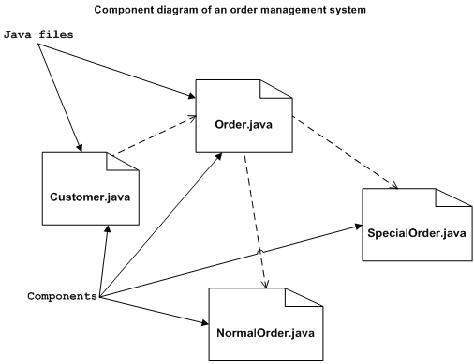
After identifying the artifacts, the following points need to be kept in mind.

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

Following is a component diagram for order management system. Here, the artifacts are files. The diagram shows the files in the application and their relationships. In actual, the component diagram also contains dlls, libraries, folders, etc.

In the following diagram, four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far as it is drawn for completely different purpose.

The following component diagram has been drawn considering all the points mentioned above.



**Deployment Diagrams:-**

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 relationships.

**Purpose of Deployment Diagrams:-**

The term Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.

Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.

UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components.

Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system. Deployment diagrams are used by the system engineers.

The purpose of deployment diagrams can be described as −

* Visualize the hardware topology of a system.
* Describe the hardware components used to deploy software components.
* Describe the runtime processing nodes.

**How to Draw a Deployment Diagram?**

Deployment diagram represents the deployment view of a system. It is related to the component diagram because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardware used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important as it controls the following parameters −

* Performance
* Scalability
* Maintainability
* Portability

Before drawing a deployment diagram, the following artifacts should be identified

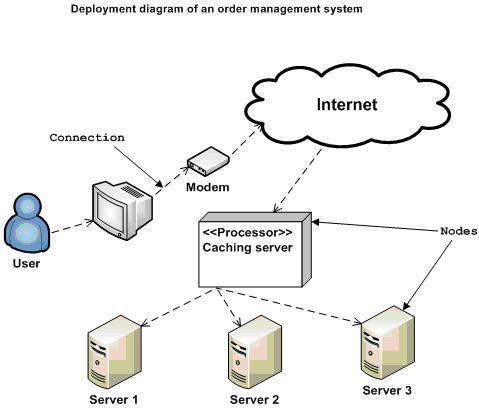
* Nodes
* Relationships among nodes

Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as

* Monitor
* Modem
* Caching server
* Server

The application is assumed to be a web-based application, which is deployed in a clustered environment using server 1, server 2, and server 3. The user connects to the application using the Internet. The control flows from the caching server to the clustered environment.

The following deployment diagram has been drawn considering all the points mentioned above.



\*\*\*\*\*

**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.

**Software Design and Architecture**

**Design Process:**

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

**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.

\*\*\*\*\*

**Principles Leading to Good Design:**

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

**1.Modularity:-**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.

**2.Coupling:-** 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.

**3.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.

**4.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.

**5.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.

**6.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 to

your objectives and priorities.

• Step 3: Determine whether any of the alternatives prevents you from meeting one

or 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 on

a 400Mhz Intel processor, using known cryptanalysis techniques.

• **Maintainability:** No specific objective.

**CPU efficiency**: Must respond to the user within one second when running on a

400 MHz intel processor.

• **Network bandwidth efficiency**: Must not require transmission of more than 8KB

of 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 else

financial benefit to users.

\*\*\*\*\*

**Good Design Document:**

A software requirements document (also known as software requirements specifications) is a document that describes the intended use-case, features, and challenges of a software application.

These documents are created before the project has started development in order to get every stakeholder on the same page regarding the software’s functionality.

Software requirements are written up by the tech team depending on the project they are working on. As non-technical colleagues, clients, and partners get involved it’s important to ensure that everyone is on the same page and agree with the scope, budget, and goal of the project.

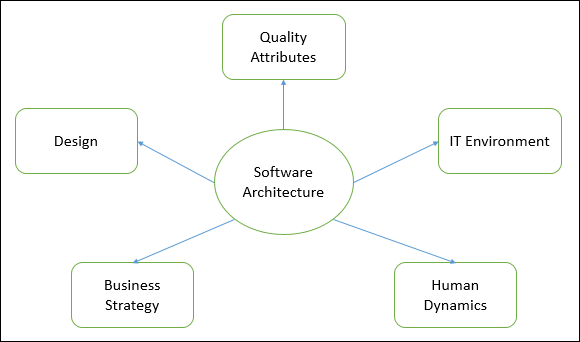
Software requirement documents provide an important map of the product being built, the features that will be included, and much more.

This roadmap helps to keep the technical and non-technical team on the same wavelength as to what the expectations are. It helps to ensure that the product is built meeting the needs whether it’s for internal purposes, for users or clients.

\*\*\*\*\*

**Software Architecture:**

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.

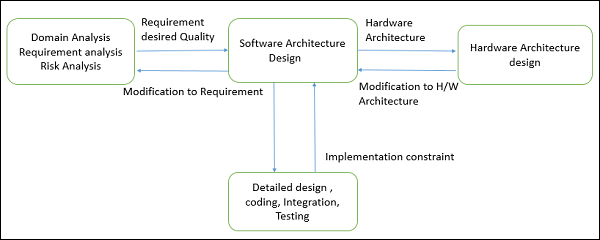
* It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
* Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of −
  + Selection of structural elements and their interfaces by which the system is composed.
  + Behavior as specified in collaborations among those elements.
  + Composition of these structural and behavioral elements into large subsystem.
  + Architectural decisions align with business objectives.
  + 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 −

* To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
* Act as a blueprint during the development process.
* Guide the implementation tasks, including detailed design, coding, integration, and testing.

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 −

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

\*\*\*\*\*

**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. For example: how to separate the UI of the data module in our architectural style? How to integrate a third-party component with our system? how many tires will we have in our client-server architecture? Examples of architectural patterns are microservices, message bus, service requester/ consumer, MVC, MVVM, microkernel, n-tier, domain-driven design, and presentation-abstraction-control.

**Design Patterns:-**

Design patterns are accumulative best practices and experiences that software professionals used over the years to solve the general problem by – trial and error – they faced during software development. The Gang of Four (GOF, refers to Eric Gamma, Richard Helm, Ralf Johnson, and John Vlissides) wrote a book in 1994 titled with “Design Pattern – Elements of reusable object-oriented software” in which they suggested that design patterns are based on two main principles of object-oriented design:

* Develop to an interface, not to an implementation.
* Favor object composition over inheritance.

Also, they presented that the design patterns set contains 23 patterns and categorized into three main sets:

**1.Creationaldesignpatterns:**  
Provide a way to create objects while hiding the creation logic. Thus, the object creation is to be done without instantiating objects directly with the “New” keyword to gives the flexibility to decide which objects need to be created for a given use case. The creational design patterns are:

* **Abstract factory pattern:** provide an interface to create objects without specifying the classes.
* **Singleton pattern:** provide only a single instance of the calls and global access to this instance.
* **Builder Pattern:** Separate the construction from representation and allows the same construction to create multiple representations.
* **Prototype pattern:** creating duplicate without affecting the performance and memory. So the duplicate object is built from the skeleton of an existing object.

**2. Structural patterns:**

Concerned with class and object composition. The Structural design patterns are:

* **Adapter pattern:** it works as a bridge between two incompatible interfaces and compines their capabilities.
* **Bridge pattern:** provide a way to decouple the abstraction from its implementation.
* **Filter pattern:** Also known as criteria pattern, it provides a way to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations.
* **Composite pattern:** provide a way to treat a group of objects in a similar way as a single object. It composes objects in term of a tree structure to represent part as well as a whole hierarchy
* **Decorator pattern:** allows adding new functionality to an existing object without altering its structure.
* **Façade pattern:** provide a unified interface to a set of interfaces.it hides the complexities of the system and provides an interface to the client using which the client can access the system.
* **Flyweight pattern:** reduce the number of objects created and to decrease memory footprint and increase performance. It helps in reusing already existing similar kind objects by storing them and creates a new object when no matching object is found.
* **Proxy pattern:** provides a place holder to another object to control access to it. The object has an original object to interface its functionality to the outer world.

**3. Behavioral patterns:-**

Behavioral patterns are concerned with communications between objects. The following is the list of behavioral patterns:

* **Responsibility pattern:** creates a chain of receiver objects for a request. This pattern decouples the sender and receiver of a request based on the type of request.
* **Command pattern:** it’s a data-driven pattern in which A request is wrapped under an object as command and passed to an invoker object.
* **Interpreter pattern:** provides a way to evaluate language grammar or expression. It involves implementing an expression interface that tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine, etc.
* **Iterator pattern:** provides a way to access the elements of a collection object in a sequential manner without any need to know its underlying representation.
* **Mediator pattern:** used to reduce communication complexity between multiple objects or classes. It provides a mediator class that normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling.
* **Memento pattern:** used to restore the state of an object to a previous state.
* **Observer pattern:** used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.
* **State pattern:** is used to change the class behavior based on its state.
* **Null object pattern:** helps to avoid null references by having a default object.
* **Strategy pattern:** provides a way to change class behavior or its algorithm at run time.
* **Template pattern:** an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.
* **Visitor pattern:**used to change the executing algorithm of an element class.

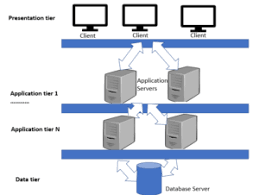
\*\*\*\*\*

**The Multilayer:**

Multi layered software architecture is one of the most popular architectural patterns today. It moderates the increasing complexity of modern applications. It also makes it easier to work in a more agile manner. That’s important when you consider the dominance of DevOps and other similar methodologies today. Sometimes called tiered architecture, or n-tier architecture, a multi layered software architecture consists of various layers, each of which corresponds to a different service or integration. Because each layer is separate, making changes to each layer is easier than having to tackle the entire architecture.

A multi layered software architecture still has the presentation layer and data layer. It simply splits up and expands the application layer. These additional aspects within the application layer are essentially different services. This means your software should now be more scalable and have extra dimensions of functionality. Of course, the distribution of application code and functions among the various tiers will vary from one architectural design to another, but the concept remains the same.

The diagram below illustrates what a multi layered software architecture looks like. As you can see, it’s a little more complex that a three-tiered architecture, but it does increase scalability quite significantly:



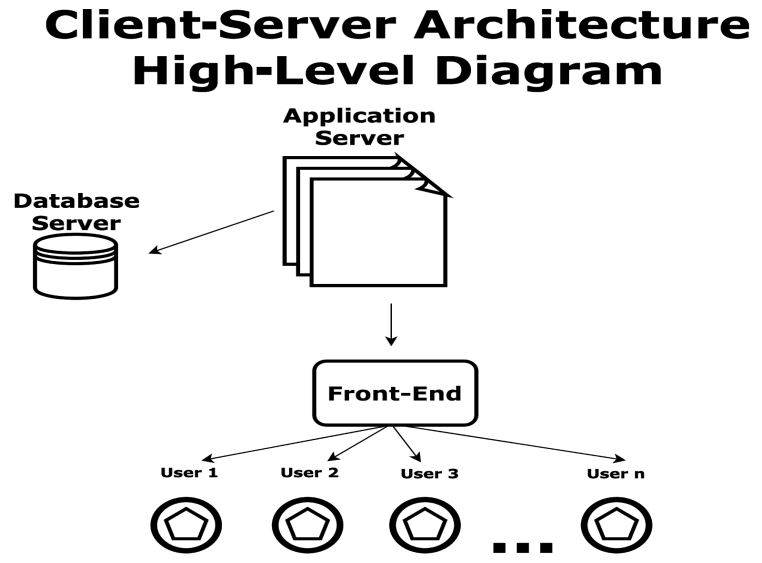
\*\*\*\*\*

**Client-Server:**

Have you ever gone into a restaurant and ordered your food directly from the cook?  Probably not. You sit at the table and wait until the waiter comes to take your order. You tell them what you want to eat, and they go into the kitchen and tell the cook. It makes sense. You don’t necessarily use the right jargon for the cook, and the cook might not know what table to send your food to when it’s done.  The intermediary - the waiter - is effective here. Especially since the most important thing for you is that your order is correct. The cook doesn’t care who you are as long as they get the instructions they need.

The client-server architecture works under the same principle: it partitions tasks between the providers of a service, called **servers,** and service requesters, called **clients.** A server is  like the cook in our restaurant metaphor, and the client is the customer.

Just like a customer, clients usually initiate communication sessions with servers, which await incoming requests. The device (laptop, tablet, smartphone) a client uses to ask for service from the server is like the waiter in our example: it mediates between the service requester and the service provider and knows how to speak with both of them.

****

As you can see in the diagram above, a standard client-server architecture has three parts:

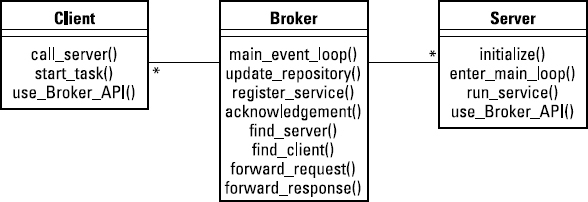
* **Front-End:** This is the piece of software that interacts with users, even if they are on different platforms with different technologies. Any front-end module in a client-server architecture is designed to interact with all existing devices on the market. This level contains the login screens, menus, data screens, and reports that give and take information to/from users. For example, most development tools and frameworks allow the creation of one version of a program that works for PCs, tablets, and phones.
* **Application server:** This is the server where the software modules of the application are installed. It connects to the database (called back-end) and interacts with users (called front-end). The application server is like the waiter on our restaurant example.
* **Database server:** This server contains the tables, indexes, and data managed by the application. Searches and insert/delete/update operations are executed here.

**\*\*\*\*\***

**Broker:**

Structure distributed systems so that the components communicate via remote service invocation. A broker component coordinates communication of requests from client to server and also coordinates returning the results from server to client.

**Looking inside a broker system:-** Three main components are involved in a broker system: the broker, the server, and the client. Figure shows a simple class diagram.

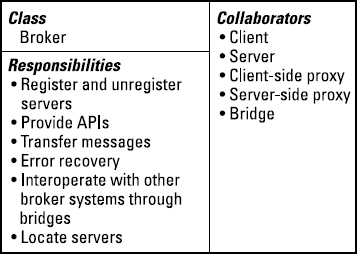


In addition to these components, a broker system features proxies and bridges. I describe all these elements in the following sections.

#### The broker:-

The broker is the message-routing component of your system. It passes messages from

client to server and from server to client. These messages are requests for services and replies to those requests, as well as messages about exceptions that have occurred. The requests are coded as calls to the broker's API. The broker is responsible for error handling in response to these exception reports.



\*\*\*\*\*

**Transaction Processing:**

A process reads a series of inputs one by one.

• Each input describes a transaction- a command that typically some change

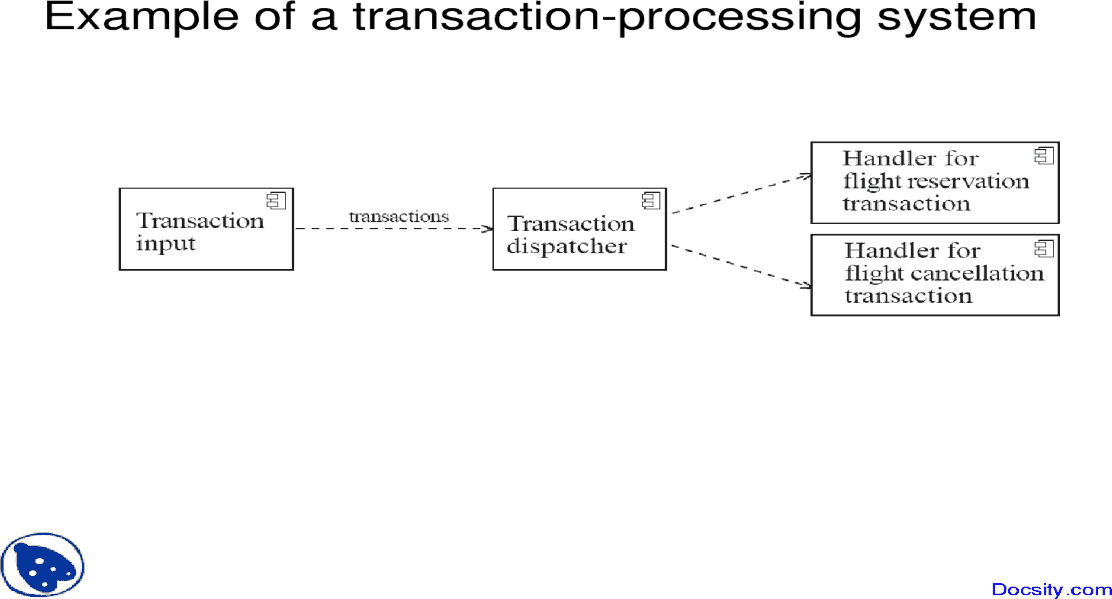
to the data stored by the system.

• There is a transaction dispatcher component that decides what to do with

each transaction.

• This dispatches a procedure call or message to one of a series of component

that will handle the transaction.

****

\*\*\*\*\*

**Pipe & Filter And MVC Architectural Patterns:**

**Pipe & Filter:-**

1. Divide and conquer: The separate processes can be independently designed. Excellent for manufacturing / process control / etc. systems.

2. Increase cohesion: The processes have *functional* *cohesion*. (single input; single output; no side effects…)

3. Reduce coupling: The processes have only *one* input and *one* output.

4. Increase abstraction: The pipeline components are often good abstractions, *hiding* their internal details. And components are usually replacable!

5.Increase reusability: The processes can often be used in many different contexts.

6. Increase reuse: It is often possible to find reusable components to insert into a pipeline.

**MVC Architectural Patterns:-**

Architectural pattern to help separate user interface layer from other parts of the system

Great way to have layered cohesion, as interfaces or controlled.

Coupling reduced between UI layer and rest of system.

Very popular architectural pattern….

THE MVC pattern separates the Model: the functional layer (business entities, ‘key abstractions,’ the objects, relations, ...) from the View: the user interface and the Controller; the director / sequencer of the activities in response to the user.

1.The *model* contains the underlying classes whose instances (objects) are to be viewed and manipulated

* + - Model will likely contain classes from the business domain that may be general and form the application itself (may be inherited) , which may be unique or specialized to the application.
    - These may be very complicated software objects.

2.The *view* contains objects used to render the appearance of the data from the model in the user interface and the controls with which an actor can interact.

3.The *controller* contains the objects that control and handle the user’s interaction with the view and the model.

* + - Controller contains business logic…and response to events.

(The Observable design pattern is normally used to separate the model from the view (later) )

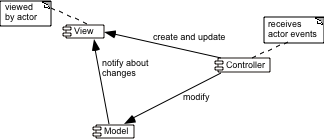
**Example of the MVC architecture for the UI**

* MVC exhibits layered cohesion, as the model has no idea

what view and controller are attached to it (doesn’t care!).

* Model is ‘passive’ in this respect.
* The View (UI), business services (controller), and model

(business entities / core abstractions) will reside in different architectural layers.

****

There may be special cases when no controller component is created, but

the separation of the model from the view is still essential.

**The MVC Architecture and Design Principles**

1. Divide and conquer: Three components can be somewhat independently designed.

2. Increase cohesion: Components have stronger layer cohesion than if the view and controller were together in a single UI layer.

3. Reduce coupling: Minimal communication channels among the three components.

6. Increase reuse: The view and controller normally make *extensive* use of reusable components for various kinds of UI controls.

7. Design for flexibility: It is usually quite easy to change the UI by changing the view, the controller, or both.

10. Design for testability: Can *test* application separately from the UI.

\*\*\*\*\*

**Design Patterns**

**Abstraction-Occurrence:**

Abstract Factory design pattern is one of the Creational pattern. Abstract Factory pattern is almost similar to [Factory Pattern](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/) is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories.

Abstract factory pattern implementation provides us with a framework that allows us to create objects that follow a general pattern. So at runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.

**Let see the GOFs representation of Abstract Factory Pattern :** 

C:\Users\user\Desktop\AbstractFactoryPattern-2.png

UML class diagram example for the Abstract Factory Design Pattern. 

* **AbstractFactory**: Declares an interface for operations that create abstract product objects.
* **ConcreteFactory**: Implements the operations declared in the AbstractFactory to create concrete product objects.
* **Product**: Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.
* **Client**: Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Factory provides interfaces for creating families of related or dependent objects without specifying their concrete classes.Client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the family of objects.

The client does not know or care which concrete objects it gets from each of these concrete factories since it uses only the generic interfaces of their products.So with this idea of Abstract Factory pattern, we will now try to create a design that will facilitate the creation of related objects.

\*\*\*\*\*

**General Hierarchical:**

Hierarchical architecture views the whole system as a hierarchy structure, in which the software system is decomposed into logical modules or subsystems at different levels in the hierarchy. This approach is typically used in designing system software such as network protocols and operating systems.

In system software hierarchy design, a low-level subsystem gives services to its adjacent upper level subsystems, which invoke the methods in the lower level. The lower layer provides more specific functionality such as I/O services, transaction, scheduling, security services, etc. The middle layer provides more domain dependent functions such as business logic and core processing services. And, the upper layer provides more abstract functionality in the form of user interface such as GUIs, shell programming facilities, etc.

It is also used in organization of the class libraries such as .NET class library in namespace hierarchy. All the design types can implement this hierarchical architecture and often combine with other architecture styles.

Hierarchical architectural styles is divided as −

* Main-subroutine
* Master-slave
* Virtual machine

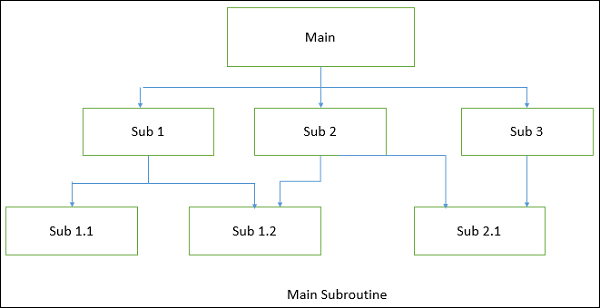
**Main-subroutine:-**

The aim of this style is to reuse the modules and freely develop individual modules or subroutine. In this style, a software system is divided into subroutines by using top-down refinement according to desired functionality of the system.

These refinements lead vertically until the decomposed modules is simple enough to have its exclusive independent responsibility. Functionality may be reused and shared by multiple callers in the upper layers.

There are two ways by which data is passed as parameters to subroutines, namely −

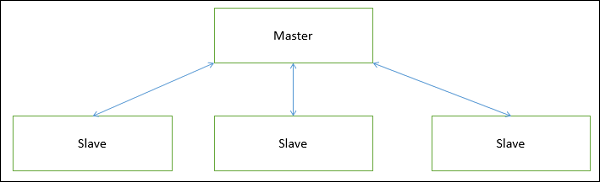
* **Pass by Value** − Subroutines only use the past data, but can’t modify it.
* **Pass by Reference** − Subroutines use as well as change the value of the data referenced by the parameter.



**Master-Slave:-**

This approach applies the 'divide and conquer' principle and supports fault computation and computational accuracy. It is a modification of the main-subroutine architecture that provides reliability of system and fault tolerance.

In this architecture, slaves provide duplicate services to the master, and the master chooses a particular result among slaves by a certain selection strategy. The slaves may perform the same functional task by different algorithms and methods or totally different functionality. It includes parallel computing in which all the slaves can be executed in parallel.



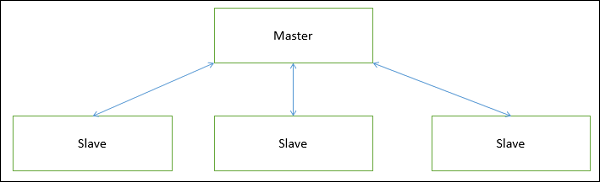
The implementation of the Master-Slave pattern follows five steps −

* Specify how the computation of the task can be divided into a set of equal sub-tasks and identify the sub-services that are needed to process a sub-task.
* Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.
* Define an interface for the sub-service identified in step 1. It will be implemented by the slave and used by the master to delegate the processing of individual sub-tasks.
* Implement the slave components according to the specifications developed in the previous step.
* Implement the master according to the specifications developed in step 1 to 3.

**Master-Slave:-**

This approach applies the 'divide and conquer' principle and supports fault computation and computational accuracy. It is a modification of the main-subroutine architecture that provides reliability of system and fault tolerance.

In this architecture, slaves provide duplicate services to the master, and the master chooses a particular result among slaves by a certain selection strategy. The slaves may perform the same functional task by different algorithms and methods or totally different functionality. It includes parallel computing in which all the slaves can be executed in parallel.



The implementation of the Master-Slave pattern follows five steps −

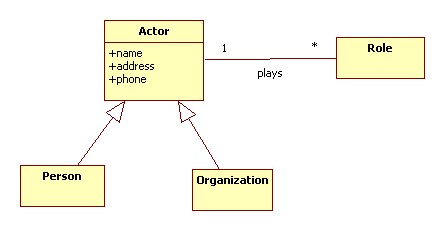
* Specify how the computation of the task can be divided into a set of equal sub-tasks and identify the sub-services that are needed to process a sub-task.
* Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.
* Define an interface for the sub-service identified in step 1. It will be implemented by the slave and used by the master to delegate the processing of individual sub-tasks.
* Implement the slave components according to the specifications developed in the previous step.
* Implement the master according to the specifications developed in step 1 to 3.

\*\*\*\*\*

**Play-Role:**

Most people don't confuse actors from the roles they play. Keanu Reeves is an actor. Ted, Neo, and Buddha are some of the roles he has played. In modeling it's a good idea to decouple actors from the roles they play. In this way an actor can quickly shift from one role to another. The actor object is also a good place to store personal information like name and address.

Here's the basic pattern:



Examples of roles include:

Employee, Customer, Supplier, Nurse, Doctor, Patient, Teacher, Student, Commander

\*\*\*\*\*

**Singleton:**

Singleton Pattern says that just"define a class that has only one instance and provides a global point of access to it".

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

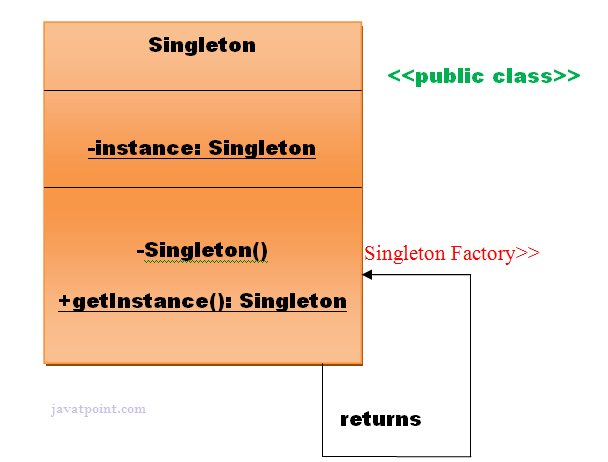
#### Advantage of Singleton design pattern:-

* Saves memory because object is not created at each request. Only single instance is reused again and again.

#### Usage of Singleton design pattern:-

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

#### Uml of Singleton design pattern:-



To create the singleton class, we need to have static member of class, private constructor and static factory method.

* **Static member:** It gets memory only once because of static, itcontains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

\*\*\*\*\*

**Observer:**

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

**Implementation:-**

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class *Observer* and a concrete class Subject that is extending class *Observer*.

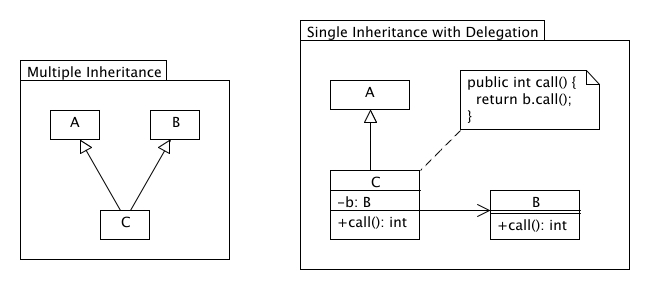
ObserverPatternDemo, our demo class, will use *Subject* and concrete class object to show observer pattern in action.



\*\*\*\*\*

**Delegation:**

"Delegation is like inheritance done manually through object composition."

****

It is a technique where an object expresses certain behavior to the outside but in reality delegates responsibility for implementing that behaviour to an associated object. This sounds at first very similar to the proxy pattern, but it serves a much different purpose. Delegation is an abstraction mechanism which centralizes object (method) behaviour.

The diagram above shows how multiple-inheritance behavior can be accomplished in Java for actual classes. Of course delegation is not limited to scenarios where multiple-inheritance has to be avoided (as Java does not support this), but can be seen in general as an alternative to inheritance. The same functionality can also be accomplished with interfaces, however sometimes the relationship you want between two classes is actually one of containment rather than inheritance.

\*\*\*\*\*

**Adaptor:**

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

We are demonstrating use of Adapter pattern via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.

**Implementation:-** We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default.

We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files.

We want to make AudioPlayer to play other formats as well. To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format.

AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format. AdapterPatternDemo, our demo class will use AudioPlayer class to play various formats.



\*\*\*\*\*

**Facade:**

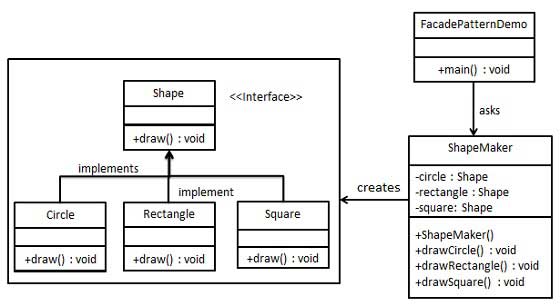
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

**Implementation:-**

We are going to create a Shape interface and concrete classes implementing the Shape interface. A facade class ShapeMaker is defined as a next step.

ShapeMaker class uses the concrete classes to delegate user calls to these classes. FacadePatternDemo, our demo class, will use ShapeMaker class to show the results.



\*\*\*\*\*

**Immutable:**

**Context**: — An immutable object is an object that has a state that never changes after creation

• **Problem**: — How do you create a class whose instances are immutable?

• **Forces**: — There must be no loopholes that would allow ‘illegal’ modification

of an immutable object

• **Solution**: — Ensure that the constructor of the immutable class is the only place

where the values of instance variables are set or modified.

— Instance methods which access properties must not have side

effects.

— If a method that would otherwise modify an instance variable is

required, then it has to return a new instance of the class.

\*\*\*\*\*

**Read-Only Interface and Proxy Patterns:**

**Read-Only Interface:-**

**Context**: — You sometimes want certain privileged classes to be able to modify

attributes of objects that are otherwise immutable

• **Problem**: — How do you create a situation where some classes see a class as

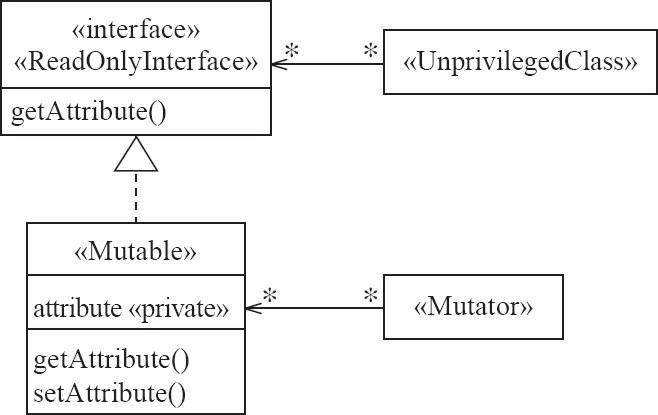
read-only whereas others are able to make modifications?

• **Forces**: — Restricting access by using the **public**, **protected** and **private**

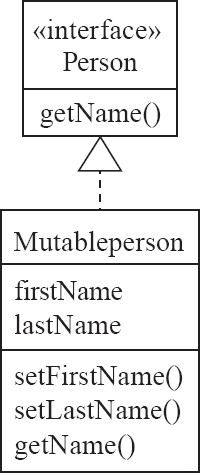
keywords is not adequately selective.

— Making access **public** makes it public for both reading and writing

• **Solution:**

******

**Example:**

****

**Proxy Patterns:-**

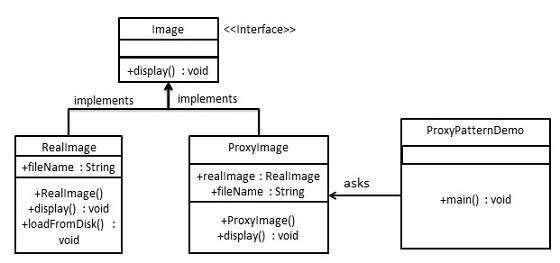
In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.

In proxy pattern, we create object having original object to interface its functionality to outer world.

**Implementation:-**

We are going to create an Image interface and concrete classes implementing the Image interface. ProxyImage is a a proxy class to reduce memory footprint of RealImage object loading.

ProxyPatternDemo, our demo class, will use ProxyImage to get an Image object to load and display as it needs.



\*\*\*\*\*

**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.

**Software Testing**

**Effective and Efficient Testing:**

**Effective Testing:-**

**Test Effectiveness** can be defined as how effectively testing is done or goal is achieved that meets the customer requirement. In SDLC (Software development Life Cycle), we have requirements gathering phase where SRS (Software Requirements Specification) and FRD (Functional Requirements Document) are prepared and based on that development team starts building the software application, at the same time test cases are carved out of SRS and FRD documents by the testing team. Test effectiveness starts right at the beginning of the development and execution of test cases and after development is completed to count the number of defects. Defects can be valid or invalid. Valid defects are required to be fixed in the application or product and invalid ones need to be closed or ignored. Thus, mathematically it is calculated as a percentage of a number of valid defects fixed in software application divided by the sum of a total number of defects injected and a total number of defects escaped.

**Efficient Testing:-**

**Test Efficiency** can be defined as the most efficient way in which the resources associated with the software project are utilized to achieve an organization goal (for example, number of projects to be completed in that particular year). Test efficiency is an internal process for an organization that evaluates the utilization of resources which could be a timeline, hardware, manpower, expertise of test team, etc. to develop a software product. Mathematically test efficiency is calculated as a percentage of the number of alpha testing (in-house or on-site) defects divided by sum of a number of alpha testing and a number of beta testing (off-site) defects. Alpha testing is the testing done by the expertise project test team on-site and this test team is expected to test the product thoroughly before the product is available to the customer or end user in the market. Once internal alpha testing is completed the product is made available to end users to test and look for the defects and provide their valuable feedback. Ideally, there should not be any defect observed by the end user during beta testing as any valid bug identified during beta testing, it will directly deteriorate the test efficiency of the on-site project team.

\*\*\*\*\*

**Defects in Ordinary Algorithms:**

**Defects** are defined as the deviation of the actual and expected result of system or software application. Defects can also be defined as any deviation or irregularity from the specifications mentioned in the product functional specification document. Defects are caused by the developer in development phase of software. When a developer or programmer during the development phase makes some mistake then that turns into bugs that are called defects. It is basically caused by the developers’ mistakes.

Defect in a software product represents the inability and inefficiency of the software to meet the specified requirements and criteria and subsequently prevent the software application to perform the expected and desired working.

**Types of Defects:**  
Following are some of the basic types of defects in the software development:

1. **Arithmetic Defects:-** It include the defects made by the developer in some arithmetic expression or mistake in finding solution of such arithmetic expression. This type of defects are basically made by the programmer due to access work or less knowledge. Code congestion may also lead to the arithmetic defects as programmer is unable to properly watch the written code.
2. **Logical Defects:-** Logical defects are mistakes done regarding the implementation of the code. When the programmer doesn’t understand the problem clearly or thinks in a wrong way then such types of defects happen. Also while implementing the code if the programmer doesn’t take care of the corner cases then logical defects happen. It is basically related to the core of the software.
3. **Syntax Defects:**- Syntax defects means mistake in the writing style of the code. It also focuses on the small mistake made by developer while writing the code. Often the developers do the syntax defects as there might be some small symbols escaped. For example, while writing a code in C++ there is possibility that a semicolon(;) is escaped.
4. **Multithreading Defects:-** Multithreading means running or executing the multiple tasks at the same time. Hence in multithreading process there is possibility of the complex debugging. In multithreading processes sometimes there is condition of the deadlock and the starvation is created that may lead to system’s failure.
5. **Interface Defects:**- Interface defects means the defects in the interaction of the software and the users. The system may suffer different kinds of the interface testing in the forms of the complicated interface, unclear interface or the platform based interface.
6. **Performance Defects:**- Performance defects are the defects when the system or the software application is unable to meet the desired and the expected results. When the system or the software application doesn’t fulfill the users’s requirements then that is the performance defects. It also includes the response of the system with the varying load on the system.

\*\*\*\*\*

**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.

\*\*\*\*\*

**Timing and Co-ordination:**

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.

Livelock is similar to deadlock, in the sense that the system is stuck in a particular behaviour that it cannot get out 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.

**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 anticipate timing and co-ordination defects. If its cost is justifiable, glass-box testing is one of the best ways to uncover these defects,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 particularly hard to test for critical racesusing black-boxtestingalone,since you often do not know the extent of the concurrency going on inside the system, and you cannot always manipulate the variousthreadsto cause race conditions.

\*\*\*\*\*

**Stress and Unusual Situations:**

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 x configurations of hardware, operating systems and external libraries, but fail if it is run using other configurations.

**Testing Strategies**: Extensively execute the system with a wide variety of configurations That might be encountered 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 data structure. No matter what the cause, the program being tested should report the problem in such a way that the user will understand.It should the neither 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 to

the 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 uses Many resources,relinquishes them and then uses them again repeatedly.

• **Defects in the process of recovering from acrash**

**Defect**:Any system will under go 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.

\*\*\*\*\*

**Testing Strategies for Large Systems:**

**Integration testing: big bang versus incremental approaches**

1.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.

2.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.

3.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 find the defect more easily.

4.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 subApplication 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**

1.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.

2.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**

1.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.

2.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. Drivers in bottom-up testing have a similar role to stubs in top-down testing, and are time-consuming to write.

**Sandwich testing**

1.Sandwich testing is a hybrid between bottom-up and top-down testing – it is sometimes therefore called mixed testing.A typical approach to sandwich testing is to test the user interface in isolation, using stubs, and also to test the very lowest- level

functions, using drivers.

2.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**

1.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.

2.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.

1.You can minimize the ripple effect by applying a very disciplined approach to the process of fixing defects.

2.However, experience shows that the ripple effect will still persist if you do not have an appropriate testing strategy.

3.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.**

4.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 much

of the system as possible.

**Deciding when to stop testing**

1.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.

2.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 be considered complete:

3.All of the level 1 test cases must have been successfully executed.

4.Certain predefined percentages of level 2 and level 3 test cases must have been executed successfully.

4.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**

1.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.

2.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**

**1.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.

**2.Beta testing** is testing performed by the user or client in their normal work environment. It can 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.

**3.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 in order to choose between several competing generic products

\*\*\*\*\*

**Software Project Management**

**Introduction to Software Project Management:**

The job pattern of an IT company engaged in software development can be seen split in two parts:

* Software Creation
* Software Project Management

A project is well-defined task, which is a collection of several operations done in order to achieve a goal (for example, software development and delivery). A Project can be characterized as:

* Every project may has a unique and distinct goal.
* Project is not routine activity or day-to-day operations.
* Project comes with a start time and end time.
* Project ends when its goal is achieved hence it is a temporary phase in the lifetime of an organization.
* Project needs adequate resources in terms of time, manpower, finance, material and knowledge-bank.

**Software Project:-**

A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.

**Need of software project management:-**

Software is said to be an intangible product. Software development is a kind of all new stream in world business and there’s very little experience in building software products. Most software products are tailor made to fit client’s requirements. The most important is that the underlying technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one. All such business and environmental constraints bring risk in software development hence it is essential to manage software projects efficiently.



The image above shows triple constraints for software projects. It is an essential part of software organization to deliver quality product, keeping the cost within client’s budget constrain and deliver the project as per scheduled. There are several factors, both internal and external, which may impact this triple constrain triangle. Any of three factor can severely impact the other two.

Therefore, software project management is essential to incorporate user requirements along with budget and time constraints.

\*\*\*\*\*

**Activities of Software Project Management:**

Software project management comprises of a number of activities, which contains planning of project, deciding scope of software product, estimation of cost in various terms, scheduling of tasks and events, and resource management. Project management activities may include:

1.Project Planning

2.Scope Management

3.Project Estimation

**1.Project Planning:-**

Software project planning is task, which is performed before the production of software actually starts. It is there for the software production but involves no concrete activity that has any direction connection with software production; rather it is a set of multiple processes, which facilitates software production. Project planning may include the following:

**2.Scope Management:-**

It defines the scope of project; this includes all the activities, process need to be done in order to make a deliverable software product. Scope management is essential because it creates boundaries of the project by clearly defining what would be done in the project and what would not be done. This makes project to contain limited and quantifiable tasks, which can easily be documented and in turn avoids cost and time overrun.

During Project Scope management, it is necessary to -

* Define the scope
* Decide its verification and control
* Divide the project into various smaller parts for ease of management.
* Verify the scope
* Control the scope by incorporating changes to the scope

**3.Project Estimation:-**

For an effective management accurate estimation of various measures is a must. With correct estimation managers can manage and control the project more efficiently and effectively.

Project estimation may involve the following:

* **Software size estimation**:- Software size may be estimated either in terms of KLOC (Kilo Line of Code) or by calculating number of function points in the software. Lines of code depend upon coding practices and Function points vary according to the user or software requirement.
* **Effort estimation**:- The managers estimate efforts in terms of personnel requirement and man-hour required to produce the software. For effort estimation software size should be known. This can either be derived by managers’ experience, organization’s historical data or software size can be converted into efforts by using some standard formulae.
* **Time estimation**:- Once size and efforts are estimated, the time required to produce the software can be estimated. Efforts required is segregated into sub categories as per the requirement specifications and interdependency of various components of software. Software tasks are divided into smaller tasks, activities or events by Work Breakthrough Structure (WBS). The tasks are scheduled on day-to-day basis or in calendar months.

The sum of time required to complete all tasks in hours or days is the total time invested to complete the project.

* **Cost estimation**:- This might be considered as the most difficult of all because it depends on more elements than any of the previous ones. For estimating project cost, it is required to consider -
  + Size of software
  + Software quality
  + Hardware
  + Additional software or tools, licenses etc.
  + Skilled personnel with task-specific skills
  + Travel involved
  + Communication
  + Training and support

\*\*\*\*\*

**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.

1.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. In general, the egoless approach is most suited 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 knowledge vary widely.

2.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.

3.The **‘chief programmer team’**is a model that is mid way between egoless and hierarchical.Itworksverymuchlikethesurgicalteaminanoperatingroom.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 sub system 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.Doublingthesizeofateamwillnothalvethedevelopmenttime.Makingsubsystems,

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:

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

2.**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.

3.**Configuration management and build specialist**. This person ensures that, as changes are made,no new problems are introduced. Everyone 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.

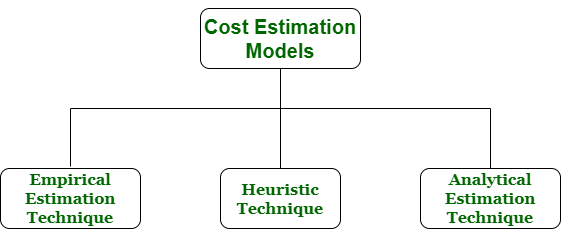
4. **User interface specialist**. Although everybody should interact with users, this Person has the particular responsibility to make sure that usability is kept at the forefront of the design process.

\*\*\*\*\*

**Software Cost Estimation:**

**Cost estimation** simply means a technique that is used to find out the cost estimates. The cost estimate is the financial spend that is done on the efforts to develop and test software in [Software Engineering](https://www.geeksforgeeks.org/software-engineering/). Cost estimation models are some mathematical algorithms or parametric equations that are used to estimate the cost of a product or a project.

Various techniques or models are available for cost estimation, also known as Cost Estimation Models as shown below :



1. **Empirical Estimation Technique:-** Empirical estimation is a technique or model in which empirically derived formulas are used for predicting the data that are a required and essential part of the software project planning step. These techniques are usually based on the data that is collected previously from a project and also based on some guesses, prior experience with the development of similar types of projects, and assumptions. It uses the size of the software to estimate the effort.

In this technique, an educated guess of project parameters is made. Hence, these models are based on common sense. However, as there are many activities involved in empirical estimation techniques, this technique is formalized. For example Delphi technique and Expert Judgement technique.

1. **Heuristic Technique :-** Heuristic word is derived from a Greek word that means “to discover”. The heuristic technique is a technique or model that is used for solving problems, learning, or discovery in the practical methods which are used for achieving immediate goals. These techniques are flexible and simple for taking quick decisions through shortcuts and good enough calculations, most probably when working with complex data. But the decisions that are made using this technique are necessary to be optimal.

In this technique, the relationship among different project parameters is expressed using mathematical equations. The popular heuristic technique is given by [Constructive Cost Model (COCOMO)](https://www.geeksforgeeks.org/software-engineering-cocomo-model/). This technique is also used to increase or speed up the analysis and investment decisions.

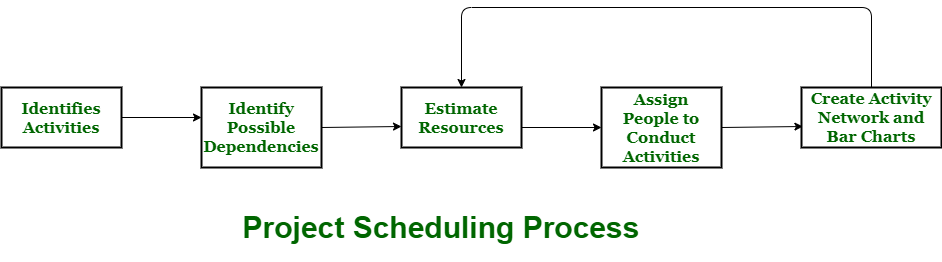
1. **Analytical Estimation Technique :-** Analytical estimation is a type of technique that is used to measure work. In this technique, firstly the task is divided or broken down into its basic component operations or elements for analyzing. Second, if the standard time is available from some other source, then these sources are applied to each element or component of work.

Third, if there is no such time available, then the work is estimated based on the experience of the work. In this technique, results are derived by making certain basic assumptions about the project. Hence, the analytical estimation technique has some scientific basis. [Halstead’s](https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/) software science is based on an analytical estimation model.

\*\*\*\*\*

**Project Scheduling:**

**Project schedule** simply means a mechanism that is used to communicate and know about that tasks are needed and has to be done or performed and which organizational resources will be given or allocated to these tasks and in what time duration or time frame work is needed to be performed. Effective project scheduling leads to success of project, reduced cost, and increased customer satisfaction. Scheduling in project management means to list out activities, deliverables, and milestones within a project that are delivered. It contains more notes than your average weekly planner notes. The most common and important form of project schedule is Gantt chart.

****

**Process:-** The manager needs to estimate time and resources of project while scheduling project. All activities in project must be arranged in a coherent sequence that means activities should be arranged in a logical and well-organized manner for easy to understand. Initial estimates of project can be made optimistically which means estimates can be made when all favorable things will happen and no threats or problems take place.

The total work is separated or divided into various small activities or tasks during project schedule. Then, Project manager will decide time required for each activity or task to get completed. Even some activities are conducted and performed in parallel for efficient performance. The project manager should be aware of fact that each stage of project is not problem-free.

**Problems arise during Project Development Stage:-**

* People may leave or remain absent during particular stage of development.
* Hardware may get failed while performing.
* Software resource that is required may not be available at present, etc.

**Advantages of Project Scheduling:-**  
There are several advantages provided by project schedule in our project management:

* It simply ensures that everyone remains on same page as far as tasks get completed, dependencies, and deadlines.
* It helps in identifying issues early and concerns such as lack or unavailability of resources.
* It also helps to identify relationships and to monitor process.
* It provides effective budget management and risk mitigation.

\*\*\*\*\*

**Tracking and Monitoring:**

**Tracking:-**

Defect tracking is an important process in software engineering as Complex and business critical systems have hundreds of defects. One of the challenging factors is Managing, evaluating and prioritizing these defects. The number of defects gets multiplied over a period of time and to effectively manage them, defect tracking system is used to make the job easier.

**Examples -** Hp Quality Center, IBM Rational Quality Manager

**Defect Tracking Parameters:-**

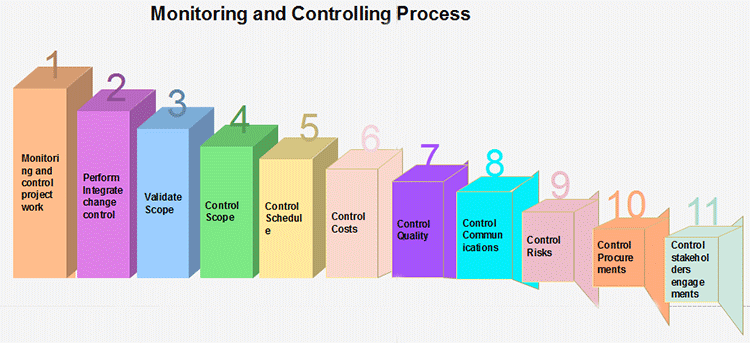
Defects are tracked based on various parameters such as:

* Defect Id
* Priority
* Severity
* Created by
* Created Date
* Assigned to
* Resolved Date
* Resolved By
* Status

**Monitoring:-**

Monitoring and Controlling are processes needed to track, review, and regulate the progress and performance of the project. It also identifies any areas where changes to the project management method are required and initiates the required changes.

The Monitoring & Controlling process group includes eleven processes, which are:



1. **Monitor and control project work:** The generic step under which all other monitoring and controlling activities fall under.
2. **Perform integrated change control:** The functions involved in making changes to the project plan. When changes to the schedule, cost, or any other area of the project management plan are necessary, the program is changed and re-approved by the project sponsor.
3. **Validate scope:** The activities involved with gaining approval of the project's deliverables.
4. **Control scope:** Ensuring that the scope of the project does not change and that unauthorized activities are not performed as part of the plan (scope creep).
5. **Control schedule:** The functions involved with ensuring the project work is performed according to the schedule, and that project deadlines are met.
6. **Control costs:** The tasks involved with ensuring the project costs stay within the approved budget.
7. **Control quality:** Ensuring that the quality of the project?s deliverables is to the standard defined in the project management plan.
8. **Control communications:** Providing for the communication needs of each project stakeholder.
9. **Control Risks:** Safeguarding the project from unexpected events that negatively impact the project's budget, schedule, stakeholder needs, or any other project success criteria.
10. **Control procurements:** Ensuring the project's subcontractors and vendors meet the project goals.
11. **Control stakeholder engagement:** The tasks involved with ensuring that all of the project's stakeholders are left satisfied with the project work.

\*\*\*\*\*

**Software Process Models**

**Waterfall Model:**

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.

The waterfall Model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

**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.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model". In this model, phases do not overlap.

**Waterfall Model – Application:-**Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors. Some situations where the use of Waterfall model is most appropriate are −

* 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.

**Waterfall Model – Advantages:-**

The advantages of waterfall development are that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.

Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.

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 specific deliverables 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.

**Waterfall Model – Disadvantages:-**

The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

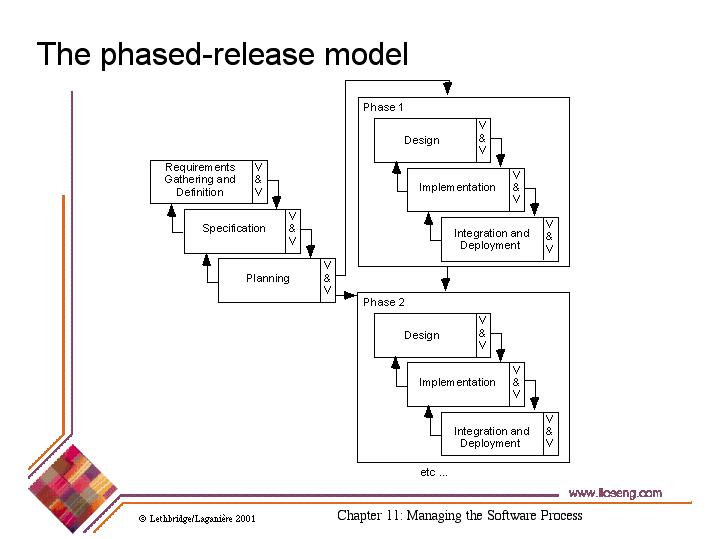
The major disadvantages of the Waterfall Model are as follows −

* 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 of changing. 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.
* Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

\*\*\*\*\*

**The Phased Released Model:**

* After requirements gathering and planning, the project should be broken into separate 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 waterfall approach.
* However, it continues to suggest that all requirements be finalized at the start of development.

****

\*\*\*\*\*

**The Spiral Model:**

The spiral model combines the idea of iterative development with the systematic, controlled aspects of the waterfall model. This Spiral model is a combination of iterative development process model and sequential linear development model i.e. the waterfall model with a very high emphasis on risk analysis. It allows incremental releases of the product or incremental refinement through each iteration around the spiral.

**Spiral Model – Design:-**

The spiral model has four phases. A software project repeatedly passes through these phases in iterations called Spirals.

**Identification:-**This phase starts with gathering the business requirements in the baseline spiral. In the subsequent spirals as the product matures, identification of system requirements, subsystem requirements and unit requirements are all done in this phase.

This phase also includes understanding the system requirements by continuous communication between the customer and the system analyst. At the end of the spiral, the product is deployed in the identified market.

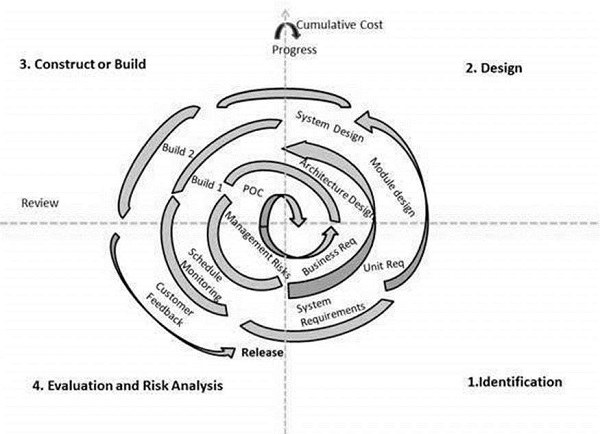
**Design:-** The Design phase starts with the conceptual design in the baseline spiral and involves architectural design, logical design of modules, physical product design and the final design in the subsequent spirals.

**Construct or Build:-** The Construct phase refers to production of the actual software product at every spiral. In the baseline spiral, when the product is just thought of and the design is being developed a POC (Proof of Concept) is developed in this phase to get customer feedback.

Then in the subsequent spirals with higher clarity on requirements and design details a working model of the software called build is produced with a version number. These builds are sent to the customer for feedback.

**Evaluation and Risk Analysis:-** Risk Analysis includes identifying, estimating and monitoring the technical feasibility and management risks, such as schedule slippage and cost overrun. After testing the build, at the end of first iteration, the customer evaluates the software and provides feedback.

The following illustration is a representation of the Spiral Model, listing the activities in each phase.



Based on the customer evaluation, the software development process enters the next iteration and subsequently follows the linear approach to implement the feedback suggested by the customer. The process of iterations along the spiral continues throughout the life of the software.

**Spiral Model Application:-**

The Spiral Model is widely used in the software industry as it is in sync with the natural development process of any product, i.e. learning with maturity which involves minimum risk for the customer as well as the development firms.

The following pointers explain the typical uses of a Spiral Model −

* When there is a budget constraint and risk evaluation is important.
* For medium to high-risk projects.
* Long-term project commitment because of potential changes to economic priorities as the requirements change with time.
* Customer is not sure of their requirements which is usually the case.
* Requirements are complex and need evaluation to get clarity.
* New product line which should be released in phases to get enough customer feedback.
* Significant changes are expected in the product during the development cycle.

**Spiral Model - Pros and Cons:-**

The advantage of spiral lifecycle model is that it allows elements of the product to be added in, when they become available or known. This assures that there is no conflict with previous requirements and design.

This method is consistent with approaches that have multiple software builds and releases which allows making an orderly transition to a maintenance activity. Another positive aspect of this method is that the spiral model forces an early user involvement in the system development effort.

On the other side, it takes a very strict management to complete such products and there is a risk of running the spiral in an indefinite loop. So, the discipline of change and the extent of taking change requests is very important to develop and deploy the product successfully.

The advantages of the Spiral SDLC Model are as follows −

* Changing requirements can be accommodated.
* Allows extensive use of prototypes.
* Requirements can be captured more accurately.
* Users see the system early.
* Development can be divided into smaller parts and the risky parts can be developed earlier which helps in better risk management.

The disadvantages of the Spiral SDLC Model are as follows −

* Management is more complex.
* End of the project may not be known early.
* Not suitable for small or low risk projects and could be expensive for small projects.
* Process is complex
* Spiral may go on indefinitely.
* Large number of intermediate stages requires excessive documentation.

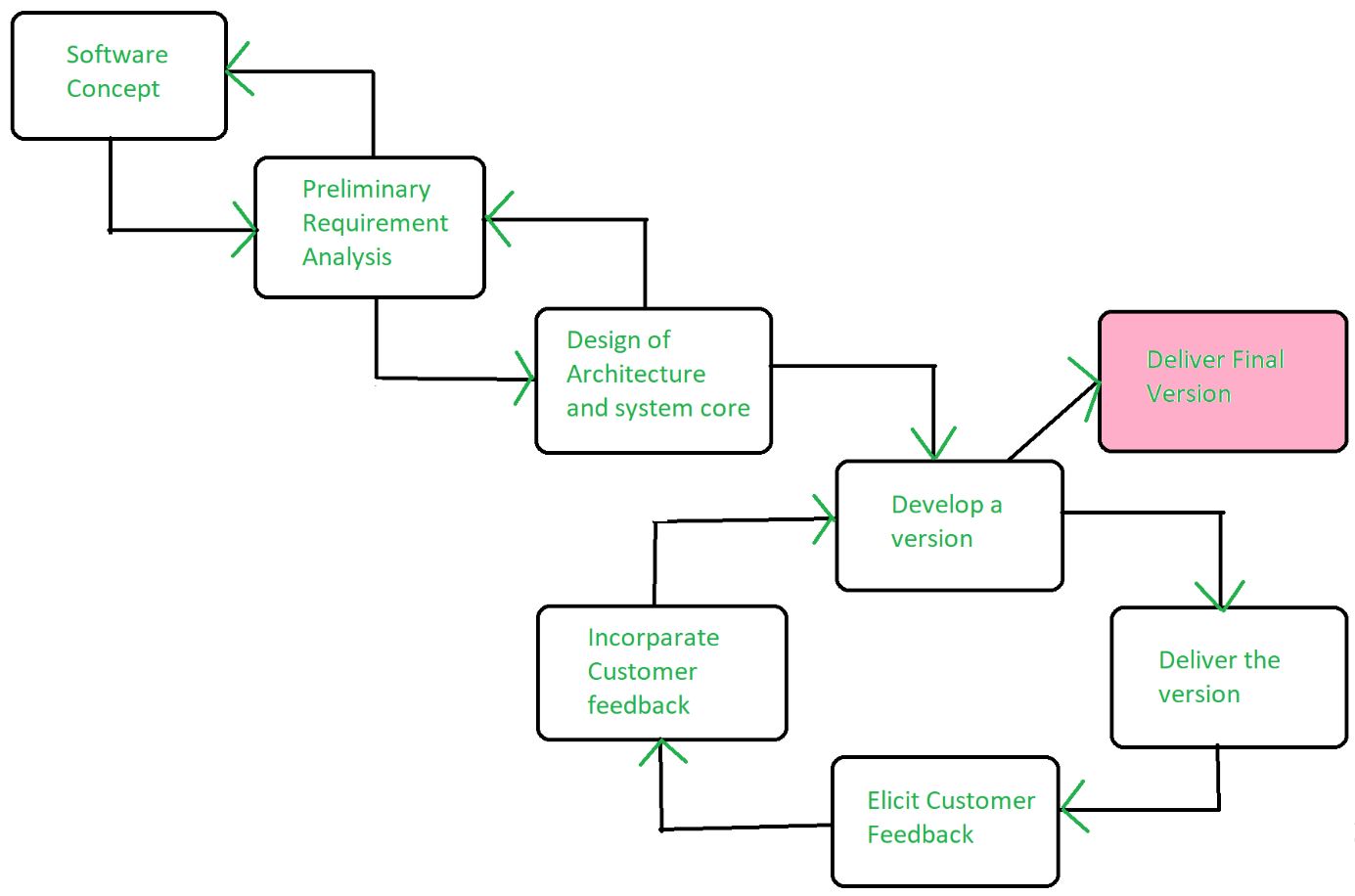
\*\*\*\*\*

**Evolutionary Model:**

**Evolutionary model** is a combination of [Iterative](https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/)and [Incremental model](https://www.geeksforgeeks.org/software-engineering-incremental-process-model/) of software development life cycle. Delivering your system in a big bang release, delivering it in incremental process over time is the action done in this model. Some initial requirements and architecture envisioning need to be done.

It 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 waterfall models 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 and the 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.



Evolutionary model suggests breaking down of work into smaller chunks, prioritizing them and then delivering those chunks to the customer one by one. The number of chunks is huge and is the number of deliveries made to the customer. The main advantage is that the customer’s confidence increases as he constantly gets quantifiable goods or services from the beginning of the project to verify and validate his requirements. The model allows for changing requirements as well as all work in broken down into maintainable work chunks.

**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 to start using the core features instead of waiting for the full software.
2. Evolutionary model is also used in object oriented software development because the system 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.

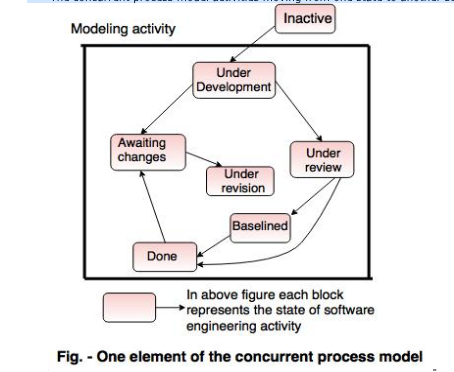
**Disadvantages:**

* 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.

\*\*\*\*\*

**The Concurrent Engineering Model:**

* The concurrent development model is called as concurrent model.
* The communication activity has completed in the first iteration and exits in the awaiting changes state.
* The modeling activity completed its initial communication and then go to the underdevelopment state.
* If the customer specifies the change in the requirement, then the modeling activity moves from the under development state into the awaiting change state.
* The concurrent process model activities moving from one state to another state

.

**Advantages of the concurrent development model**

* This model is applicable to all types of software development processes.
* It is easy for understanding and use.
* It gives immediate feedback from testing.
* It provides an accurate picture of the current state of a project.

**Disadvantages of the concurrent development model**

* It needs better communication between the team members. This may not be achieved all the time.
* It requires to remember the status of the different activities.

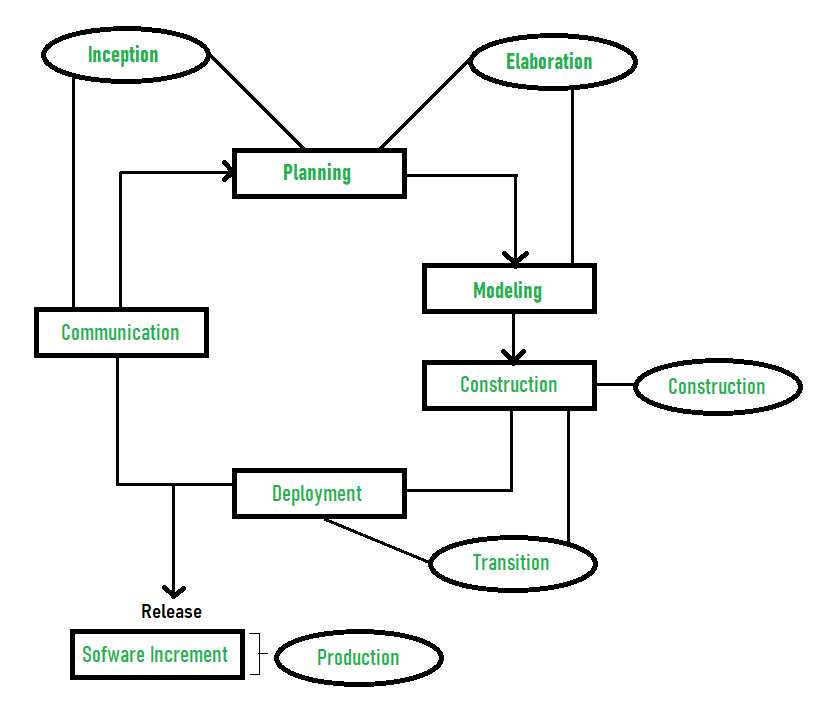
\*\*\*\*\*

**Rational Unified Process:**

**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 to estimate costs and time required.
   * Customers requirements are identified and then it becomes easy to make a plan of 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 these criteria 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 criteria then 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.

\*\*\*\*\*

THE-END