SE Notes UNIT 3 - software engineering

Software Engineering (Dr. A.P.J. Abdul Kalam Technical University)

UNIT 3

Software Design is the process to transform the user requirements into some suitable form, which helps the programmer in software coding and implementation. During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. Hence the aim of this phase is to transform the SRS document into the design document.

The following items are designed and documented during the design phase:

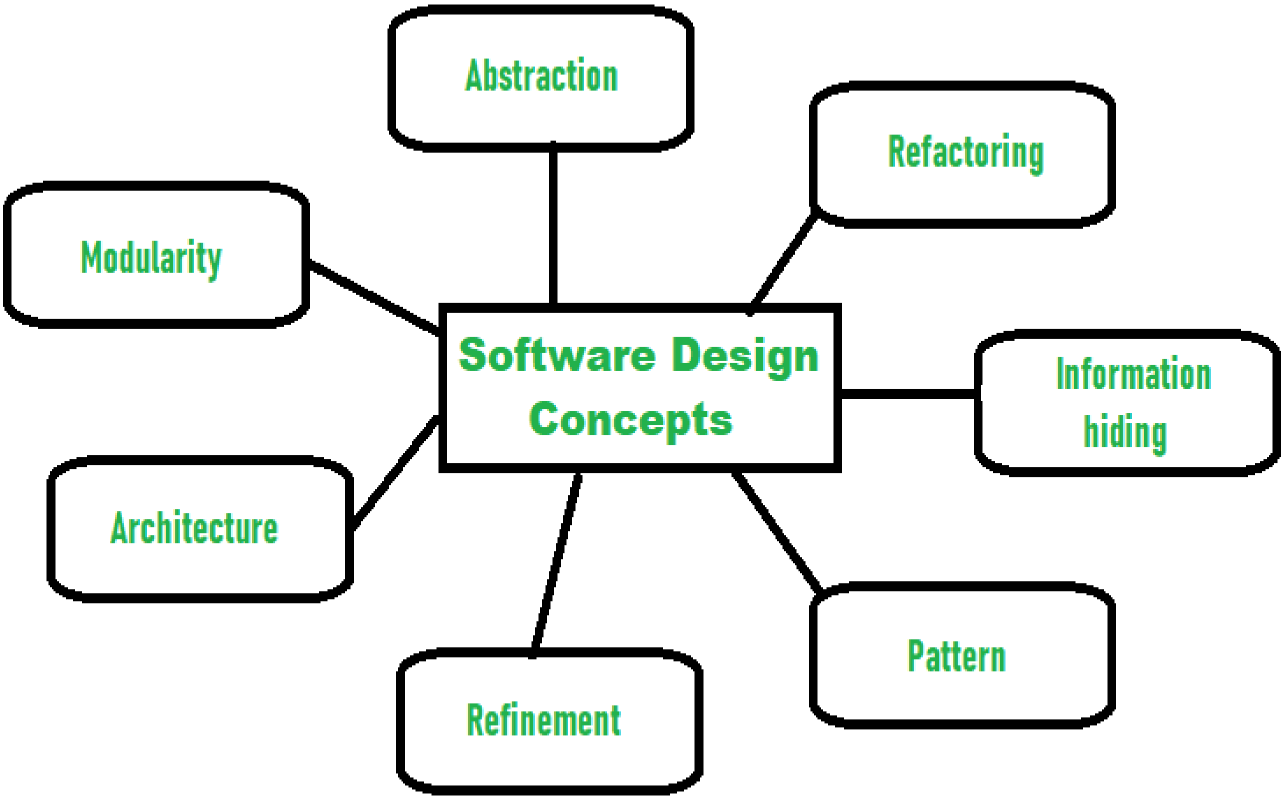
- Different modules required.
- Control relationships among modules.
- Interface among different modules.
- Data structure among the different modules.
- Algorithms required to implement among the individual modules.

Objectives of Software Design:

1. **Correctness:**
   A good design should be correct i.e. it should correctly implement all the functionalities of the system.
2. **Efficiency:**
   A good software design should address the resources, time, and cost optimization issues.
3. **Flexibility:**
   A good software design should have the ability to adapt and accommodate changes easily. It includes designing the software in a way, that allows for modifications, enhancements, and scalability without requiring significant rework or causing major disruptions to the existing functionality.
4. **Understandability:**
   A good design should be easily understandable, for which it should be modular and all the modules are arranged in layers.
5. **Completeness:**
   The design should have all the components like data structures, modules, and external interfaces, etc.
6. **Maintainability:**
   A good software design aims to create a system that is easy to understand, modify, and maintain over time. This involves using modular and well-structured design principles eg.(employing appropriate naming conventions and providing clear documentation). Maintainability in software Design also enables developers to fix bugs, enhance features, and adapt the software to changing requirements without excessive effort or introducing new issues.

Software Design Concepts:

Concepts are defined as a principal idea or invention that comes into our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system or software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:



The following **points should be considered while designing Software:**

1. **Abstraction- hide Irrelevant data**
   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.
2. **Modularity- subdivide the system**
   Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we are able to divide the system into components then the cost would be small.
3. **Architecture- design a structure of something**
   Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements

and the data of the structure. These components interact with each other and use the data of the structure in architecture.

4. **Refinement- removes impurities**
   Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
5. **Pattern- a repeated form**
   The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
6. **Information Hiding- hide the information**
   Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.
7. **Refactoring- reconstruct something**
   Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't affect the behavior of the design and improves the internal structure".

Different levels of Software Design:

There are three different levels of software design. They are:

1. **Architectural Design:**
   The architecture of a system can be viewed as the overall structure of the system & the way in which structure provides conceptual integrity of the system. The architectural design identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of the proposed solution domain.

2. **Preliminary or high-level design:**
   Here the problem is decomposed into a set of modules, the control relationship among various modules identified, and also the interfaces among various modules are identified. The outcome of this stage is called the program architecture. Design representation techniques used in this stage are structure chart and UML.

3. **Detailed design:**
   Once the high-level design is complete, a detailed design is undertaken. In detailed design, each module is examined carefully to design the data structure and algorithms. The stage outcome is documented in the form of a module specification document.

Software Design Strategies:

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Let us study them briefly:

Structured Design

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on 'divide and conquer' strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

Cohesion - grouping of all functionally related elements.

Coupling - communication between different modules.

A good structured design has high cohesion and low coupling arrangements.

Function Oriented Design

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions changes data and state of entire system.

- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

- **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.
  In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.
- **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Design Process

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.

- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them is defined.
- Application framework is defined.

Software Design Approaches

Here are two generic approaches for software designing:

Top Down Design

We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their on set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Bottom-up Design

The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

ANOTHER EXPLANATION OF DESIGN APPROACHES:

A good system design is to organize the program modules in such a way that are easy to develop and change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program.

Software Engineering is the process of designing, building, testing, and maintaining software. The goal of software engineering is to create software that is reliable, efficient, and easy to maintain. System design is a critical component of software engineering and involves making decisions about the architecture, components, modules, interfaces, and data for a software system.

System Design Strategy refers to the approach that is taken to design a software system. There are several strategies that can be used to design software systems, including the following:

1. Top-Down Design: This strategy starts with a high-level view of the system and gradually breaks it down into smaller, more manageable components.
2. Bottom-Up Design: This strategy starts with individual components and builds the system up, piece by piece.
3. Iterative Design: This strategy involves designing and implementing the system in stages, with each stage building on the results of the previous stage.
4. Incremental Design: This strategy involves designing and implementing a small part of the system at a time, adding more functionality with each iteration.
5. Agile Design: This strategy involves a flexible, iterative approach to design, where requirements and design evolve through collaboration between self-organizing and cross-functional teams.

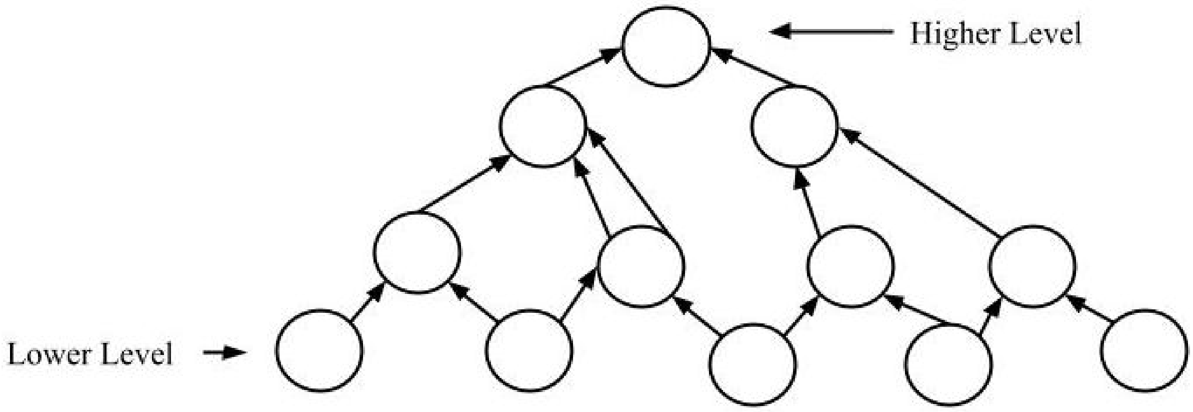
The choice of system design strategy will depend on the particular requirements of the software system, the size and complexity of the system, and the development methodology being used. A well-designed system can simplify the development process, improve the quality of the software, and make the software easier to maintain.

Importance :

1. If any pre-existing code needs to be understood, organized, and pieced together.
2. It is common for the project team to have to write some code and produce original programs that support the application logic of the system.

There are many strategies or techniques for performing system design. They are:

- **Bottom-up approach:**
  The design starts with the lowest level components and subsystems. By using these components, the next immediate higher-level components and subsystems are created or composed. The process is continued till all the components and subsystems are composed into a single component, which is considered as the complete system. The amount of abstraction grows high as the design moves to more high levels.
  By using the basic information existing system, when a new system needs to be created, the bottom-up strategy suits the purpose.



Advantages:

- The economics can result when general solutions can be reused.
- It can be used to hide the low-level details of implementation and be merged with the top-down technique.

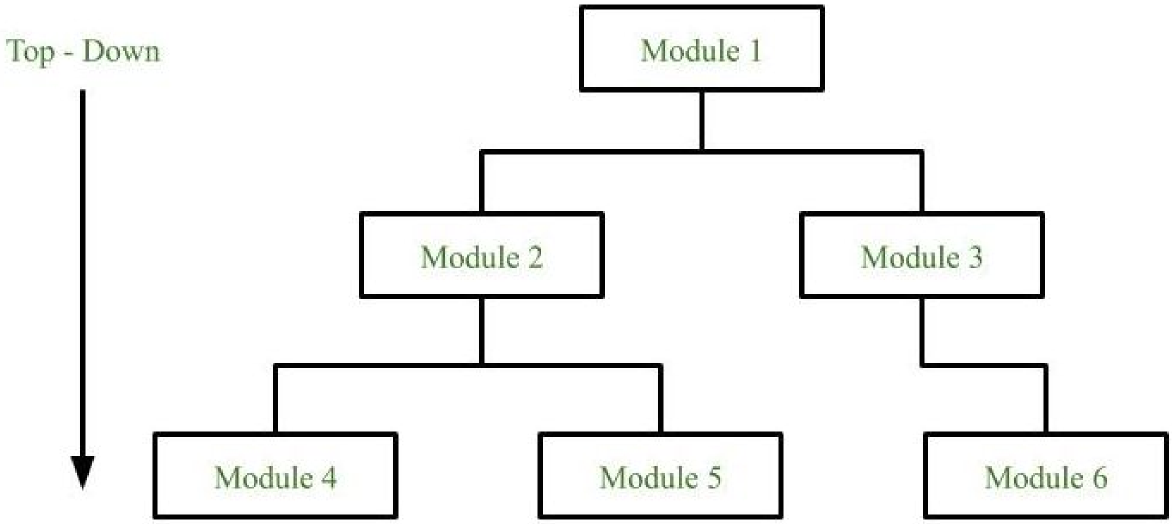
Disadvantages:

- It is not so closely related to the structure of the problem.

- High-quality bottom-up solutions are very hard to construct.
- It leads to the proliferation of ‘potentially useful’ functions rather than the most appropriate ones.

Top-down approach: Each system is divided into several subsystems and components. Each of the subsystems is further divided into a set of subsystems and components. This process of division facilitates forming a system hierarchy structure. The complete software system is considered a single entity and in relation to the characteristics, the system is split into sub-systems and components. The same is done with each of the sub-systems.
This process is continued until the lowest level of the system is reached. The design is started initially by defining the system as a whole and then keeps on adding definitions of the subsystems and components. When all the definitions are combined together, it turns out to be a complete system.

For the solutions of the software that need to be developed from the ground level, a top-down design best suits the purpose.



Advantages:

- The main advantage of the top-down approach is that its strong focus on requirements helps to make a design responsive according to its requirements.

Disadvantages:

- Project and system boundaries tend to be application specification-oriented. Thus it is more likely that the advantages of component reuse will be missed.
- The system is likely to miss, the benefits of a well-structured, simple architecture.
- **Hybrid Design:**
  It is a combination of both top-down and bottom-up design strategies. In this, we can reuse the modules.

Advantages of using a System Design Strategy:

1. Improved quality: A well-designed system can improve the overall quality of the software, as it provides a clear and organized structure for the software.
2. Ease of maintenance: A well-designed system can make it easier to maintain and update the software, as the design provides a clear and organized structure for the software.

3. Improved efficiency: A well-designed system can make the software more efficient, as it provides a clear and organized structure for the software that reduces the complexity of the code.
4. Better communication: A well-designed system can improve communication between stakeholders, as it provides a clear and organized structure for the software that makes it easier for stakeholders to understand and agree on the design of the software.
5. Faster development: A well-designed system can speed up the development process, as it provides a clear and organized structure for the software that makes it easier for developers to understand the requirements and implement the software.

Disadvantages of using a System Design Strategy:

1. Time-consuming: Designing a system can be time-consuming, especially for large and complex systems, as it requires a significant amount of documentation and analysis.
2. Inflexibility: Once a system has been designed, it can be difficult to make changes to the design, as the process is often highly structured and documentation-intensive.

Effective Modular Design in Software Engineering:

The role of effective modular design in software engineering:

Any software comprises of many systems which contains several sub-systems and those sub-systems further contains their sub-systems. So, designing a complete system in one go comprising of each and every required functionality is a hectic work and the process can have many errors because of its vast size.

Thus in order to solve this problem the developing team breakdown the complete software into various modules. A module is defined as the unique and addressable components of the software which can be solved and modified independently without disturbing (or affecting in very small amount) other modules of the software. Thus every software design should follow modularity.

The process of breaking down a software into multiple independent modules where each module is developed separately is called **Modularization**.

Effective modular design can be achieved if the partitioned modules are separately solvable, modifiable as well as compilable. Here separate compilable modules means that after making changes in a module there is no need of recompiling the whole software system.

In order to build a software with effective modular design there is a factor **"Functional Independence"** which comes into play. The meaning of Functional Independence is that a function is atomic in nature so that it performs only a single task of the software without or with least interaction with other modules. Functional Independence is considered as a sign of growth in modularity i.e., presence of larger functional independence results in a software system of good design and design further affects the quality of the software.

Benefits of Independent modules/functions in a software design:

Since the functionality of the software have been broken down into atomic levels, thus

developers get a clear requirement of each and every functions and hence designing of the software becomes easy and error free.
As the modules are independent they have limited or almost no dependency on other modules. So, making changes in a module without affecting the whole system is possible in this approach.
Error propagation from one module to another and further in whole system can be neglected and it saves time during testing and debugging.

Independence of modules of a software system can be measured using 2 criteria : Cohesion, and Coupling. These are explained as following below.

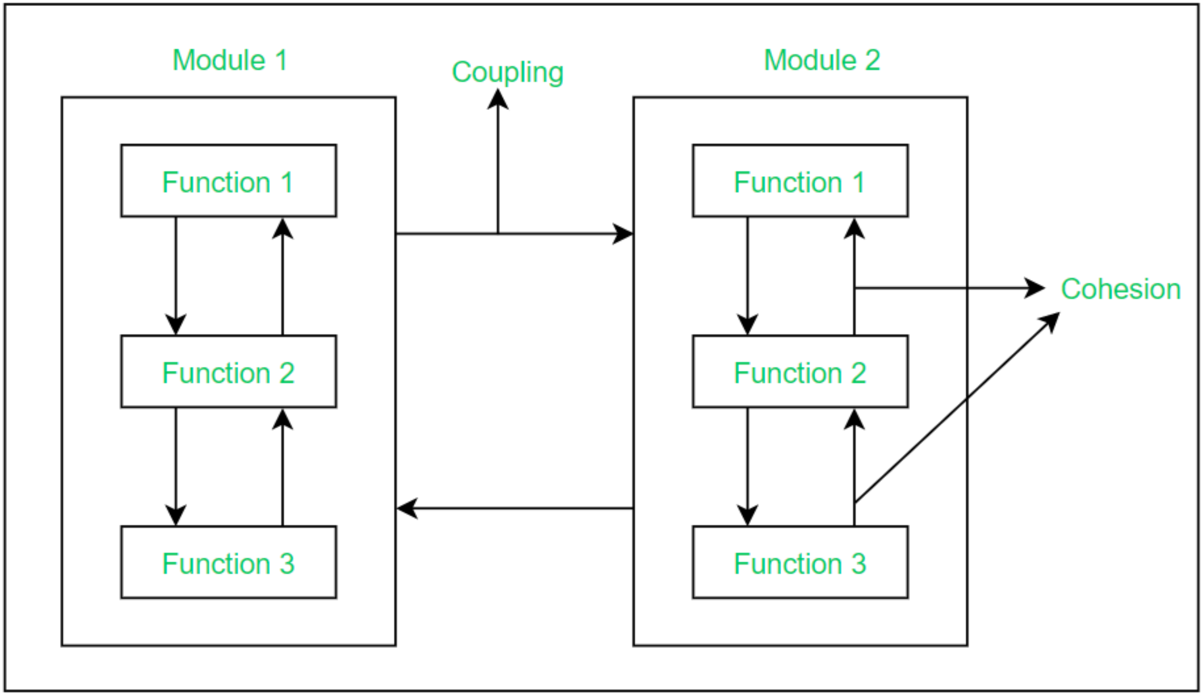


Figure – Cohesion and Coupling between 2 modules

Cohesion:
Cohesion is a measure of strength in relationship between various functions within a module. It is of 7 types which are listed below in the order of high to low cohesion:
1. Functional cohesion
2. Sequential cohesion
3. Communicational cohesion
4. Procedural cohesion
5. Temporal cohesion
6. Logical cohesion
7. Co-incidental cohesion

Coupling:
Coupling is a measure of strength in relationship between various modules within a software. It is of 6 types which are listed below in the order of low to high coupling:
1. Data Coupling
2. Stamp Coupling
3. Control Coupling
4. External Coupling
5. Common Coupling
6. Content Coupling

*A good software design requires **high cohesion** and **low coupling**.*

Coupling and Cohesion

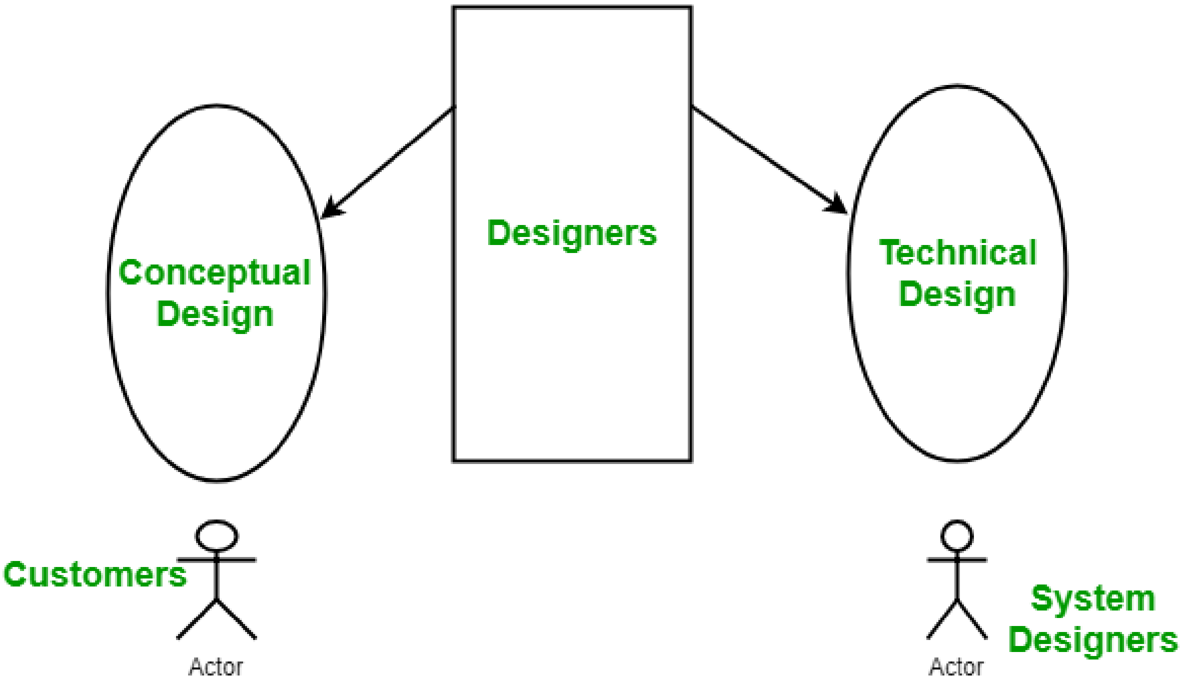
ntroduction: The purpose of Design phase in the Software Development Life Cycle is to produce a solution to a problem given in the SRS(Software Requirement Specification) document. The output of the design phase is Software Design Document (SDD). Coupling and Cohesion are two key concepts in software engineering that are used to measure the quality of a software system's design.

Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent and changes in one module have little impact on other modules.

Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.

Both coupling and cohesion are important factors in determining the maintainability, scalability, and reliability of a software system. High coupling and low cohesion can make a system difficult to change and test, while low coupling and high cohesion make a system easier to maintain and improve.

Basically, design is a two-part iterative process. First part is Conceptual Design that tells the customer what the system will do. Second is Technical Design that allows the system builders to understand the actual hardware and software needed to solve customer's problem.



Conceptual design of the system:

- Written in simple language i.e. customer understandable language.
- Detailed explanation about system characteristics.
- Describes the functionality of the system.
- It is independent of implementation.
- Linked with requirement document.

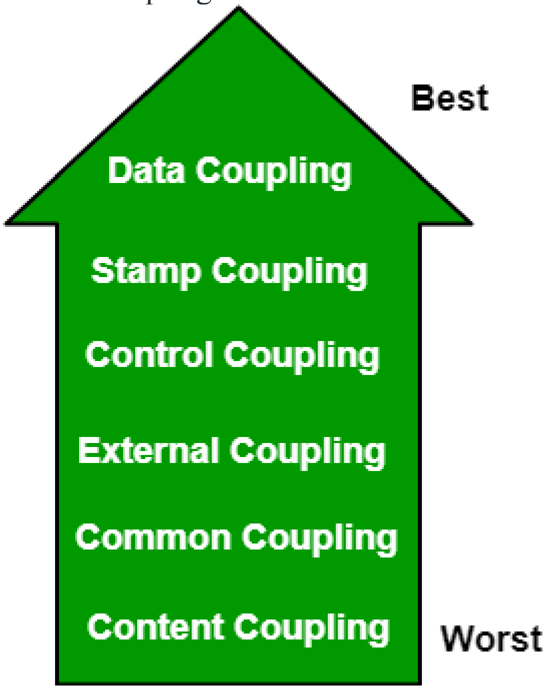
Technical Design of the System:

- Hardware component and design.
- Functionality and hierarchy of software components.
- Software architecture
- Network architecture
- Data structure and flow of data.
- I/O component of the system.
- Shows interface.

Modularization: Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

- Easy to understand the system.
- System maintenance is easy.
- A module can be used many times as their requirements. No need to write it again and again.

Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

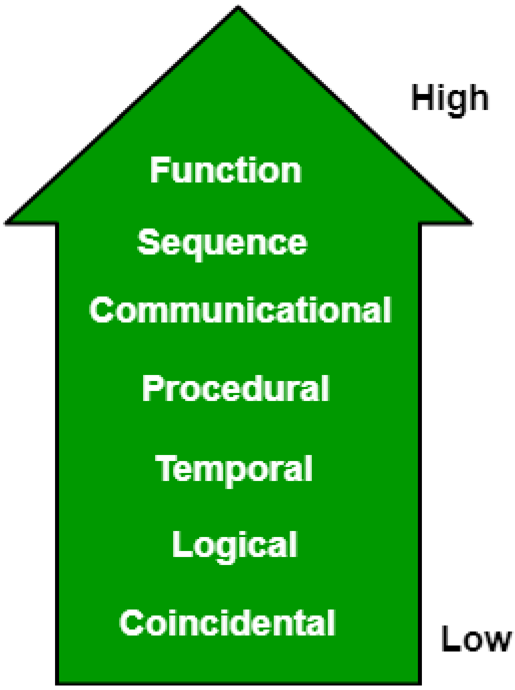


Types of Coupling:

- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data. Example-customer billing system.
- **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer.
- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.

- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.
- **Content Coupling:** In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.
- **Temporal Coupling:** Temporal coupling occurs when two modules depend on the timing or order of events, such as one module needing to execute before another. This type of coupling can result in design issues and difficulties in testing and maintenance.
- **Sequential Coupling:** Sequential coupling occurs when the output of one module is used as the input of another module, creating a chain or sequence of dependencies. This type of coupling can be difficult to maintain and modify.
- **Communicational Coupling:** Communicational coupling occurs when two or more modules share a common communication mechanism, such as a shared message queue or database. This type of coupling can lead to performance issues and difficulty in debugging.
- **Functional Coupling:** Functional coupling occurs when two modules depend on each other's functionality, such as one module calling a function from another module. This type of coupling can result in tightly-coupled code that is difficult to modify and maintain.
- **Data-Structured Coupling:** Data-structured coupling occurs when two or more modules share a common data structure, such as a database table or data file. This type of coupling can lead to difficulty in maintaining the integrity of the data structure and can result in performance issues.

Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



Types of Cohesion:

- **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
- **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
- **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.
- **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex-calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
- **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.
- **Logical Cohesion:** The elements are logically related and not functionally. Ex-A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
- **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.
- **Procedural Cohesion:** This type of cohesion occurs when elements or tasks are grouped together in a module based on their sequence of execution, such as a module that performs a set of related procedures in a specific order. Procedural cohesion can be found in structured programming languages.

- **Communicational Cohesion:** Communicational cohesion occurs when elements or tasks are grouped together in a module based on their interactions with each other, such as a module that handles all interactions with a specific external system or module. This type of cohesion can be found in object-oriented programming languages.
- **Temporal Cohesion:** Temporal cohesion occurs when elements or tasks are grouped together in a module based on their timing or frequency of execution, such as a module that handles all periodic or scheduled tasks in a system. Temporal cohesion is commonly used in real-time and embedded systems.
- **Informational Cohesion:** Informational cohesion occurs when elements or tasks are grouped together in a module based on their relationship to a specific data structure or object, such as a module that operates on a specific data type or object. Informational cohesion is commonly used in object-oriented programming.
- **Functional Cohesion:** This type of cohesion occurs when all elements or tasks in a module contribute to a single well-defined function or purpose, and there is little or no coupling between the elements. Functional cohesion is considered the most desirable type of cohesion as it leads to more maintainable and reusable code.
- **Layer Cohesion:** Layer cohesion occurs when elements or tasks in a module are grouped together based on their level of abstraction or responsibility, such as a module that handles only low-level hardware interactions or a module that handles only high-level business logic. Layer cohesion is commonly used in large-scale software systems to organize code into manageable layers.

Advantages of low coupling:

- Improved maintainability: Low coupling reduces the impact of changes in one module on other modules, making it easier to modify or replace individual components without affecting the entire system.
- Enhanced modularity: Low coupling allows modules to be developed and tested in isolation, improving the modularity and reusability of code.
- Better scalability: Low coupling facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed.

Advantages of high cohesion:

- Improved readability and understandability: High cohesion results in clear, focused modules with a single, well-defined purpose, making it easier for developers to understand the code and make changes.
- Better error isolation: High cohesion reduces the likelihood that a change in one part of a module will affect other parts, making it easier to
- isolate and fix errors. Improved reliability: High cohesion leads to modules that are less prone to errors and that function more consistently,
- leading to an overall improvement in the reliability of the system.

Disadvantages of high coupling:

- Increased complexity: High coupling increases the interdependence between modules, making the system more complex and difficult to understand.
- Reduced flexibility: High coupling makes it more difficult to modify or replace individual components without affecting the entire system.
- Decreased modularity: High coupling makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code.

Disadvantages of low cohesion:

- Increased code duplication: Low cohesion can lead to the duplication of code, as elements that belong together are split into separate modules.
- Reduced functionality: Low cohesion can result in modules that lack a clear purpose and contain elements that don't belong together, reducing their functionality and making them harder to maintain.
- Difficulty in understanding the module: Low cohesion can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

DIFFERENCE BETWEEN COHESION AND COUPLING:

Cohesion	Coupling
Cohesion is the concept of intra-module.	**Coupling is the concept of inter-module.**
Cohesion represents the relationship within a module.	**Coupling represents the relationships between modules.**
Increasing cohesion is good for software.	**Increasing coupling is avoided for software.**
Cohesion represents the functional strength of modules.	**Coupling represents the independence among modules.**
Highly cohesive gives the best software.	**Whereas loosely coupling gives the best software.**
In cohesion, the module focuses on a single thing.	**In coupling, modules are connected to the other modules.**
Cohesion is created between the same module.	**Coupling is created between two different modules.**
There are Six types of Cohesion **1. Functional Cohesion.** **2. Procedural Cohesion.** **3. Temporal Cohesion.** **4. Sequential Cohesion.** **5. Layer Cohesion.** **6. Communication Cohesion.**	**There are Six types of Coupling** **1. Common Coupling.** **2. External Coupling.** **3. Control Coupling.** **4. Stamp Coupling.** **5. Data Coupling** **6. Content Coupling.**

Design Documentation

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language. The software design process can be divided into the following three levels of phases design:

- Interface Design
- Architectural Design
- Detailed Design

Software Design Document:

Software Design Document is a written document that provides a description of a software product in terms of architecture of software with various components with specified functionality.

The design specification addresses different aspects of the design model and is completed as the designer refines his representation of the software. These design documents are written by software engineers/designers or project managers and further passed to the software development team to give them an overview of what needs to be built and how.

S.No	Software Design Document	Module, Subpart
01.	Reference Documents	1. Existing software documentation 2. System Documentation 3. Vendor(hardware or software) documents 4. Technical reference
02.	Modules for each module	1. Processing narrative 2. Interface description 3. Design language(or other) description 4. Modules used
03.	Scope	1. System objective 2. Hardware, software and human interfaces 3. Major software functions 4. Externally defined database 5. Major design constraints, limitations
04.	Design Description	1. Data description 2. Derived program structure

S.No	Software Design Document	Module, Subpart
		3. Interface within structure
05.	Test Provisions	1. Test guidelines 2. Integration strategy 3. Special considerations
06.	Packaging	1. Special program overlay provisions 2. Transfer consideration
07.	File Structure and global data	1. External Files structure 2. Global data 3. File and data cross – reference
08.	Requirement cross-reference	1. cross-reference

Importance of Design Documentation:

1. Requirements are well understood: With proper documentation, we can remove inconsistencies and conflicts about the requirements. Requirements are well understood by every team member.
2. Architecture/Design of product: Architecture/Design documents give us a complete overview of how the product look like and better insight to the customer/user about their product.
3. New Person can also work on the project: New person to the project can very easily understand the project through documentations and start working on it. So, developers need to maintain the documentation and keep upgrading it according to the changes made in the product/software.
4. Everything is well Stated: This documentation is helpful to understand each and every working of the product. It explains each and every feature of the product/software.
5. Proper Communication: Through documentation, we have good communication with every member who is part of the project/software. Helpful in understanding role and contribution of each and every member.

Unified Modeling Language (UML) | An Introduction:
Unified Modeling Language (UML)
is a general purpose modelling language. The main aim of UML is to define a standard way to
visualize
the way a system has been designed. It is quite similar to blueprints used in other fields of engineering. UML is
not a programming language
, it is rather a visual language. We use UML diagrams to portray the
behavior and structure

of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO) published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

Do we really need UML?

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

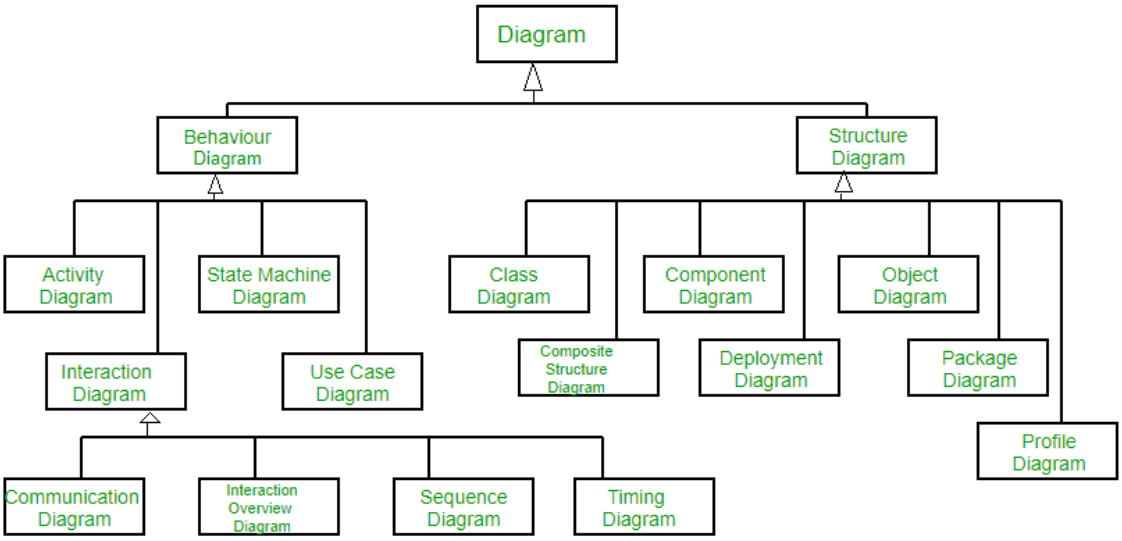
UML is linked with

object oriented

design and analysis. UML makes the use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:

1. **Structural Diagrams** – Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
2. **Behavior Diagrams** – Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

The image below shows the hierarchy of diagrams according to UML 2.2



Object Oriented Concepts Used in UML –

1. **Class** – A class defines the blue print i.e. structure and functions of an object.
2. **Objects** – Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.
3. **Inheritance** – Inheritance is a mechanism by which child classes inherit the properties of their parent classes.

4. **Abstraction –** Abstraction in UML refers to the process of emphasizing the essential aspects of a system or object while disregarding irrelevant details. By abstracting away unnecessary complexities, abstraction facilitates a clearer understanding and communication among stakeholders.
5. **Encapsulation –** Binding data together and protecting it from the outer world is referred to as encapsulation.
6. **Polymorphism –** Mechanism by which functions or entities are able to exist in different forms.

Additions in UML 2.0 –

- Software development methodologies like agile have been incorporated and scope of original UML specification has been broadened.
- Originally UML specified 9 diagrams. UML 2.x has increased the number of diagrams from 9 to 13. The four diagrams that were added are : timing diagram, communication diagram, interaction overview diagram and composite structure diagram. UML 2.x renamed statechart diagrams to state machine diagrams.
- UML 2.x added the ability to decompose software system into components and sub-components.

Structural UML Diagrams –

1. **Class Diagram –** The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes,their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.
2. **Composite Structure Diagram –** We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.
3. **Object Diagram –** An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.
4. **Component Diagram –** Component diagrams are used to represent how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.

5. **Deployment Diagram –** Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.
6. **Package Diagram –** We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organise UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organise class and use case diagrams.

Behavior Diagrams –

1. **State Machine Diagrams –** A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams** . These terms are often used interchangeably.So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.
2. **Activity Diagrams –** We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram.An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.
3. **Use Case Diagrams –** Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.
4. **Sequence Diagram –** A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.
5. **Communication Diagram –** A Communication Diagram(known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams,however, communication diagrams represent objects and links in a free form.
6. **Timing Diagram –** Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them

to show time and duration constraints which govern changes in states and behavior of objects.

7. **Interaction Overview Diagram –** An Interaction Overview Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams.
   Use Case, Object and Class, Interaction diagram: Sequence & Collaboration: COVERED IN ASSIGNMENT AND LAB

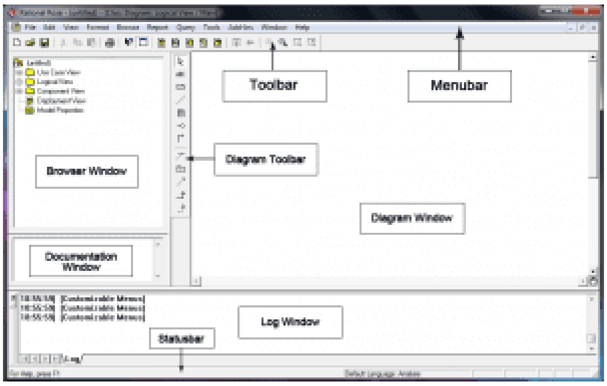
   INTRODUCTION TO RATIONAL ROSE TOOLS:
   Rational Rose is an object-oriented programming (OOP) and unified modeling language (UML) tool to design enterprise-level software applications and components. It creates visual software application models under object-oriented principles. Example application models include the creation of actors, use cases, relationships, objects, entities, etc. Rational Rose uses classical UML concepts to graphically model software applications. This facilitates documenting the environment, requirements and overall design.

OOP methodologies use UML to graphically depict software application behavior and architecture. These models are the building blocks and development blueprint of the application's entire construction process. The popular Rational Rose modeling tool allows developers to create entire architecture or component-level system models, while depicting relationship and control flow.

Rational Rose is a powerful graphical user interface (GUI) modeling tool using efficient and user-friendly drag and drop and design maneuverability. Certain Rational Rose versions actually produce relevant source code for designed models.

- ROSE stands for Rational Object-oriented Software Engineering.
- Rational Rose is developed by Rational Corporation which is under IBM.
- Rational Rose is a tool for modeling software systems.
- Rational Rose supports UML.
- Rational Rose is a tool that supports round-trip engineering means a tool that supports conversion of a model to code and from code to a model.

Rational Rose Interface:



Menubar: The menubar consists of several menus like the file menu, edit menu, view menu etc. All these menus contain several options.

Toolbar: The toolbar contains the most frequently used actions like New, Open, Save etc…

Statusbar: The statusbar at the bottom displays status messages.

Browser Window: The browser window displays the views: Use Case View, Logical View, Component View and Deployment View. Each of these views contains the diagrams.

Diagram Toolbar: The diagram toolbar displays the symbols of the respective type of diagram.

Diagram Window: The diagram window is the place where the user draws the diagrams using the symbols from the diagram toolbar.

Log Window: This window is used to display error messages, warnings and information messages.