UNIT-3

DESIGN

For small projects, after analysing the requirements, using the SRS document we can start writing programs i.e we can start coding (implementation of s/w). But for large & complex projects, it is very difficult to implement the requirements in the SRS document directly. It is necessary to bridge the gap between requirements specification and coding. This bridge is the software design.

- "Design" in the software development process describes "how" a software system should be produced in order to make it functional, reliable and easy to understand & maintain.
- SRS document tells "what" a system does, and becomes input to the design process, which tells "how" the system does and result of the design process is a software design document ("SDD"). So the purpose of design phase is to produce a solution to a problem given in SRS document.
- Design creates a representation (or) model of the software which provides details about software data structures, architecture, interfaces, and components that are necessary to implement the system.
- Design is a two part, iterative process. First, we produce "conceptual design" that tells the customer exactly what the system will do. Once the customer approves the conceptual design, we translate the conceptual design into a much more detailed document, which is "technical design" that allows the system developers to understand the actual hardware & software needed to build the system.

Design Process & Design quality →

- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the s/w.
- Initially, the blueprint depicts (represents) a complete view of system to be implemented. That is, the design is represented at a high level of abstraction. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.
- A good design is the key to successful product. A well designed system is easy to implement & easy to understand and it will be reliable.
- Throughout the design process, the quality of the evolving design is assessed with a series of "formal technical reviews". There are three characteristics that serve as a guide for the evolution of a good design:
  - ⊛ The design must implement all of the explicit requirements contained in the analysis model, and all of the implicit requirements desired by the customer.
  - ⊛ The design must be a readable & understandable guide for those who generate code and for those who test & support the software.
  - ⊛ The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from the implementation perspective.

Quality guidelines : →

Good design is achieved through the application of fundamental design principles, systematic methodology, and thorough review.

B. Roy

B. Roy

In order to evaluate the quality of design representation, we must establish technical criteria for good design. Good design is achieved by following the quality guidelines as below:

1) A design should exhibit an architecture (a) that has been created using recognizable architectural styles (&) patterns, (b) that is composed of components that exhibit good design characteristics, (c) that can be implemented in an evolutionary fashion.

2) A design should be modular; that is, the S/W should be logically partitioned into elements (&) sub systems.

3) A design should contain distinct representations of data, architecture, interfaces and components.

4) A design should lead to data structures that are appropriate & the classes to be implemented and are drawn from recognizable data patterns.

5) A design should lead to components that exhibit independent functional characteristics.

6) A design should lead to interfaces that reduce the complexity of connection between components and with the external environment.

7) A design should be derived using a repeatable method that is driven by information obtained during S/W requirements analysis.

8) A design should be represented using a notation that effectively communicates its meaning.

Quality attributes →

There are a set of quality attributes that are specified with the acronym "FURPS" ← functionality, usability, reliability, performance & supportability. The "FURPS" quality attributes represent a target for software design.

⊛ Functionality is assessed by evaluating the capabilities of the program, the generality of functions & security of overall system.

⊛ Usability is assessed by considering human factors, overall aesthetics consistency, and documentation.

⊛ Reliability is evaluated by measuring the frequency & severity of failure the accuracy of output results, the ability to recover from failure, and the predictability of program.

⊛ Performance is measured by processing speed, response time, resource consumption, throughput and efficiency.

⊛ Supportability combines extensibility (ability to extend the program), adaptability serviceability. In addition testability, compatability, configurability.

- All the SW quality attributes are not considered equally. One application may stress functionality with a special emphasis on security, another application may focus on performance with the importance of processing speed. Regardless of the importance, these quality attributes must be considered as design commences, not after the design is complete.

Design Concepts ⇒

- A set of fundamental SW design concepts has evolved over the history of SW engineering. Each provides the SW designer with a foundation from which more sophisticated design methods can be applied.
- A good SW engineer is one, who recognizes the difference between getting a program to work, and getting it right. Fundamental "design concepts" provide the necessary framework for "getting it right". We need to follow the design concepts while designing any SW product.

Abstraction →

When we consider a modular solution to any problem, many levels of abstraction can be presented. At the highest level of abstraction, a solution is stated in broad terms using the language of problem environment. At lower levels of abstraction, a solution is stated with more detailed description. Finally, at the lowest levels of abstraction, the solution is stated in a manner that can be directly implemented.

- As we move through different levels of abstraction, we create procedural and data abstractions.
- A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of procedural abstraction implies these functions, but specific details are not expressed (hidden). An example of a procedural abstraction could be the word "open" for a door. Open implies a long sequence of actions (procedural steps) such as walk to the door; grasp knob & turn it, pull the door & step away from moving door.
- A data abstraction is a named collection of data that describes a data object. In the above eg; for procedural abstraction open, we can define a data abstraction "door". The data abstraction for "door" would encompass a set of attributes (such as: door type, swing direction, door weight, dimensions etc) that describe the door. So procedural abstraction (open) would make use of information contained in the attributes of data abstraction (door).

(2) Architecture →

- The architecture broadly refers to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.

- Architecture is the structure of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
- Architectural design can be represented using different models such as structural models; Framework models, Dynamic models, process models and functional models.

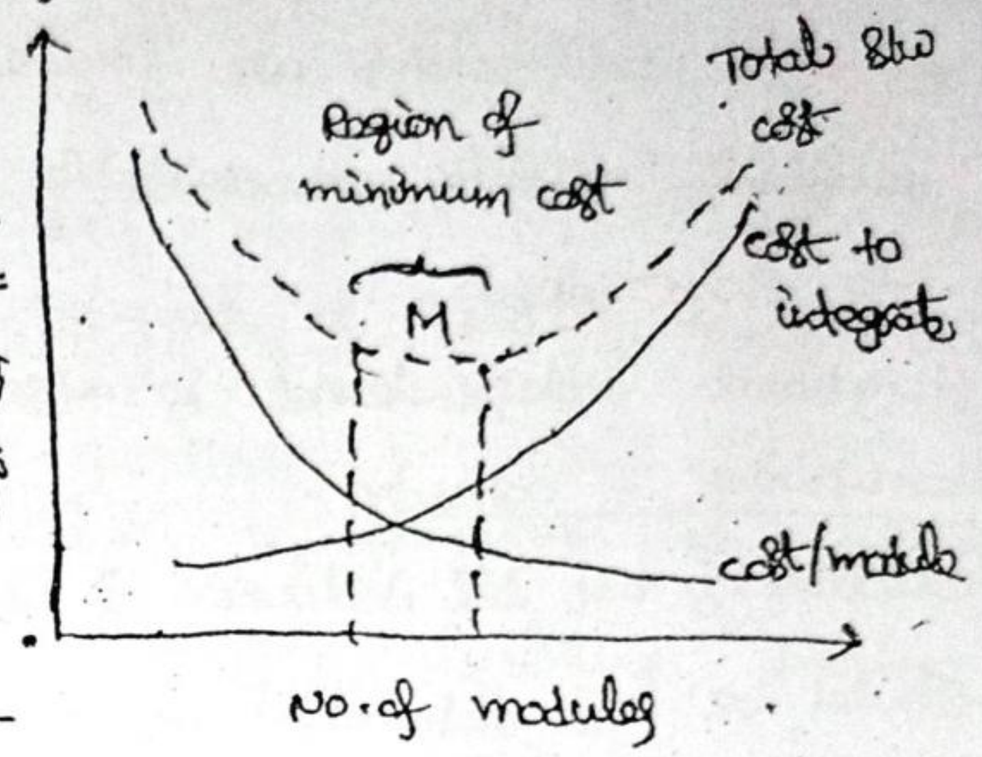
(3) Patterns →

- A pattern is a model to making something. It is an example to follow.
- A design pattern describes a design structure that solves a particular design problem within a specific context.
- The intent of each design pattern is to provide a description that enables a designer to determine (i) whether the pattern is applicable to current work, (ii) whether the pattern can be reused (saving design time) (iii) whether the pattern can be used as a guide for developing a similar, but functionally different pattern.

(4) Modularity →

- Modularity is the only feature of software that allows a program to be intellectually manageable.
- Monolithic software is a large program composed of a single module can not be easily understood by a software engineer. So the software that is going to be developed should be divided into components (modules) to implement easily and should be integrated to satisfy problem requirements. This leads to "divide and conquer" strategy ↘
- It is easier to solve a complex problem when we break it into manageable pieces.

If we subdivide software indefinitely, the effort required to integrate the modules will become more. But the effort required to develop an individual module decreases as the no. of modules increases. But at the same time, as the no. of modules grows, the effort (or) cost required to integrate the modules also grows. This leads to a total cost (or) effort curve as shown in figure.



- So we should divide into 'M' modules where 'M' is the region of minimum total cost (to divide & integrate). we can modularise effectively based on other design concepts.
- we should modularise a s/w so that development can be more easily planned; changes can be easily accomodated; testing & debugging can be conducted more efficiently and maintenance can be done without serious side effects.

) Information hiding →

- Modules should be specified and designed so that information (algorithms & data) contained with in a module is not accessible to other modules that have no need for such information. Information hiding is done with the help of abstraction (procedural & data abstraction). Information hiding is useful when modifications are required during testing & during s/w maintenance. Because most data & procedure are hidden from other parts of s/w and so easily introduced during modifications are less likely to ~~more~~ propagate (more) to other locations with in the s/w.

(6) Functional Independence →

- Functional independence is achieved by developing modules with single minded function and with less interacting with other modules.
- Independent modules are easier to maintain and test because design changes are limited, error propagation is reduced & reusable modules are possible. So functional independence is a key to good design.
- Functional independence is assessed using two qualitative criteria: coupling & cohesion.
- Coupling is an indication of interdependence among modules. The modules should be less dependent on one another.
- Cohesion is an indication of functional strength of a module. A cohesive module performs a single task effectively.
- The new modules must be designed with low coupling and high-cohesion to achieve functional independence.

(7) Refinement →

- Stepwise refinement is a top-down design strategy.
- Refinement is a process of elaboration. we begin with a statement of function that is defined at high level of abstraction. That is, the statement describes function & information conceptually but provides no information about internal workings of the function.
- Refinement causes the designer to elaborate on the original statement, providing more & more details as each successive refinement occurs.
- Abstraction enables a designer to specify procedure & data hiding low-level details. Refinement helps the designer to reveal low-level details as design progresses.

(8) Refactoring →

- Refactoring is a reorganization technique that simplifies the design of code of a component without changing its function or behavior.
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of code of design but improves its internal structure.
- when sw is refactored, the existing design is examined for redundancy unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures that can be corrected to yield a better design.
- for eg; a first design iteration might yield a ~~low cohesion~~ component that exhibits low cohesion. That component can be refactored into three separate components, each exhibiting high cohesion. The result will be the sw that is easier to integrate, test and maintain.

(9) Design classes →

- Analysis model defines a set of analysis classes. Each of these classes describes some elements of the problem domain, focusing on aspects of the problem that are visible to customer. The analysis classes are represented at higher levels of abstraction.
- As the design model evolves, the sw team must define a set of design classes that (1) refine the analysis classes by providing design details that will enable the classes to be implemented, and (2) create a new set of design classes that implement a software infrastructure to support the business solution.

Five different types of design classes can be defined : Each represents a different layer of design architecture :

* User interface classes → Define all abstractions that are necessary for human - computer interaction.
* Business domain classes → Identify the attributes & services (methods) that are required to implement some element of business domain.
* process classes → Implement lower - level abstractions required to fully manage the business domain classes.
* Persistent classes → Represent data stores that will persist (remain unchanged) beyond the execution of s/w.
* System classes → Implement s/w management & control functions that enable the system to operate and communicate within its computing environment and with the outside world.

- As the design model evolves, the s/w team must develop a complete set of attributes & operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design class. i.e analysis classes represent objects using business domain terminology. Design classes present more technical details as a guide for the implementation.
- A well - formed design class should have four characteristics :
  (i) A design class should be complete & sufficient to implement.
  (ii) A design class should be primitive.
  (iii) A design class should exhibit high cohesion.
  (iv) A design class should have low - coupling.