SEMCOM

Faculty Name: Dr. Jay Nanavati
Sem: 5 (TYBCA)
Subject: Software Engineering
Unit-3 Software Design

Introduction

Design activity begins when requirements document is available.
Requirements specifications activity is a problem domain; design activity is the first step in moving towards solution domain.

Design (as a verb): The process of creating a blueprint or a plan for the system.
Goal of the design process is to produce a model of a system.

Design (as a noun): A blueprint or a plan for the system. It is the output of the design *process*.

There are two levels of design process:

At first level the focus is on deciding which modules are needed, specifications of modules, how modules should be interconnected. This is called system design or top-level design. The system design controls the major structural characteristics of the system.
The system design has major impact on the testability and modifiability of a system, and it impacts the efficiency.

In second level, the internal design of the modules, or how the specifications are to be satisfied is decided. It is called detailed design or logic design.
Detailed design expands the system design to contain a more detailed description of processing logic and data structures so that the design is sufficiently complete for coding.

Design objectives

1. Correctness:
A design is correct if a system is built precisely according to design satisfies the requirements.
2. Verifiability:
A design is verifiable if we easily verify whether design is correct or not.
3. Completeness:
A design is complete if it implements all the specifications.
4. Efficiency:
Efficiency of any system is concerned with proper use of scarce resources.
These resources are processor time and memory. An efficient system is on that consumes less processor time and requires less memory.
5. Traceability:
A design is traceable if all design elements can be traced to some requirements.
6. Simplicity and Understandability:
Simplicity and understandability helps a lot in maintenance of software. During maintenance the first necessary step is that a maintainer has to understand the system to be maintained. A simple and understandable design will make job of maintainer easier.

Input /Output /Exit Criteria of Design Phase

Input: SRS Output: System Design & Detailed Design
Exit Criteria: Design which is verified & approved for quality.

Design Principles

Design Principles are guidelines that may be followed to create a good design of the system.

1. Problem Partitioning

The goal of software design is to divide the problem in to manageable pieces that can be solved separately. The reason behind solving problem dividing in pieces is that cost of solving entire problem is more than the sum of cost of solving all the pieces. Different pieces are not independent as they together form the entire system. The different pieces have to cooperate and communicate to solve larger problem. This communication adds complexity, which arises due to partitioning. As the number of components increases the cost of partitioning together with cost of added complexity may become more than the savings achieved by partitioning. It is at this point that no further partitioning is required.
Total independence of modules of one system is not possible, but design process should support as much independence as possible between modules. Proper partitioning will make things easier to maintain by making design easier to understand.

2. Modularity

A system is considered modular if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.
Modularity helps in

- ✓ System debugging as the system problem is isolated to a component
- ✓ System Repair as changing a part of the system is easy as it few other parts.
- ✓ System building as a modular system can be easily built by putting its modules together.

A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support well-defined abstraction and have a clear interface through which it can interact with other modules.
For easily understandable and maintainable systems, modularity is clearly the basic objective.

3. Abstraction

Abstraction means to consider what a component does, without considering how it does so.
Abstraction of component describes external behavior of that component without considering internal details that produce the behavior.
Abstraction is important in problem partitioning. In partitioning we divide system in to smaller components. These components are not isolated, they interact with each other. To decide how a component interacts with other components, the designer has to know the external behavior **i.e. abstraction** of other components.
Abstraction of existing components plays an important role in maintenance phase. For maintenance first step is to understand existing system for which we need to identify the abstractions of components. Using these abstractions working of entire system can be understood.
The basic goal of system design is to specify modules and their abstract specifications. Once modules are specified, during detailed designer can concentrate on one module at a time. The modules are implemented in so that abstract specifications of each module are satisfied.
There are two common abstraction mechanisms for software systems are: functional abstraction and data abstraction.
In functional abstraction, a module is specified by function it performs. For example, a module to compute the log of a value can be abstractly represented by function log.
In functional abstraction, the overall transformation function for the function is partitioned into smaller functions that comprise system function.
In Data abstraction, data is treated as objects with some predefined operations on them.

4. Top-Down and Bottom-Up Approaches

A system consists of components, which have components of their own. A system is hierarchy of components. The highest-level component corresponds to the total system. To design such a system, there are two possible approaches top-down and bottom-up approaches.
The top-down approach starts from highest-level component of the hierarchy and proceeds through to lower levels.

In Bottom-Up approach starts with the lower level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved. Top-down design methods often result in some form of stepwise refinement. Starting from an abstract design, in each step the design is refined to more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.
A bottom-up design approach starts with designing the most basic components and proceeds to higher level components so that use lower-level components.

A top-down approach is suitable if the specifications of the system are clearly known and the system development is from scratch.
A bottom-up approach is suitable if the is to be built from existing system.

Module-Level Concepts
A module is logically separable part of a program. It is program unit that is discreet and identifiable with respect to compiling and loading.
A module can be a macro, function, procedure, process or a package.
Characteristics of modules:

1. Modules contain instructions, processing logic and data structures.
2. Modules can be separately compiled.
3. Module segments can be used by invoking a name and some parameters.
4. Modules can use other modules.

Modularization allows the designer to decompose a system in to functional units, to impose hierarchical ordering on function usage, to implement data abstraction and develop independently useful system.
In system using functional abstraction, Coupling and Cohesion are two criteria for selecting modules

Coupling

- Two modules are considered independent if one can function completely without the presence of other.
- All the modules in system cannot be independent of each other, because they must interact with each other so that together they produce the behavior of system.
- The more connections between modules the more they are dependent.
- If one module is dependent on other module and if you want to know the behavior of the module then we are required to understand the behavior of other module.
- If there are fewer and simpler connections between modules, it is easier to understand one without understanding another.
- Coupling between modules is strength of interconnections between modules.
- Coupling is a measure of interdependence among modules.
- Highly coupled modules are joined by strong interconnections. Loosely coupled modules are joined by weaker interconnections. Independent modules have no interconnections.
- To solve and modify a module separately, the module has to be loosely coupled with other modules.
- Coupling is an abstract concept and is not easily quantifiable. So no formulas can be given to determine the coupling between the two modules.
- The major factors influencing coupling between modules are the type of connection between modules, the complexity of the interface, and the type of information flow between modules.
- Coupling increases with the complexity and obscurity of interface between modules.
- To keep coupling low we minimize the number of interfaces per module and complexity of each interface.
- Complexity of interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling.
- There two types of information that can flow from one module to another: Control information, Data Information.

Cohesion

- Cohesion of module represents how tightly bound the internal elements are to one another.
- The greater the cohesion of each module the lower the coupling between modules is.

- There are several levels of the cohesion:

1) Coincidental
2) Logical
3) Temporal
4) Procedural
5) Communicational
6) Sequential
7) Functional

- 1. Coincidental cohesion:
  - ✓ Coincidental Cohesion occurs when there is no meaningful relationship among the elements of a module.
  - ✓ It can occur if an existing program is modularized by dividing it into pieces and making different pieces modules.
  - ✓ If a module is created to save duplicate code by combining some part of code that occurs at many different places, that module is likely to have coincidental cohesion.
  - ✓ In this situation, the statements in the module have no relationship with each other.
  - ✓ If one of the module using the code needs to be modified then this modification includes common code, it is likely that other modules using the code do not want the code to be modified and may behave incorrectly. The modules using such common modules are therefore not modifiable separately and have strong interconnections between them.
- 2. Logical Cohesion:
  - ✓ A module has logical cohesion if there is some logical relationship between the elements of module and the elements perform function that falls in same logical class.
  - ✓ A typical example of this kind of cohesion is a module that performs all inputs and all outputs.
- 3. Temporal Cohesion:
  - ✓ Temporal cohesion is same as logical except that the elements are also related in time are executed together.
  - ✓ Modules that perform activities like activities like “initialization”, “clean-up” and “termination” are usually temporally bound.
- 4. Procedural Cohesion:
  - ✓ A procedurally cohesive module contains elements that belong to common procedural unit.
  - ✓ For example, a loop or sequence of decision statements in a module may be combined to form a separate modules.
- 5. Communicational Cohesion:
  - ✓ A module with communicational cohesion has elements that are related by reference to same input and output data.
  - ✓ For example, a module to print a record.
- 6. Sequential Cohesion:
  - ✓ When the elements are together in a module because the output of one forms input to another it is called sequential cohesion.
  - ✓ For example, a function which reads data from files and processes data.
- 7. Functional Cohesion:
  - ✓ In functional cohesion all elements together perform a single function.
  - ✓ For example, modules for “sorting the array” and “compute square root”

Structured Design Methodology (Overview)

- SDM views every software system as having inputs that are converted to desired outputs.
- Here entire system is viewed as transformation function that transforms the given inputs to outputs.
- So designing a software system is called designing transformation function.
- SDM is function-oriented and heavily based on functional abstractions and functional decomposition.
- The aim of SDM is to design a system so that the programs implementing the design would have a nice hierarchical structure.
- A module with subordinates does not actually perform much computation. The actual computation is performed by its subordinates, and the module largely coordinated data flow between subordinates to get computation done.
- The subordinates in turn can get bulk of their work done by their subordinates until their atomic modules have no subordinates are reached.
- Factoring is the process of decomposing a module so that the bulk of its work done by its subordinates.

- A system is said to be completely factored if all the actual processing is accomplished by bottom-level atomic modules if non-atomic modules largely perform the jobs of control and coordination.
- There are four major steps:
  1. Restate problem as a DFD.
  2. Identify the most abstract input and output data elements.
  3. First-Level Factoring.
  4. Factoring of input, output and transform branches.

SDM Functional/Traditional/Conventional approach	**Object Oriented/ Data Abstraction Approach**
1. The design consists of module definitions with each module supporting a functional abstraction	In object-oriented approach, the modules represent data abstraction
2. A system is viewed as a transformation function transforming input to output.	A system is viewed as a set of objects providing some services.
3. It is not easier to produce and understand design.	3. It is easier to produce and understand design.
4. Reuse of modules is not possible and hence development cost and time increases.	4. Reuse of objects is possible thereby reducing development cost & time.
5. It does not allow building Object libraries.	5. It allows building Object libraries.