6 SOFTWARE DESIGN BASICS

Software Design Basics

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 for more specific and detailed requirements in software terms. The output of this process can directly be used in implementation in programming languages. Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from the problem domain to the 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 an idea of the proposed solution domain.
- **High-level Design**- The high-level design breaks the 'single entity-multiple component' concept of architectural design into a less-abstracted view of sub-systems and modules and depicts their interaction with each other. The high-level design focuses on how the system along with all of its components can be implemented in the form of modules. It recognizes the modular structure of each sub-system and their relation and interaction with each other.
- **Detailed Design**- Detailed design deals with the implementation part of what is seen as a system and its subsystems in the previous two designs. It is more detailed about modules and their implementations. It defines the logical structure of each module and its interfaces to communicate with other modules.

Modularization

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out the task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.
Modular design unintentionally follows the rules of the 'divide and conquer' problem-solving strategy this is because there are many other benefits attached to the modular design of the software.

Advantages of modularization:

- Smaller components are easier to maintain
- Programs can be divided based on functional aspects
- The desired level of abstraction can be brought into the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from a security aspect

Concurrency

Back in time, all software are meant to be executed sequentially. By sequential execution, we mean that the coded instruction will be executed one after another implying only one portion of the program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.
In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules, and executing them in parallel. In other words, concurrency provides the capability of the software to execute more than one part of code in parallel to each other. The programmers and designers must recognize those modules, which can be made parallel execution.

Example

The spell check feature in a word processor is a module of the software, which runs alongside the word processor itself.

Coupling and Cohesion

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together to achieve some tasks. They are though, considered as a single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

Cohesion

Cohesion is a measure that defines the degree of intradependability within elements of a module. The greater the cohesion, the better the program design. There are seven types of cohesion, namely –

- **Co-incidental cohesion** - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not accepted.
- **Logical cohesion** - When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion** - When elements of the module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion** - When elements of the module are grouped, which are executed sequentially to perform a task, it is called procedural cohesion.
- **Communicational cohesion** - When elements of the module are grouped, which are executed sequentially and work on the same data (information), it is called communicational cohesion.
- **Sequential cohesion** - When elements of a module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion** - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of a module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program. There are five levels of coupling, namely -

- **Content coupling** - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling**- When multiple modules have read and write access to some global data, it is called common or global coupling.

- **Control coupling**- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling**- When multiple modules share a common data structure and work on a different part of it, it is called stamp coupling.
- **Data coupling**- Data coupling is when two modules interact with each other utilizing passing data (as a parameter). If a module passes data structure as a parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

Design Verification

The output of the software design process is design documentation, pseudo-codes, detailed logic diagrams, process diagrams, and a detailed description of all functional or non-functional requirements.
The next phase, which is the implementation of software, depends on all outputs mentioned above.
It then becomes necessary to verify the output before proceeding to the next phase. The early any mistake is detected, the better it is or it might not be detected until testing of the product. If the outputs of the design phase are informal notation forms, then their associated tools for verification should be used otherwise a thorough design review can be used for verification and validation.
With a structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is important for good software design, accuracy, and quality.

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 the 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 a problem into several well-organized elements of a solution. It is concerned with the solution design. The benefit of the structured design is, that it gives a better understanding of how the problem is being solved. The structured design also makes it simpler for a designer to concentrate on the problem more accurately.
Structured design is mostly based on the '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 the problem are solved through solution modules. The structured design emphasizes that these modules be well organized to achieve a precise solution.
These modules are arranged in a hierarchy. They communicate with each other. A well-structured design always follows some rules for communication among multiple modules, namely -

Cohesion - a grouping of all functionally related elements.
Coupling - communication between different modules.

A well-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 a significant task in the system. The system is considered as a 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 its operation. These functional modules can share information among themselves using 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 employing a data flow diagram.
- DFD depicts how functions change data and the state of the entire system.
- The entire system is logically broken down into smaller units known as functions based on 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 strategy focuses on entities and their characteristics. The whole concept of software solutions 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, people, banks, companies, and customers are treated as objects. Every entity has some attributes associated with 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 define the functionality of the object. In the solution design, attributes are stored as variables, and functionalities are defined through 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 to the data and methods from the outside world. This is called information hiding.
- Inheritance - OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and reuse allowed variables and
- methods from their immediate superclasses. This property of OOD is known as inheritance. This makes it easier to define a 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 the same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, a respective portion of the code gets executed.

Design Process

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

- A solution design is created from a requirement or previously 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.
- The application framework is defined.

Software Design Approaches

Here are two general approaches for software design:

Top-Down Design

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

The top-down design takes the whole software system as one entity and then decomposes it to achieve more than one subsystem 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 the system in the top-down hierarchy is achieved. Top-down design starts with a generalized model of a 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 the most specific and basic components. It proceeds with composing higher levels 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. A 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.