Software design

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

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

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

Software Design Levels

Software design yields three levels of results:

- **Architectural Design** - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design**- The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
- **Detailed Design**- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

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 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 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

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

The Objectives of the Design Phase

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes ―good‖ software, and an overall commitment to quality are critical success factors for a competent design. The design model is the equivalent of

an architect‘s plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process. Davis [DAV95] suggests a set of principles for software design, which have been adapted and extended in the following list:

- **The design process should not suffer from “tunnel vision.”** A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job.
- **The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- **The design should not reinvent the wheel.** Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
- **The design should “minimize the intellectual distance” between the software and the problem as it exists in the real world.** That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- **The design should exhibit uniformity and integration.** A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
- **The design should be structured to accommodate change.** The design concepts discussed in the next section enable a design to achieve this principle.
- **The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.** Well- designed software should never ―bomb.‖ It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
- **Design is not coding, coding is not design.** Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.
- **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts and design measures are available to assist the designer in assessing quality.
- **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency) have been addressed before worrying about the syntax of the design model.

Design is essentially the bridge between requirements specification and the final solution for satisfying the requirements. The goal of the design process is to produce a model or representation of a system, which can be used later to build that system. The produced model is called the design of the system. The design of a system is essentially a blueprint or a plan for a solution for the system. The design process for software systems often has two levels.

At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected. This is what is called the system design or top-level design.

In the second level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided.

This design level is often called detailed design or logic design. Detailed design essentially expands to the system design to contain a more detailed description of the processing logic and data structures so that the design is sufficiently complete for coding. A design methodology is a systematic approach to creating a design by applying a set of techniques and guidelines. Most design methodologies essentially offer a set of guidelines that can be used by the developer to design a system.

The input to the design phase is the specifications for the system to be designed. Hence a reasonable entry criteria can be that the specifications are stable and have been approved, hoping that the approval mechanism will ensure that the specifications are complete, consistent, unambiguous, etc. The output of the top-level design phase is the architectural design or the system design for the software system to be built.

A design can be object-oriented or function-oriented. In function-oriented design the design consists of module definitions, with each module supporting a functional abstraction. In object-oriented design, the modules in the design represent data abstraction.

The purpose of the design phase is to specify the components for this transformation function, so that each component is also a transformation function. Hence, the basic output of the system design phase, when a function oriented design approach is being followed, is the definition of all the major data structures in the system, all the major modules of the system, and how the modules interact with each other.

Principles of Software Design & Concepts in Software Engineering

Once the requirements document for the software to be developed is available, the software design phase begins. While the requirement specification activity deals entirely with the problem domain, design is the first phase of transforming the problem into a solution. In the design phase, the customer and business requirements and technical considerations all come together to formulate a product or a system.

The design process comprises a set of principles, concepts and practices, which allow a software engineer to model the system or product that is to be built. This model, known as design model, is assessed for quality and reviewed before a code is generated and tests are conducted. The design model provides details about software data structures, architecture, interfaces and components which are required to implement the system. This chapter discusses the design elements that are required to develop a software design model. It also discusses the design patterns and various software design notations used to represent a software design.

Basic of Software Design

Software design is a phase in software engineering, in which a blueprint is developed to serve as a base for constructing the software system. IEEE defines software design as 'both a process of defining, the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.'

In the design phase, many critical and strategic decisions are made to achieve the desired functionality and quality of the system. These decisions are taken into account to successfully develop the software and carry out its maintenance in a way that the quality of the end product is improved.

Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice.

Some of the commonly followed design principles are as following.

1. **Software design should correspond to the analysis model:** Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.

2. Choose the right programming paradigm: A programming paradigm describes the structure of the software system. Depending on the nature and type of application, different programming paradigms such as procedure oriented, object-oriented, and prototyping paradigms can be used. The paradigm should be chosen keeping constraints in mind such as time, availability of resources and nature of user's requirements.

3. Software design should be uniform and integrated: Software design is considered uniform and integrated, if the interfaces are properly defined among the design components. For this, rules, format, and styles are established before the design team starts designing the software.

4. Software design should be flexible: Software design should be flexible enough to adapt changes easily. To achieve the flexibility, the basic design concepts such as abstraction, refinement, and modularity should be applied effectively.

5. Software design should ensure minimal conceptual (semantic) errors: The design team must ensure that major conceptual errors of design such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.

6. Software design should be structured to degrade gently: Software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.

7. Software design should represent correspondence between the software and real-world problem: The software design should be structured in such away that it always relates with the real-world problem.

8. Software reuse: Software engineers believe on the phrase: 'do not reinvent the wheel'. Therefore, software components should be designed in such a way that they can be effectively reused to increase the productivity.

9. Designing for testability: A common practice that has been followed is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed (designed and implemented) and then handed over to the testers who subsequently determine whether the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as if any type of design or implementation errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should be involved with analysts to prepare tests for determining whether the user requirements are being met.

10. Prototyping: Prototyping should be used when the requirements are not completely defined in the beginning. The user interacts with the developer to expand and refine the requirements as the development proceeds. Using prototyping, a quick 'mock-up' of the system can be developed. This mock-up can be used as a effective means to give the users a feel of what the system will look like and demonstrate functions that will be

included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer's requirements.

Note that design principles are often constrained by the existing hardware configuration, the implementation language, the existing file and data structures, and the existing organizational practices. Also, the evolution of each software design should be meticulously designed for future evaluations, references and maintenance.

Software Design Concepts

Every software process is characterized by basic concepts along with certain practices or methods. Methods represent the manner through which the concepts are applied. As new technology replaces older technology, many changes occur in the methods that are used to apply the concepts for the development of software. However, the fundamental concepts underlining the software design process remain the same, some of which are described here.

Abstraction

Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components. IEEE defines abstraction as 'a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.' The concept of abstraction can be used in two ways: as a process and as an entity. As a process, it refers to a mechanism of hiding irrelevant details and representing only the essential features of an item so that one can focus on important things at a time. As an entity, it refers to a model or view of an item.

Each step in the software process is accomplished through various levels of abstraction. At the highest level, an outline of the solution to the problem is presented whereas at the lower levels, the solution to the problem is presented in detail. For example, in the requirements analysis phase, a solution to the problem is presented using the language of problem environment and as we proceed through the software process, the abstraction level reduces and at the lowest level, source code of the software is produced.

There are three commonly used abstraction mechanisms in software design, namely, functional abstraction, data abstraction and control abstraction. All these mechanisms allow us to control the complexity of the design process by proceeding from the abstract design model to concrete design model in a systematic manner.

Functional abstraction: This involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as 'groups'. Within these groups there exist routines which may be visible or hidden. Visible routines can be used within the containing groups as well as within other groups, whereas hidden routines are hidden from other groups and can be used within the containing group only.

Data abstraction: This involves specifying data that describes a data object. For example, the data object window encompasses a set of attributes (window type, window dimension) that describe the window object clearly. In this abstraction mechanism, representation and manipulation details are ignored.

Control abstraction: This states the desired effect, without stating the exact mechanism of control. For example, if and while statements in programming languages (like C and C++) are abstractions of machine code implementations, which involve conditional instructions. In the architectural design level, this abstraction mechanism permits specifications of sequential subprogram and exception handlers without the concern for exact details of implementation.

Architecture

Software architecture refers to the structure of the system, which is composed of various components of a program/ system, the attributes (properties) of those components and the relationship amongst them. The

software architecture enables the software engineers to analyse the software design efficiently. In addition, it also helps them in decision-making and handling risks. The software architecture does the following.

- Provides an insight to all the interested stakeholders that enable them to communicate with each other
- Highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase
- Creates intellectual models of how the system is organized into components and how these components interact with each other.

Currently, software architecture is represented in an informal and unplanned manner. Though the architectural concepts are often represented in the infrastructure (for supporting particular architectural styles) and the initial stages of a system configuration, the lack of an explicit independent characterization of architecture restricts the advantages of this design concept in the present scenario.

Note that software architecture comprises two elements of design model, namely, data design and architectural design.

Patterns

A pattern provides a description of the solution to a recurring design problem of some specific domain in such a way that the solution can be used again and again. The objective of each pattern is to provide an insight to a designer who can determine the following.

1. Whether the pattern can be reused
2. Whether the pattern is applicable to the current project
3. Whether the pattern can be used to develop a similar but functionally or structurally different design pattern.

Types of Design Patterns

Software engineer can use the design pattern during the entire software design process. When the analysis model is developed, the designer can examine the problem description at different levels of abstraction to determine whether it complies with one or more of the following types of design patterns.

1. **Architectural patterns:** These patterns are high-level strategies that refer to the overall structure and organization of a software system. That is, they define the elements of a software system such as subsystems, components, classes, etc. **In** addition, they also indicate the relationship between the elements along with the rules and guidelines for specifying these relationships. Note that architectural patterns are often considered equivalent to software architecture.
2. **Design patterns:** These patterns are medium-level strategies that are used to solve design problems. They provide a means for the refinement of the elements (as defined by architectural pattern) of a software system or the relationship among them. Specific design elements such as relationship among components or mechanisms that affect component-to-component interaction are addressed by design patterns. Note that design patterns are often considered equivalent to software components.
3. **Idioms:** These patterns are low-level patterns, which are programming-language specific. They describe the implementation of a software component, the method used for interaction among software components, etc., in a specific programming language. Note that idioms are often termed as coding patterns.

Modularity

Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as **modules.** A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing the modules, they are integrated together to meet the software requirements. Note that larger the number of modules a system is divided into, greater will be the effort required to integrate the modules.

Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conduct maintenance work without adversely affecting the functioning of the software.

Information Hiding

Modules should be specified and designed in such a way that the data structures and processing details of one module are not accessible to other modules. They pass only that much information to each other, which is required to accomplish the software functions. The way of hiding unnecessary details is referred to as information hiding. IEEE defines information hiding as 'the technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus each module is a 'black box' to the other modules in the system.

Information hiding is of immense use when modifications are required during the testing and maintenance phase. Some of the advantages associated with information hiding are listed below.

- Leads to low coupling
- Emphasizes communication through controlled interfaces
- Decreases the probability of adverse effects
- Restricts the effects of changes in one component on others
- Results in higher quality software.

Stepwise Refinement

Stepwise refinement is a top-down design strategy used for decomposing a system from a high level of abstraction into a more detailed level (lower level) of abstraction. At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data. As we proceed towards the lower levels of abstraction, more and more details are available.

Software designers start the stepwise refinement process by creating a sequence of compositions for the system being designed. Each composition is more detailed than the previous one and contains more components and interactions. The earlier compositions represent the significant interactions within the system, while the later compositions show in detail how these interactions are achieved.

To have a clear understanding of the concept, let us consider an example of stepwise refinement. Every computer program comprises input, process, and output.

Refactoring

Refactoring is an important design activity that reduces the complexity of module design keeping its behaviour or function unchanged. Refactoring can be defined as a process of modifying a software system to improve the internal structure of design without changing its external behavior. During the refactoring process, the existing design is checked for any type of flaws like redundancy, poorly constructed algorithms and data structures, etc., in order to improve the design. For example, a design model might yield a component which exhibits low cohesion (like a component performs four functions that have a limited relationship with one another). Software designers may decide to refactor the component into four different components, each exhibiting high cohesion. This leads to easier integration, testing, and maintenance of the software components.

Structural Partitioning

When the architectural style of a design follows a hierarchical nature, the structure of the program can be partitioned either horizontally or vertically. In horizontal partitioning, the control modules are used to communicate between functions and execute the functions. Structural partitioning provides the following benefits.

The testing and maintenance of software becomes easier.

The negative impacts spread slowly.

The software can be extended easily.

Besides these advantages, horizontal partitioning has some disadvantage also. It requires to pass more data across the module interface, which makes the control flow of the problem more complex. This usually happens in cases where data moves rapidly from one function to another.

In vertical partitioning, the functionality is distributed among the modules--in a top-down manner. The modules at the top level called control modules perform the decision-making and do little processing whereas the modules at the low level called worker modules perform all input, computation and output tasks.

Concurrency

Computer has limited resources and they must be utilized efficiently as much as possible. To utilize these resources efficiently, multiple tasks must be executed concurrently. This requirement makes concurrency one of the major concepts of software design. Every system must be designed to allow multiple processes to execute concurrently, whenever possible. For example, if the current process is waiting for some event to occur, the system must execute some other process in the mean time.

One way to achieve synchronization is mutual exclusion, which ensures that two concurrent processes do not interfere with the actions of each other. To ensure this, mutual exclusion may use locking technique. In this technique, the processes need to lock the data item to be read or updated. The data item locked by some process cannot be accessed by other processes until it is unlocked. It implies that the process, that needs to access the data item locked by some other process, has to wait.

Developing a Design Model

To develop a complete specification of design (design model), four design models are needed. These models are listed below.

1. **Data design:** This specifies the data structures for implementing the software by converting data objects and their relationships identified during the analysis phase. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design models.
2. **Architectural design:** This specifies the relationship between the structural elements of the software, design patterns, architectural styles, and the factors affecting the ways in which architecture can be implemented.
3. **Component-level design:** This provides the detailed description of how structural elements of software will actually be implemented.
4. **Interface design**: This depicts how the software communicates with the system that interoperates with it and with the end-users.

Software design Methodologies

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.

Data design

Data design is the first design activity, which results in less complex, modular and efficient program structure. The information domain model developed during analysis phase is transformed into data structures needed for implementing the software. The data objects, attributes, and relationships depicted in entity relationship diagrams and the information stored in data dictionary provide a base for data design activity. During the data design process, data types are specified along with the integrity rules required for the data. For specifying and designing efficient data structures, some principles should be followed. These principles are listed below.

1. The data structures needed for implementing the software as well-as the operations that can be applied on them should be identified.
2. A data dictionary should be developed to depict how different data objects interact with each other and what constraints are to be imposed on the elements of data structure.
3. Stepwise refinement should be used in data design process and detailed design decisions should be made later in the process.
4. Only those modules that need to access data stored in a data structure directly should be aware of the representation of the data structure.
5. A library containing the set of useful data structures along with the operations that can be performed on them should be maintained.
6. Language used for developing the system should support abstract data types.

The structure of data can be viewed at three levels, namely, program component level, application level, and business level. At the program component level, the design of data structures and the algorithms required to manipulate them is necessary, if high-quality software is desired. At the application level, it is crucial to convert the data model into a database so that the specific business objectives of a system could be achieved. At the business level, the collection of information stored in different databases should be reorganized into data warehouse, which enables data mining that has an influential impact on the business.

Architectural Design in Software Engineering

Requirements of the software should be transformed into an architecture that describes the software's top-level structure and identifies its components. This is accomplished through architectural design (also called system design), which acts as a preliminary 'blueprint' from which software can be developed. IEEE defines architectural design as 'the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.' This framework is established by examining the software requirements document and designing a model for providing implementation details. These details are used to specify the components of the system along with their inputs, outputs, functions, and the interaction between them. An architectural design performs the following functions.

1. It defines an abstraction level at which the designers can specify the functional and performance behaviour of the system.

2. It acts as a guideline for enhancing the system (when ever required) by describing those features of the system that can be modified easily without affecting the system integrity.

3. It evaluates all top-level designs.

4. It develops and documents top-level design for the external and internal interfaces.

5. It develops preliminary versions of user documentation.

6. It defines and documents preliminary test requirements and the schedule for software integration.

7. The sources of architectural design are listed below.

8. Information regarding the application domain for the software to be developed

9. Using data-flow diagrams

10. Availability of architectural patterns and architectural styles.

Architectural design is of crucial importance in software engineering during which the essential requirements like reliability, cost, and performance are dealt with. This task is cumbersome as the software engineering paradigm is shifting from monolithic, stand-alone, built-from-scratch systems to componentized, evolvable, standards-based, and product line-oriented systems. Also, a key challenge for designers is to know precisely how to proceed from requirements to architectural design. To avoid these problems, designers adopt strategies such as reusability, componentization, platform-based, standards-based, and so on.

Though the architectural design is the responsibility of developers, some other people like user representatives, systems engineers, hardware engineers, and operations personnel are also involved. All these stakeholders must also be consulted while reviewing the architectural design in order to minimize the risks and errors.

Architectural Design Representation

Architectural design can be represented using the following models.

1. Structural model: Illustrates architecture as an ordered collection of program components

2. Dynamic model: Specifies the behavioural aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment

3. Process model: Focuses on the design of the business or technical process, which must be implemented in the system

4. Functional model: Represents the functional hierarchy of a system

5. Framework model: Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.

Architectural Design Output

The architectural design process results in an Architectural Design Document (ADD). This document consists of a number of graphical representations that comprises software models along with associated descriptive text. The software models include static model, interface model, relationship model, and dynamic process model. They show how the system is organized into a process at run-time.

Architectural design document gives the developers a solution to the problem stated in the Software Requirements Specification (SRS). Note that it considers only those requirements in detail that affect the program structure. In addition to ADD, other outputs of the architectural design are listed below.

1. Various reports including audit report, progress report, and configuration status accounts report

2. Various plans for detailed design phase, which include the following

3. Software verification and validation plan

4. Software configuration management plan

5. Software quality assurance plan

6. Software project management plan.

Architectural Styles

Architectural styles define a group of interlinked systems that share structural and semantic properties. In short, the objective of using architectural styles is to establish a structure for all the components present in a system. If an existing architecture is to be re-engineered, then imposition of an architectural style results in fundamental changes in the structure of the system. This change also includes re-assignment of the functionality performed by the components.

By applying certain constraints on the design space, we can make different style-specific analysis from an architectural style. In addition, if conventional structures are used for an architectural style, the other stakeholders can easily understand the organization of the system.

A computer-based system (software is part of this system) exhibits one of the many available architectural styles. Every architectural style describes a system category that includes the following.

1. Computational components such as clients, server, filter, and database to execute the desired system function

2. A set of connectors such as procedure call, events broadcast, database protocols, and pipes to provide communication among the computational components

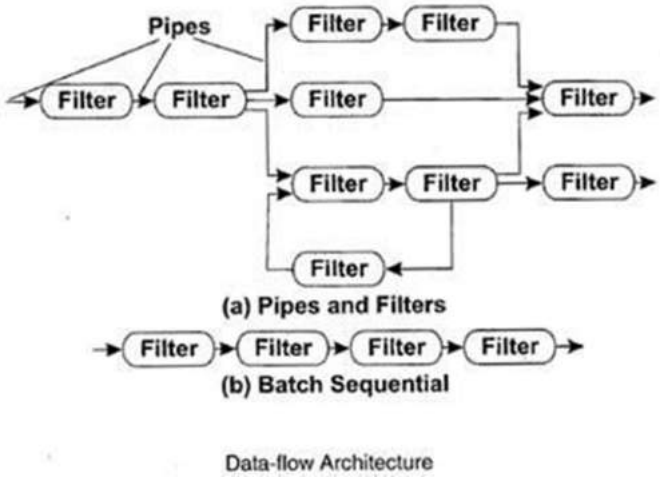
3. Constraints to define integration of components to form a system

4. A semantic model, which enable the software designer to identify the characteristics of the system as a whole by studying the characteristics of its components.

Some of the commonly used architectural styles are data-flow architecture, object oriented architecture, layered system architecture, data-centered architecture, and call and return architecture. Note that the use of an appropriate architectural style promotes design reuse, leads to code reuse, and supports interoperability.

Data-flow Architecture

Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations. Each component, known as filter, transforms the data and sends this transformed data to other filters for further processing using the connector, known as pipe. Each filter works as an independent entity, that is, it is not concerned with the filter which is producing or consuming the data. A pipe is a unidirectional channel which transports the data received on one end to the other end. It does not change the data in anyway; it merely supplies the data to the filter on the receiver end.



Data-flow Architecture

Most of the times, the data-flow architecture degenerates a batch sequential system. In this system, a batch of data is accepted as input and then a series of sequential filters are applied to transform this data. One common example of this architecture is UNIX shell programs. In these programs, UNIX processes act as filters and the file system through which UNIX processes interact, act as pipes. Other well-known examples of this architecture are compilers, signal processing systems, parallel programming, functional programming, and distributed systems. Some advantages associated with the data-flow architecture are listed below.

1. It supports reusability.
2. It is maintainable and modifiable.
3. It supports concurrent execution.
4. Some disadvantages associated with the data-flow architecture are listed below.
5. It often degenerates to batch sequential system.
6. It does not provide enough support for applications requires user interaction.
7. It is difficult to synchronize two different but related streams.

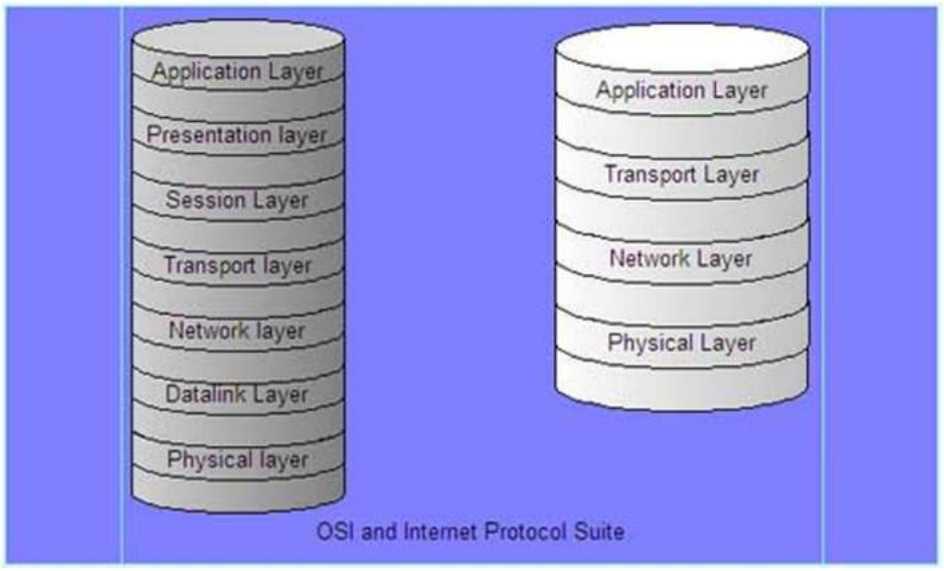
Object-oriented Architecture

In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data. In this style, components are represented as objects and they interact with each other through methods (connectors). This architectural style has two important characteristics, which are listed below.

1. Objects maintain the integrity of the system.
2. An object is not aware of the representation of other objects.
3. Some of the advantages associated with the object-oriented architecture are listed below.
4. It allows designers to decompose a problem into a collection of independent objects.
5. The implementation detail of objects is hidden from each other and hence, they can be changed without affecting other objects.

Layered Architecture

In layered architecture, several layers (components) are defined with each layer performing a well-defined set of operations. These layers are arranged in a hierarchical manner, each one built upon the one below it. Each layer provides a set of services to the layer above it and acts as a client to the layer below it. The interaction between layers is provided through protocols (connectors) that define a set of rules to be followed during interaction. One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organization for Standardization) communication system.



Data-centered Architecture

A data-centered architecture has two distinct components: a central data structure or data store (central repository) and a collection of client software. The datastore (for example, a database or a file) represents the current state of the data andthe client software performs several operations like add, delete, update, etc., onthe data stored in the data store. In some cases, the data storeallows the client software to access the data independent of any changes or theactions of other client software.

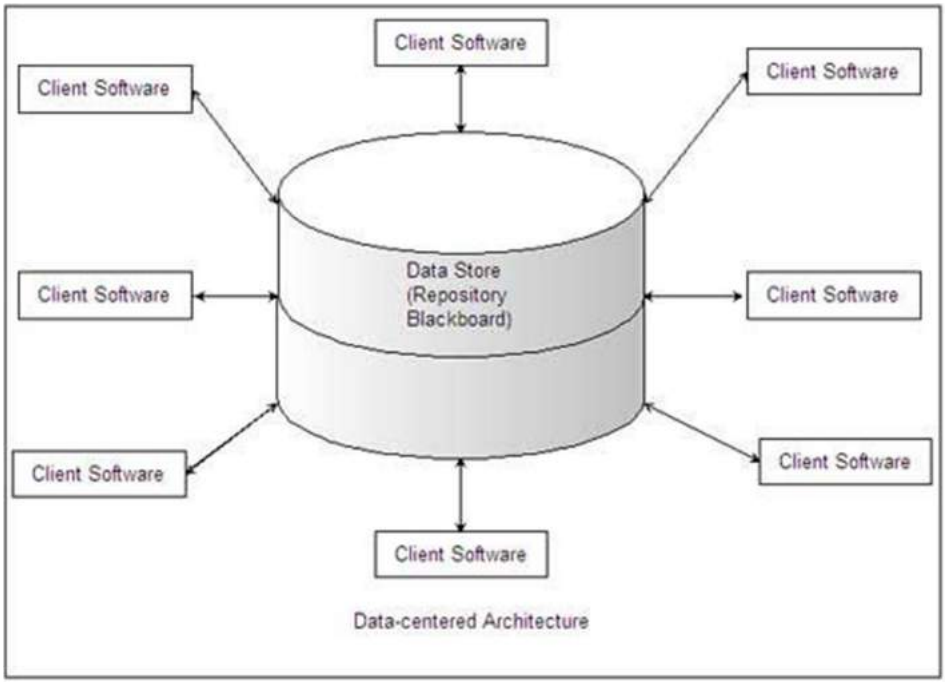
In this architectural style, new components corresponding to clients can be added and existing components can be modified easily without taking into account other clients. This is because client components operate independently of one another.

A variation of this architectural style is blackboard system in which the data store is transformed into a blackboard that notifies the client software when the data (of their interest) changes. In addition, the information can be transferred among the clients through the blackboard component.

Some advantages of the data-centered architecture are listed below.

1. Clients operate independently of one another.
2. Data repository is independent of the clients.
3. It adds scalability (that is, new clients can be added easily).
4. It supports modifiability.

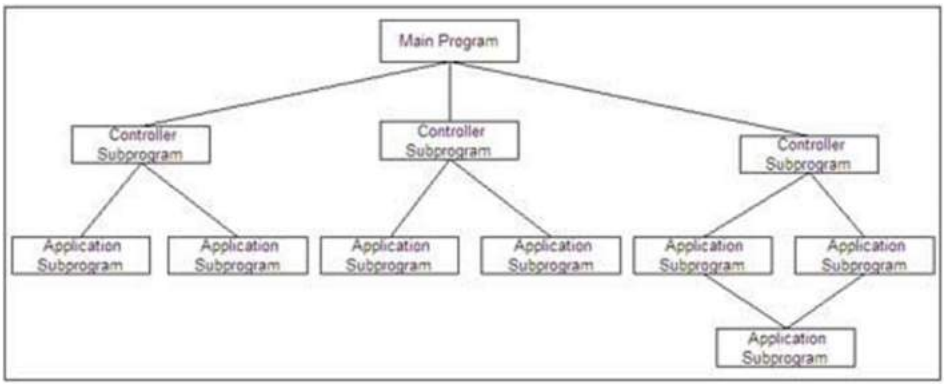
It achieves data integration in component-based development using blackboard.



Call and Return Architecture

A call and return architecture enables software designers to achieve a program structure, which can be easily modified. This style consists of the following two substyles.

Main program/subprogram architecture: In this, function is decomposed into a control hierarchy where the main program invokes a number of program components, which in turn may invoke other components.



1. Remote procedure call architecture: In this, components of the main or subprogram architecture are distributed over a network across multiple computers.

Procedural Design

The *procedural design* is often understood as a software design process that uses mainly control commands such as: sequence, condition, repetition, which are applied to the predefined data.

Sequences serve to achieve the processing steps in order that is essential in the specification of any algorithm.

Conditions provide facilities for achieving selected processing according to some logical statement.

Repetitions serve to achieve loopings during the computation process.

These three commands are implemented as ready programming language constructs.

The programming languages that provide such command constructs are called *imperative programming languages*.

The software design technique that relies on these constructs is called *procedural design*, or also *structured design*.