Software Design (Unit-II)

Software Design

Software design is a process of problem-solving and planning for a software solution. A software design creates meaningful engineering representation (or model) of software product that is to be built.

- ✓ Encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product
- ✓ Design principles establish and overriding philosophy that guides the designer as the work is performed
- ✓ Design concepts must be understood before the mechanics of design practice are applied
- ✓ Goal of design engineering is to produce a model or representation that is bug free (firmness), suitable for its intended uses (commodity), and pleasurable to use (delight)
- ✓ Software design practices change continuously as new methods, better analysis, and broader understanding evolve

Design Principles

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. The design model is the equivalent of an architect's plans for a house. 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.

- ✓ 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, and the design concepts.
- ✓ The design should be traceable to the analysis model. It should provide means of checking whether the design elements have the ability to meet and satisfy requirements identified in the analysis phase.

- ✓ The design should not reinvent the wheel. If reusable design components are available within a system, then they must be used instead of wasting precious time and efforts involved in recreating them.
- ✓ 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. If there are any changes in system requirements, the design should be able to integrate those changed requirements easily. In fact, the design should be maintainable at all stages and it can be adapted to modify existing functions and add new functions.
- ✓ The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered. 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 reviewed to minimize conceptual (semantic) errors. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

Design Concepts

The fundamental design concepts are:

Abstraction - allows designers to focus on solving a problem without being concerned about irrelevant lower level details (procedural abstraction - named sequence of events, data abstraction - named collection of data objects). At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Finally, at the lowest level of abstraction,

the solution is stated in a manner that can be directly implemented. As we move through different levels of abstraction, we work to create procedural and data abstractions.

A procedural abstraction is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A data abstraction is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

Refinement - process of elaboration where the designer provides successively more detail for each design component. Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs. Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low level details. Refinement helps the designer to reveal low level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

Modularity - the degree to which software can be understood by examining its components independently of one another. Software architecture embodies modularity; that is, software is divided into separately named and addressable components, often called modules that are integrated to satisfy problem requirements. Software engineering modularity allows typical applications to be divided into modules, as well as integration with similar modules, which helps developers use prewritten code. Modules are divided based on functionality, and programmers are not involved with the functionalities of other modules. Thus, new functionalities may be easily programmed in separate modules.

Software architecture - overall structure of the software components and the ways in which that structure provides conceptual integrity for a system. Software architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enable a software engineer to reuse design level concepts.

The architectural design can be represented using one or more of a number of different models:

Structural models: represent architecture as an organized collection of program components.

Framework models: increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.

Dynamic models: address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models: focus on the design of the business or technical process that the system must accommodate.

Functional models: can be used to represent the functional hierarchy of a system.

A number of different architectural description languages (ADLs) have been developed to represent these models. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

Control hierarchy or program structure - Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

The control hierarchy represents two subtly different characteristics of the software architecture: visibility and connectivity. Visibility indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All of the objects are visible to the module. Connectivity indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution is connected to it.

Structural partitioning – Program structure partitioned horizontally and vertically. Horizontal partitioning defines three partitions (input, data transformations/ processing and output); Partitioning the architecture horizontally provides a number of distinct benefits:

- ✓ results in software that is easier to test
- ✓ leads to software that is easier to maintain
- ✓ results in propagation of fewer side effects
- ✓ results in software that is easier to extend

Vertical partitioning, often called factoring, suggests that control (decision making) and work should be distributed top-down in the program structure. Top level modules should perform control functions and do-little actual processing work. Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

Data structure - Data structure is a representation of the logical relationship among individual elements of data. Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. There are limited numbers of classic data structures that form the building blocks for more sophisticated structures.

A scalar item is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer or floating-point number that is 8 to 64 bits long, or a character string that is hundreds or thousands of bytes long. When scalar items are organized as a list or contiguous group, a sequential vector is

formed. When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an n-dimensional space is created. The most common n- dimensional space is the two -dimensional matrix. In many programming languages, an n-dimensional space is called an array. Items, vectors, and spaces may be organized in a variety of formats. A linked list is a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner (called nodes) that enables them to be processed as a list. Each node contains the appropriate data organization (e.g., a vector) and one or more pointers that indicate the address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.

Software procedure - Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure. There is, of course, a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described.

Information hiding - The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Types of Design

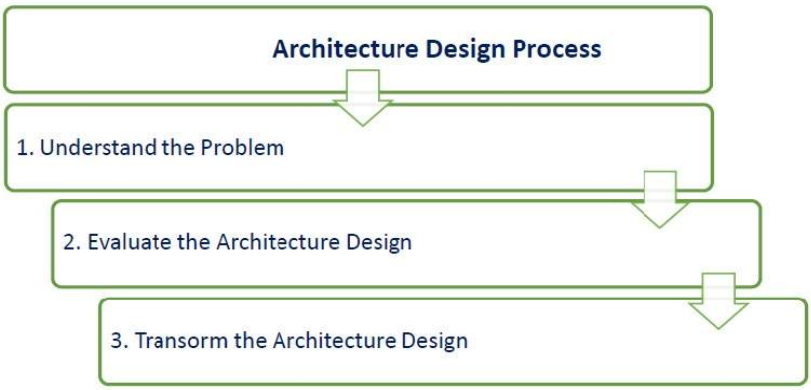
I. Architectural Design

It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components. The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

- ✓ Analyze the effectiveness of the design in meeting its stated requirements,
- ✓ Consider architectural alternatives at a stage when making design changes is still relatively easy, and
- ✓ Reduce the risks associated with the construction of the software.

The architecture design process focuses on the decomposition of a system into different components and their interactions to satisfy functional and nonfunctional requirements. The key inputs to software architecture design are –

- ✓ The requirements produced by the analysis tasks.
- ✓ The hardware architecture (the software architect in turn provides requirements to the system architect, who configures the hardware architecture).
- ✓ The result or output of the architecture design process is an architectural description.



The basic architecture design process is composed of the following steps –

1. Understand the Problem

This is the most crucial step because it affects the quality of the design that follows. Without a clear understanding of the problem, it is not possible to create an effective solution. In fact, many software projects and products are considered as unsuccessful because they did not actually solve a valid business problem.

This phase builds a baseline for defining the boundaries and context of the system. Decomposition of the system into its main components is based on the functional requirements. In this step, the first validation of the architecture is done by describing a number of system instances and this step is referred as functionality based architectural design.

2. Evaluate the Architecture Design

It involves evaluating the architecture for conformance to architectural quality attributes requirements. If all the estimated quality attributes are as per the required standard, the architectural design process is finished. If not, then the third phase of software architecture design is entered i.e. architecture transformation. However, if the observed quality attribute does not meet its requirements, then a new design must be created.

3. Transform the Architecture Design

This step is performed after an evaluation of the architectural design. The architectural design must be changed until it completely satisfies the quality attribute requirements. It is concerned with selecting design solutions to improve the quality attributes while preserving the domain functionality.

II. User Interface Design

User interface is the first impression of a software system from the user's point of view. Therefore, any software system must satisfy the requirement of user.

User Interface (UI) mainly performs two functions:

- ✓ Accepting the user's input
- ✓ Displaying the output

A good user interface must attract the user to use the software system without mistakes. It should help the user to understand the software system easily without misleading information. UI has its syntax and semantics. The syntax comprises component types such as textual, icon, button etc. and usability summarizes the semantics of UI. The quality of UI is characterized by its look and feel (syntax) and its usability (semantics).

Design of User Interface

It starts with task analysis which understands the user's primary tasks and problem domain. It should be designed in terms of User's terminology and outset of user's job rather than programmers. Proper or good UI design works from the user's capabilities and limitations not the machines. While designing the UI, knowledge of the nature of the user's work and environment is also critical.

Elements of UI

To perform user interface analysis, the practitioner needs to study and understand four elements −

- ✓ The users who will interact with the system through the interface
- ✓ The tasks that end users must perform to do their work
- ✓ The content that is presented as part of the interface
- ✓ The work environment in which these tasks will be conducted

Steps of UI Design

User interface design is an iterative process, where all the iteration explains and refines the information developed in the preceding steps. General steps for user interface design −

- ✓ Defines user interface objects and actions (operations).
- ✓ Defines events (user actions) that will cause the state of the user interface to change.

- ✓ Indicates how the user interprets the state of the system from information provided through the interface.
- ✓ Describe each interface state as it will actually look to the end user.

Design Considerations of User Interface

User centered

A user interface must be a user-centered product which involves users throughout a product's development lifecycle. The prototype of a user interface should be available to users and feedback from users, should be incorporated into the final product.

Simple and Intuitive

UI provides simplicity and intuitiveness so that it can be used quickly and effectively without instructions. GUI is better than textual UI, as GUI consists of menus, windows, and buttons and is operated by simply using mouse.

Place Users in Control

Do not force users to complete predefined sequences. Give them options—to cancel or to save and return to where they left off. Use terms throughout the interface that users can understand, rather than system or developer terms.

Use progressive disclosure

Always provide easy access to common features and frequently used actions. Hide less common features and actions and allow users to navigate them. Do not try to put every piece of information in one main window. Use secondary window for information that is not key information.

Consistency

UI maintains the consistency within and across product, keep interaction results the same, UI commands and menus should have the same format, command punctuations should be similar. UI should include the mechanism that allows users to recover from their mistakes.

Reduce Users' Memory Load

Do not force users to remember and repeat what the computer should be doing for them. For example, when filling in online forms, customer names, addresses, and telephone numbers should be remembered by the system once a user has entered them, or once a customer record has been opened.

Software Engineering | Architectural Design

The software needs the architectural design to represents the design of software. 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." The software that is built for computer-based systems can exhibit one of these many architectural styles. Each style will describe a system category that consists of:

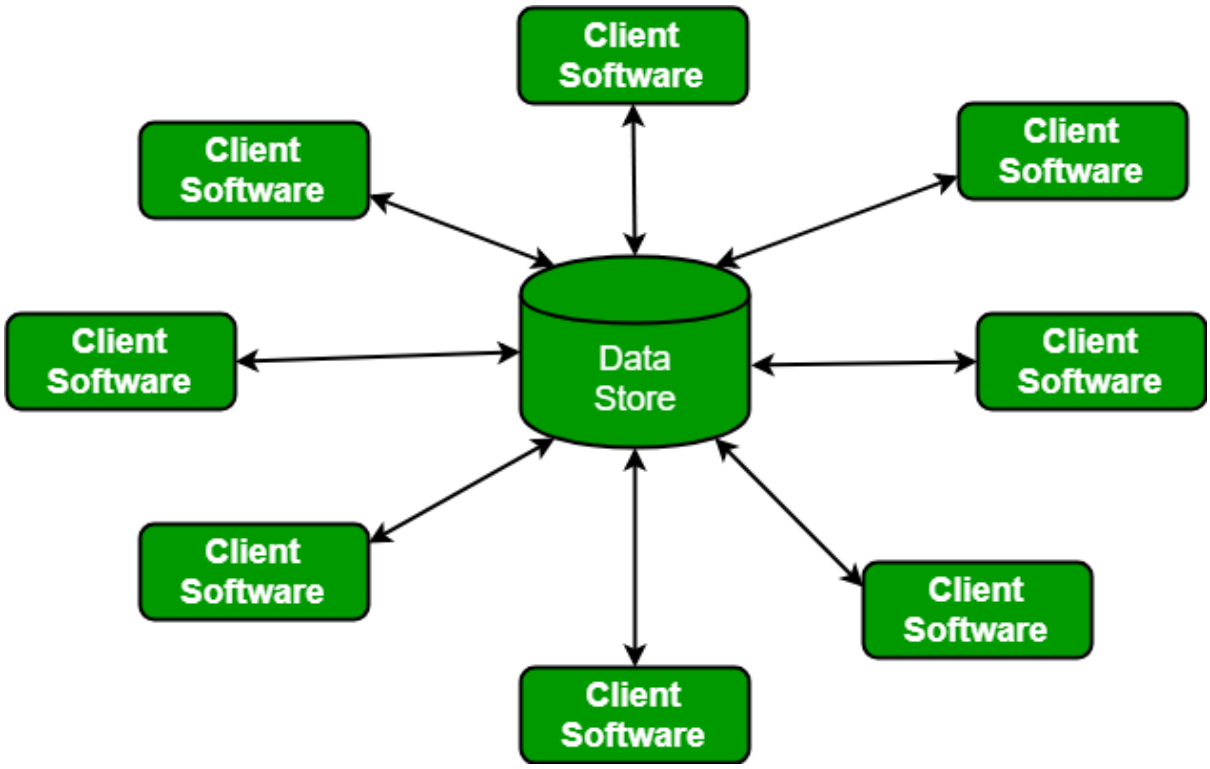
- ✓ A set of components (eg: a database, computational modules) that will perform a function required by the system.
- ✓ The set of connectors will help in coordination, communication, and cooperation between the components.
- ✓ Conditions that how components can be integrated to form the system.
- ✓ Semantic models that help the designer to understand the overall properties of the system.
- ✓ The use of architectural styles is to establish a structure for all the components of the system.

Taxonomy of Architectural styles:

Data centered architectures:

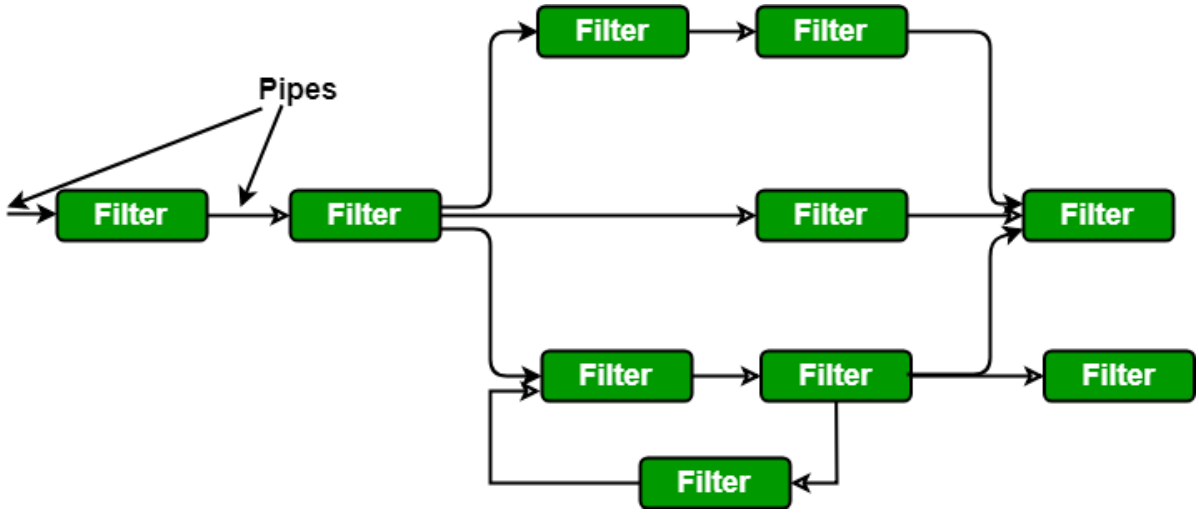
A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store. The figure illustrates a typical data centered style. The client software accesses a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software. This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the

architecture without the permission or concern of other clients. Data can be passed among clients using blackboard mechanism.



Data flow architectures:

This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components. The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes. Pipes are used to transmit data from one component to the next. Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters. If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

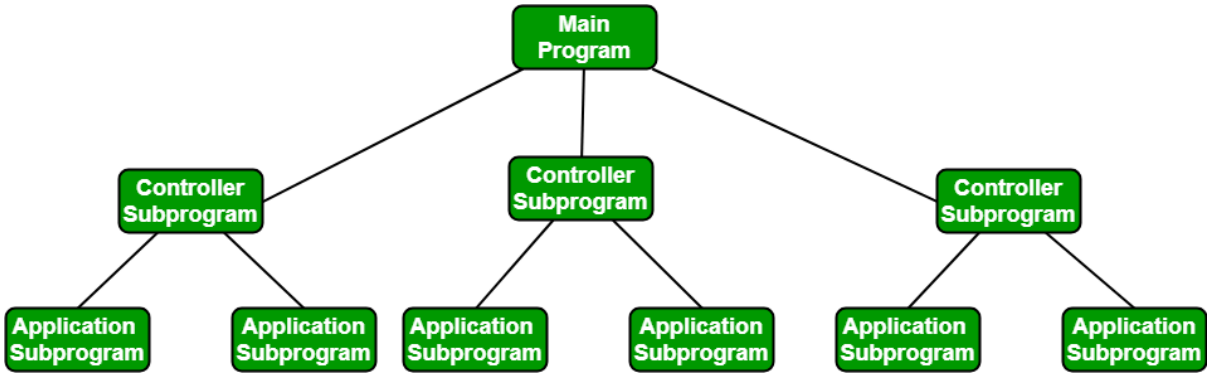


Call and Return architectures: It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

Remote procedure call architecture: This component is used to present in a main program or sub program architecture distributed among multiple computers on a network.

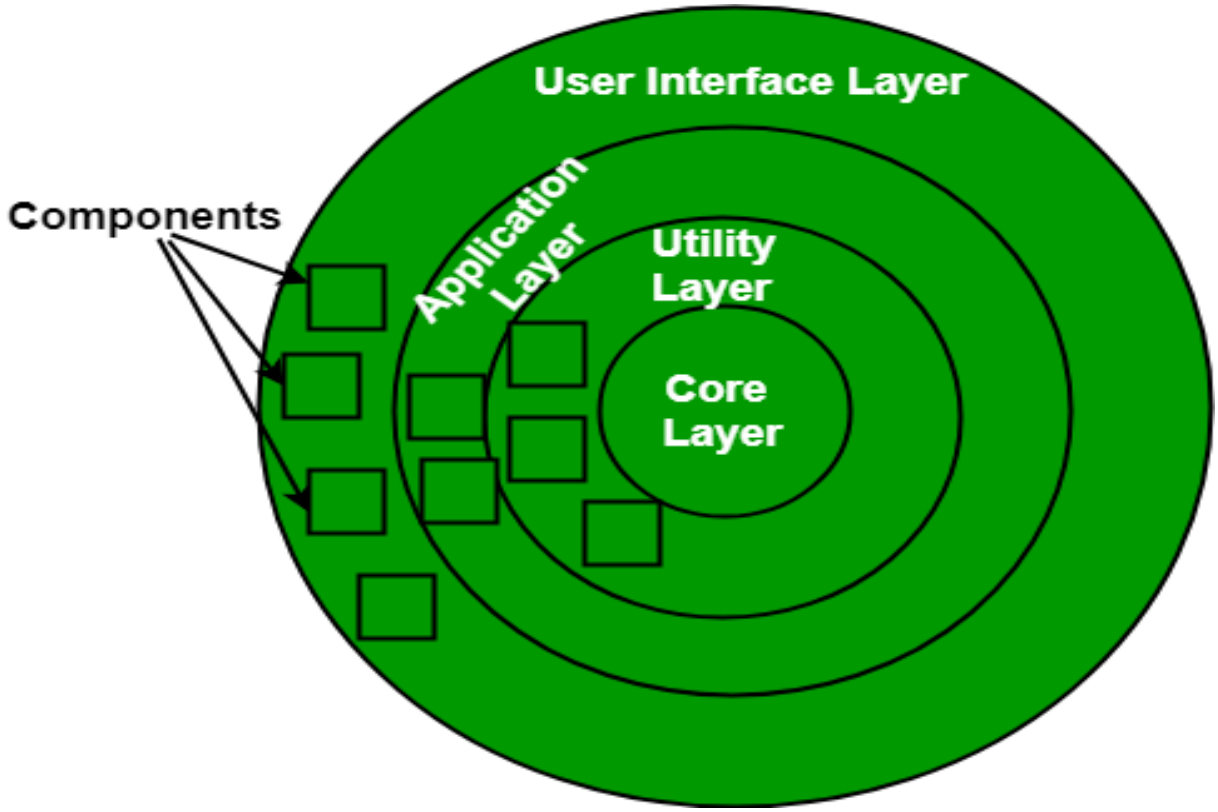
Main program or Subprogram architectures: The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.

Object Oriented architecture: The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.



Layered architecture:

A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively. At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing (communication and coordination with OS) Intermediate layers to utility services and application software functions.

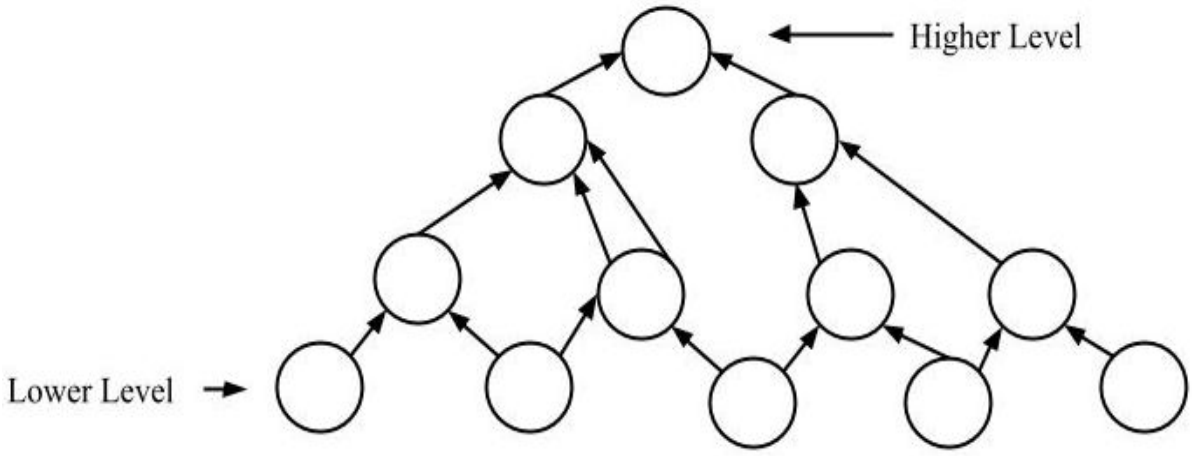


Software Engineering | System Design Strategy

A good system design is to organize the program modules in such a way that are easy to develop and change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. If any pre-existing code needs to be understood, organized and pieced together. It is common for the project team to have to write some code and produce original programs that support the application logic of the system.

Bottom-up approach:

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

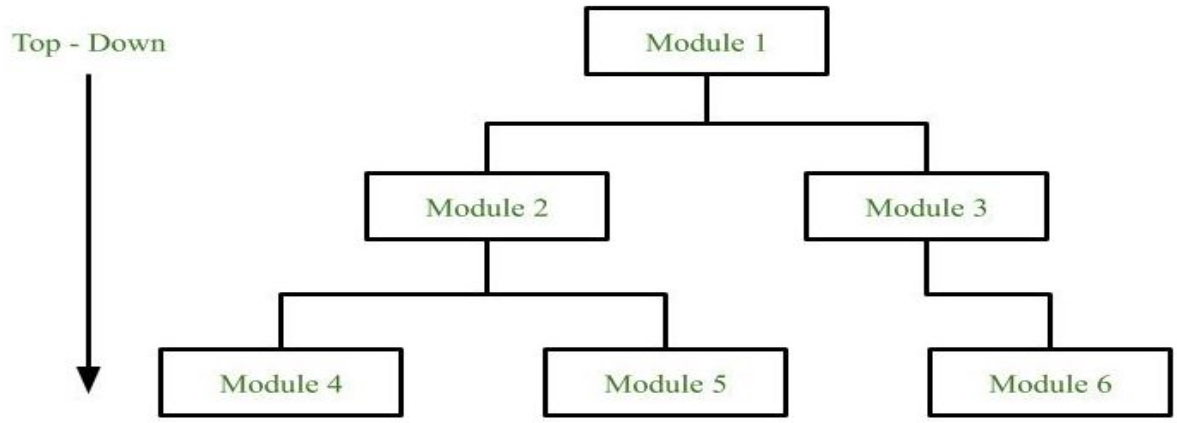


Advantages:

- ✓ The economics can result when general solutions can be reused.
- ✓ It can be used to hide the low-level details of implementation and be merged with top-down technique.
- ✓ Disadvantages:
- ✓ It is not so closely related to the structure of the problem.
- ✓ High quality bottom-up solutions are very hard to construct.
- ✓ It leads to proliferation of 'potentially useful' functions rather than most approprite ones.

Top-down approach:

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



Advantages:

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

Disadvantages:

- ✓ Project and system boundaries tends to be application specification oriented. Thus it is more likely that advantages of component reuse will be missed.
- ✓ The system is likely to miss, the benefits of a well-structured, simple architecture.

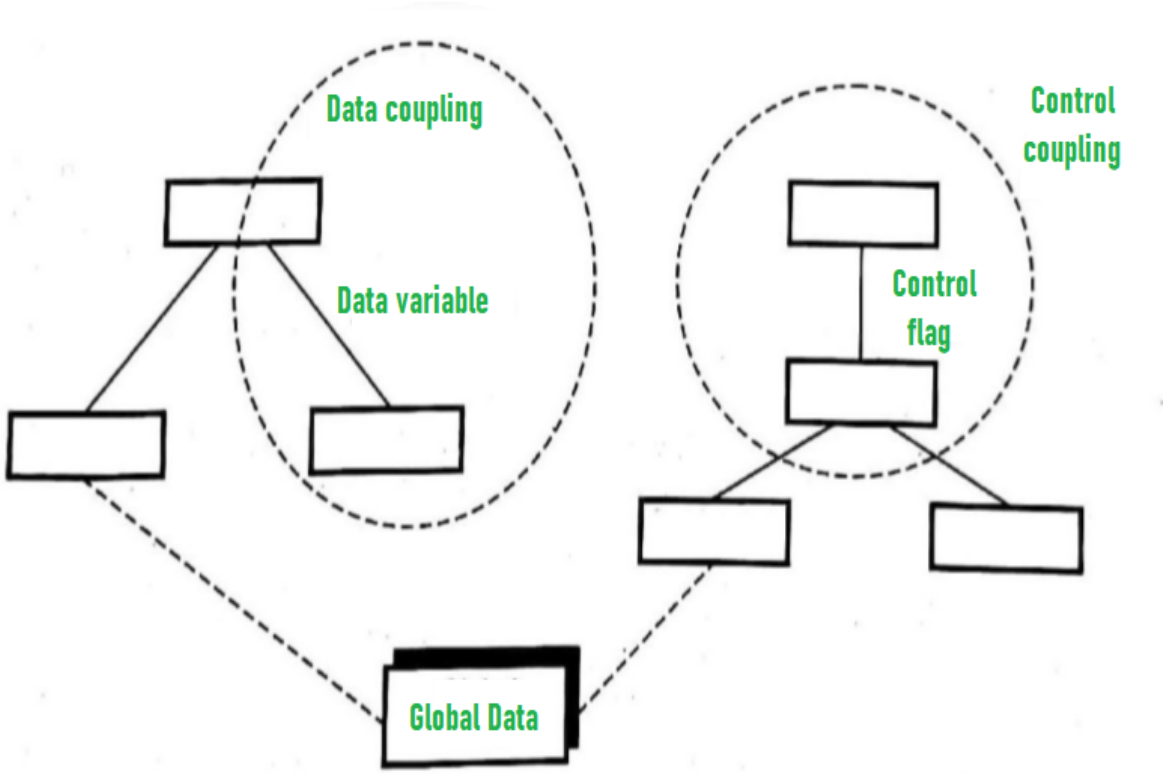
Refactoring

Refactoring is the activity of changing the software's internal structure without changing its external behaviour. It is specifically concerned with improving the software's internal structure. Refactoring provides higher quality software, and eases the work of the software engineer. During refactoring, the software is examined for: • Redundant code: portions of the software that perform the same function should be merged. Having the functionality repeated in multiple areas makes the software harder to maintain. • Unused design elements: if portions of the software are not being used, and removing it will not change the software's behaviour, those portions should not be in the software. • Poorly constructed data structures: these should be modified to improve, for instance, information hiding and functional independence. As with all of the design concepts we have discussed, refactoring should be done to make code easier to understand, easier to develop and test, and easier to change. A good indication that you need to refactor a particular software application is when it has become difficult to either add new functionality to it, or to fix a bug in it.

Module Coupling and Its Types:

Coupling simply means to connect two or more things together. It means the pairing of two things. It actually measures the degree of independence between two things and how closely two things are connected or represent the strength of the relationship between them. Module coupling means to couple two to more modules with each other and with the outside world. It generally represents how the modules are connected with another module and the outside world. Coupling is related to cohesion. Cohesion means that the cohesive module performs only one task or one thing in the overall software procedure with a small amount of interaction with other modules. With the help of cohesion, data hiding can be done. Low coupling correlates with high cohesion and high coupling correlates with low cohesion. Lower will be the coupling, higher will be the cohesion and better will be the program and these programs can be said as functionally independent of other modules.

Low coupling minimizes the "ripple effect (spreading results of any action or error or failure)" which means it reduces the effects of changes in one module that cause errors in other modules. The main aim or goal of module coupling is to make efforts to achieve or obtain the lowest possible coupling among modules in software design. The coupling is said well if it reduces the ripple effect, the cost in program changes, testing, and maintenance.



MODULE COUPLING

These are following types of coupling –

Data Coupling: Data coupling simply means the coupling of data i.e. interaction between data when they are passed through parameters using or when modules share data through parameters. When data of one module is shared with other modules or passed to other modules, this condition is said to be data coupling.

Control Coupling: Control coupling simply means to control data sharing between modules. If the modules interact or connects by sharing controlled data, then they are said to be control coupled. The controlled coupling means that one module controls the flow of data or information by other modules by them the information about what to do.

Common Coupling: Common coupling simply means the sharing of common data or global data between several modules. If two modules share the information through global data items or interact by sharing common data, then they are said to be commonly coupled.

Content Coupling: Content coupling simply means using of data or control information maintained in other modules by one module. This coupling is also known as pathological

coupling. In these coupling, one module relies or depends upon the internal workings of another module. Therefore, if any changes have to be done in the inner working of a module then this will lead to the need for change in the dependent module.

Stamp Coupling: Stamp coupling simply means the sharing of composite data structure between modules. If the modules interact or communicate by sharing or passing day structure that contains more information than the information required to perform their actions, then these modules are said to be stamp coupled.

External Coupling: The external coupling means the sharing of data structure or format that are imposed externally between the modules. External coupling is very important but there should be a limit also. It should be limited to a smaller number of modules with structures.

Design Reuse

Design reuse is the process of building new software applications and tools by reusing previously developed designs. New features and functionalities may be added by incorporating minor changes. Design reuse involves the use of designed modules, such as logic and data, to build a new and improved product. The reusable components, including code segments, structures, plans and reports, minimize implementation time and are less expensive. This avoids reinventing existing software by using techniques already developed and to create and test the software. Design reuse is used in a variety of fields, from software and hardware to manufacturing and aeronautics. Design reuse involves many activities utilizing existing technologies to cater to new design needs. The ultimate goal of design reuse is to help the developers create better products maximizing its value with minimal resources, cost and effort. Today, it is almost impossible to develop an entire product from scratch. Reuse of design becomes necessary to maintain continuity and connectivity. In the software field, the reuse of the modules and data helps save implementation time and increases the possibility of eliminating errors due to prior testing and use. Design reuse requires that a set of designed products already exist and the design information pertaining to the product is accessible. Large software companies usually have a range of designed products. Hence the reuse of design facilitates making new and better software products. Many software companies have incorporated design reuse and have seen considerable success. The effectiveness of design reuse is measured in terms of production, time, cost and quality of the product. These key factors determine whether a company has been successful in making

design reuse a solution to its new software needs and demands. With proper use of existing technology and resources, a company can benefit in terms of cost, time, performance and product quality. A proper process requires an intensive design reuse process model. There are two interrelated process methodologies involved in the systematic design reuse process model.

The data reuse process is as follows:

- ✓ Gathering Information: This involves the collection of information, processing and modeling to fetch related data.
- ✓ Information Reuse: This involves the effective use of data.

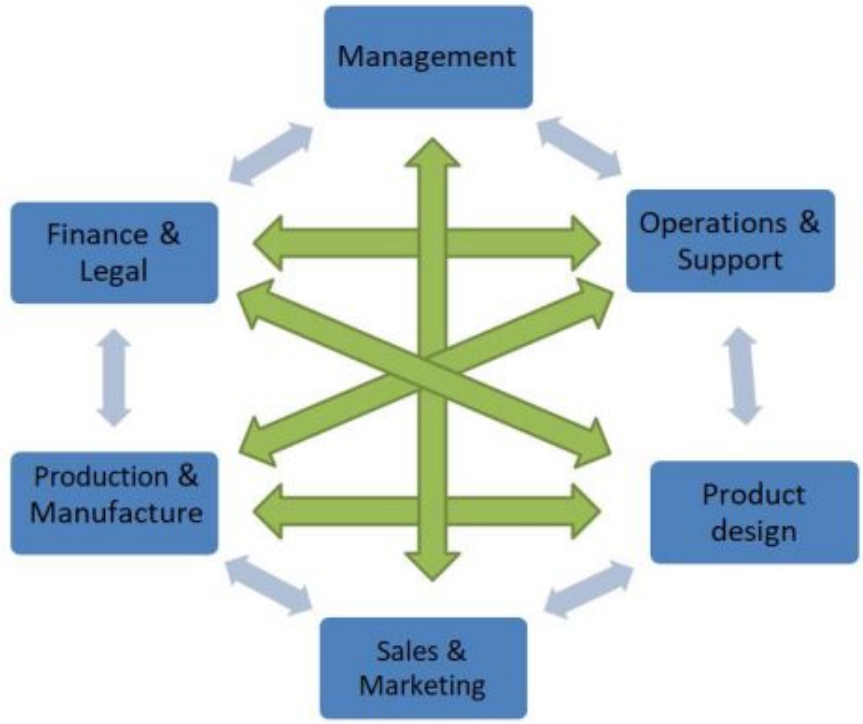
The design reuse process has four major issues:

- ✓ Retrieve
- ✓ Reuse
- ✓ Repair
- ✓ Recover

These are generally referred to as the four Rs. In spite of these challenges, companies have used the design reuse concept as a successfully implemented concept in the software field at different levels, ranging from low level code reuse to high level project reuse.

Concurrent Engineering

Concurrent engineering is a method of designing and developing engineering products, in which different departments work on the different stages of engineering product development simultaneously. If managed well, it helps to increase the efficiency of product development and marketing considerably reducing the time and contributing to the reduction of the overall development cost while improving the final product quality.



Concurrent Engineering interactions

This streamlined approach towards an engineering product forces several teams such as product design, manufacturing, production, marketing, product support, finance, etc., within the organization to work simultaneously on new product development. For instance, while engineering product designers begin to design the product, the sales team can start working on the marketing and the product support department can start thinking about the after-sale support. While the mechanical designers work on the packaging design to incorporate the PCB being developed by the electrical engineering team, the software engineers can start looking at the software code. Concurrent engineering, also known as the integrated product development (IPD) or simultaneous engineering was introduced a few decades ago to eliminate the issues from sequential engineering or so-called "over the wall" process. This systematic approach is intended to force all the stakeholders to be involved and the full engineering product cycle to be considered from concept to after-sale support. There are plenty of incentives to choose Concurrent engineering over sequential engineering product development.

The popularity of integrated product development has been growing recently, thanks to the ever-increasing demand for quality products expeditiously at affordable prices. Although

managing a simultaneous engineering process is very challenging, the techniques and practices followed as part of concurrent engineering benefits from several competitive advantages to the company and to the final engineering product itself.

Elements of concurrent engineering

Concurrent engineering presents an environment which encourages and improves the interaction of different disciplines and departments towards a single goal of satisfying an engineering product requirement. Key elements of concurrent engineering can be summarized using a PPT framework or the Golden Triangle.

People, process & technology framework

People, process, and technology are crucial to any organization and essential in implementing concurrent engineering to achieve shorter development time, lower cost, improved product quality and to fulfil customer needs.

People

Concurrent product development is a multidisciplinary team task and it's necessary that companies utilize the right skilled personnel at the right time to accelerate product development. It is also necessary to find people with the right skills and experience along with the following key aspects;

- ✓ Multidisciplinary team to suit the product at the start of the NPD
- ✓ Teamwork culture at the core of the program
- ✓ Good communication and collaboration between teams – sharing relevant and up to date information across departments and personnel
- ✓ The harmonized goal across the company from the top management to the bottom of the organizational structure

Process

A process is a series of product development steps that need to happen to achieve a goal. These can be project planning stages, milestone management, problem-solving methodologies, product development key stages, information sharing workflow, etc., as just people are ineffective without processes in place to support their tasks and decisions. Following are some of the processes that can be adopted in concurrent engineering;

- ✓ Project planning processes and workflow management which include key new product development elements such as key design stages, milestones for cross-departmental interaction, etc.
- ✓ Workflow for product data management such as sharing information, manage engineering change, control specification creep, etc.
- ✓ Product requirement tracking and checkpoints using techniques such as Quality Function Deployment (QFD) across departments
- ✓ Design evaluation workflow processes
- ✓ Design analysis methodologies such as brainstorming
- ✓ Failure Mode and Effects Analysis (FMEA) allows for a systematic investigation of the occurrence and impact of possible flaws in the new product design
- ✓ The use of Design of Experiments (DOE) enables the systematic identification of critical product/process parameters that influence performance

Technology

For concurrent engineering to be successful, the effective introduction of tools, techniques, and technologies to aid a smooth integration of people and processes is vital. Following key aspects should be considered before any implementation.

- ✓ Identifying the correct tools and technologies that suit the company size, number of team members, processes implemented and product type
- ✓ Identifying the training needs and training people to use the tools and technologies identified above