K.D. K. College of Engineering

Department of Information Technology

B.E. (Information Technology) Fifth Semester (C.B.S.)

Software Engineering

UNIT: 04

Design Engineering Concepts

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.

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.

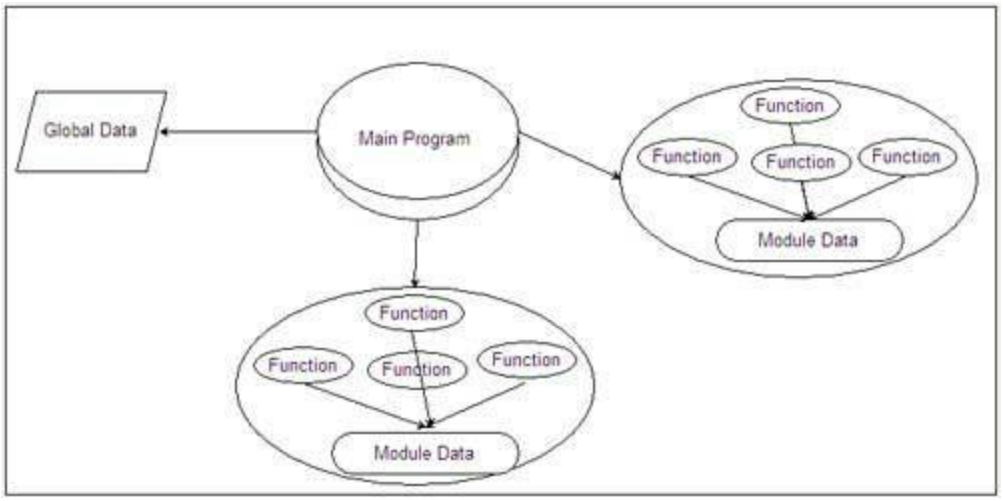
Each step in the software process is accomplished through various levels of abstraction. At the highest level, an outline of the solution to the problem is presented whereas at the lower levels, the solution to the problem is presented in detail. For example, in the requirements analysis phase, a solution 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.

1. **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.
2. **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.
3. **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.

Modularity

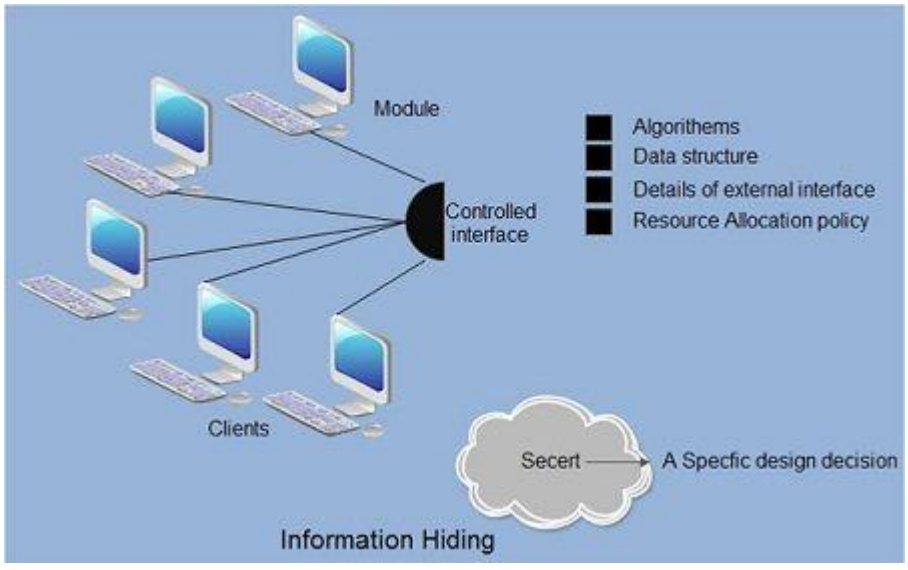
Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as **modules.** A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing 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.

1. Leads to low coupling
2. Emphasizes communication through controlled interfaces

3. Decreases the probability of adverse effects
4. Restricts the effects of changes in one component on others
5. 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.

1. INPUT

- Get user's name (string) through a prompt.
- Get user's grade (integer from 0 to 100) through a prompt and validate.

2. PROCESS
3. OUTPUT

This is the first step in refinement. The input phase can be refined further as given here.

1. INPUT
   - Get user's name through a prompt.
   - Get user's grade through a prompt.
   - While (invalid grade)

   Ask again:

2. PROCESS
3. OUTPUT

Note: Stepwise refinement can also be performed for PROCESS and OUTPUT phase.

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

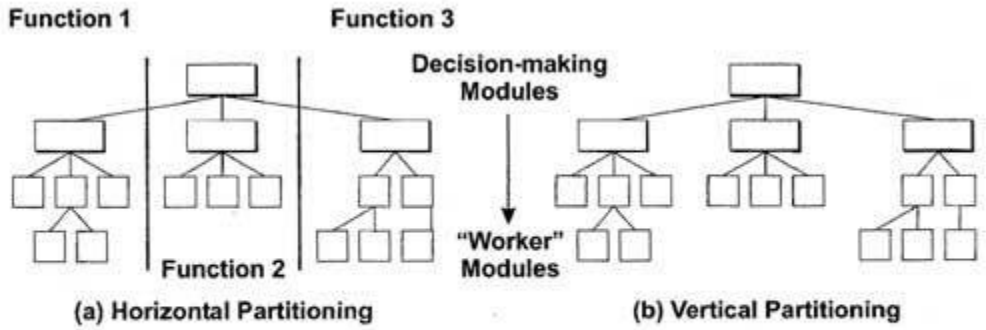
high cohesion. This leads to easier integration, testing, and maintenance of the software components.

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.



Horizontal and Vertical Partitioning

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.

However, concurrent execution of multiple processes sometimes may result in undesirable situations such as an inconsistent state, deadlock, etc. For example, consider two processes A and B and a data item Q1 with the value '200'. Further, suppose A and B are being executed concurrently and firstly A reads the value of Q1 (which is '200') to add '100' to it. However, before A updates es the value of Q1, B reads the value ofQ1 (which is still '200') to add '50' to it.

In this situation, whether A or B first updates the value of Q1, the value of would definitely be wrong resulting in an inconsistent state of the system. This is because the actions of A and B are not synchronized with each other. Thus, the system must control the concurrent execution and synchronize the actions of concurrent processes.

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.

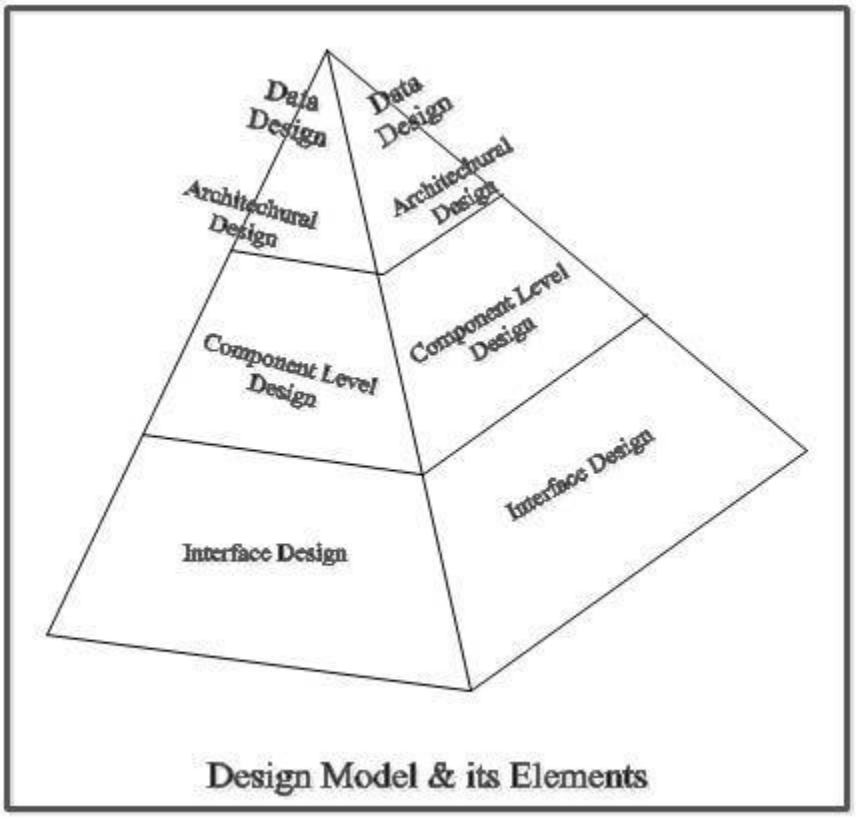
Developing a Design Model

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.

Design Model

A **design model** in Software Engineering is an object-based picture or pictures that represent the use cases for a system. Or to put it another way, it is the means to describe a system's implementation and source code in a diagrammatic fashion. This type of representation has a couple of advantages. First, it is a simpler representation than words alone. Second, a group of people can look at these simple diagrams and quickly get the general idea behind a system. In the end, it boils down to the old adage, 'a picture is worth a thousand words.'



Design Model & its Elements

Moving from requirements (what is needed) to design specifications (a description of what should be built) involves a number of steps. These can be different depending on who you talk to, but we'll use the following:

- Architectural - This involves breaking the system down into its major functional pieces and describing each in a diagrammatic fashion. The interaction between each piece is also described. For example, an instant messaging system might have a 'send message' piece and an interaction might be the user types characters that act as input to the send message piece.
- Interface - The architectural interactions are broken down and described in greater detail. For example, the user typed characters mentioned above might be described as arriving a character at a time, each character is displayed on the cell phone screen as it is typed.
- Component Level - The pieces described in the architectural item above are broken down into lower level components. For example, the send message piece might become 'receive character' component, 'display character' component, and 'transmit character' component.
- Deployment Level - The components arrived at during the previous step are grouped for the purpose of delivery to their final destination. For example, the components of the 'send message' piece might be grouped together with other pieces for deployment on a cell phone.

Developing a Design Model

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

2. **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.

   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.

   a) The data structures needed for implementing the software as well-as the operations that can be applied on them should be identified.
   b) 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.
   c) Stepwise refinement should be used in data design process and detailed design decisions should be made later in the process.
   d) Only those modules that need to access data stored in a data structure directly should be aware of the representation of the data structure.
   e) A library containing the set of useful data structures along with the operations that can be performed on them should be maintained.
   f) 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.

3. **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.

   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.

- It defines an abstraction level at which the designers can specify the functional and performance behaviour of the system.
- 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.
- It evaluates all top-level designs.
- It develops and documents top-level design for the external and internal interfaces.
- It develops preliminary versions of user documentation.
- It defines and documents preliminary test requirements and the schedule for software integration.
- The sources of architectural design are listed below.
- Information regarding the application domain for the software to be developed
- Using data-flow diagrams
- 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 behavioral 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 thatcomprises software models along with associated descriptive text. The softwaremodels include static model, interface model, relationship model, and dynamic processmodel. 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.

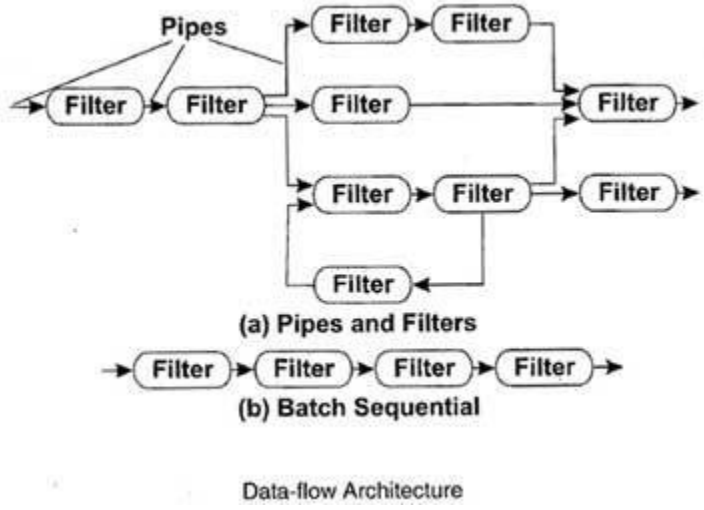
A computer-based system (software is part of this system) exhibits one of the many available architectural styles. Every architectural style describes a system category that includes the following.

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

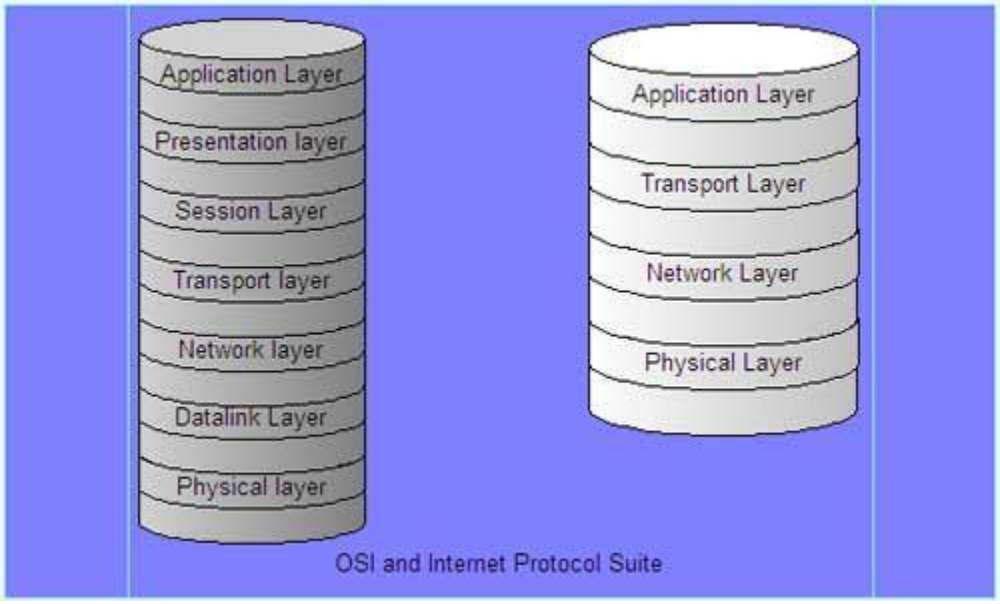
In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data. In this style, components are represented as *objects* and they interact with each other through methods (connectors). This architectural style has two important characteristics,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

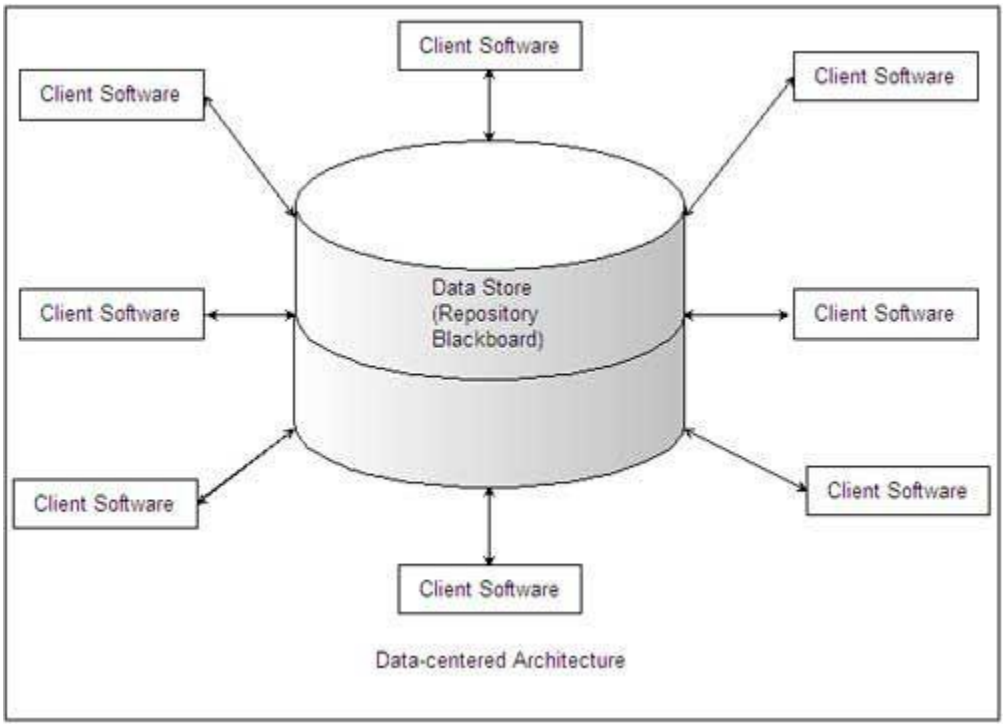
A data-centered architecture has two distinct components: a **central data structure** or data store (central repository) and a **collection of client software.** The datastore (for example, a database or a file) represents the current state of the data andthe client software performs several operations 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.

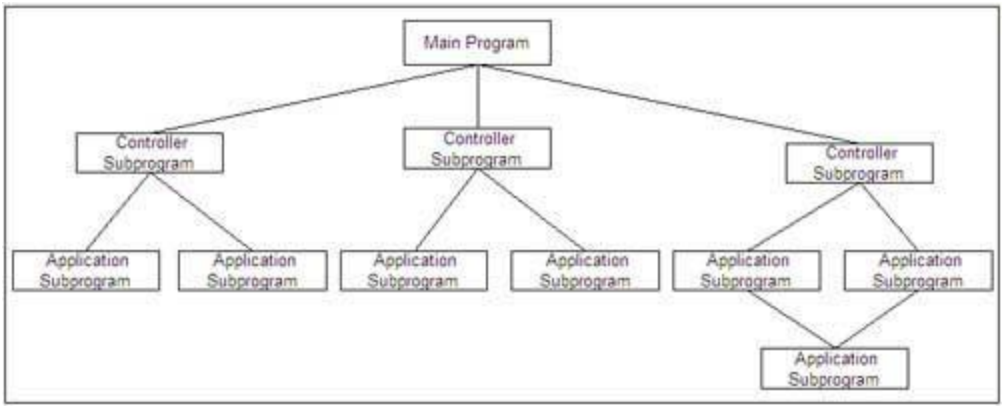
1. Clients operate independently of one another.
2. Data repository is independent of the clients.
3. It adds scalability (that is, new clients can be added easily).
4. It supports modifiability.
5. It achieves data integration in component-based development using blackboard.



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.

1. **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.

4. **Component-level design:** This provides the detailed description of how structural elements of software will actually be implemented.

As soon as the first iteration of architectural design is complete, component-level design takes place. The objective of this design is to transform the design model into functional software. To achieve this objective, the component-level design represents -the internal data structures and processing details of all the software components (defined during architectural design) at an abstraction level, closer to the actual code. In addition, it specifies an interface that may be used to access the functionality of all the software components.

The component-level design can be represented by using different approaches. One approach is to use a programming language while other is to use some intermediate design notation such as graphical (DFD, flowchart, or structure chart), tabular (decision table), or text-based (program design language) whichever is easier to be translated into source code.

The component-level design provides a way to determine whether the defined algorithms, data structures, and interfaces will work properly. Note that a component (also known as **module)** can be defined as a modular building block for the software. However, the meaning of component differs according to how software engineers use it. The modular design of the software should exhibit the following sets of properties.

1. **Provide simple interface:** Simple interfaces decrease the number of interactions. Note that the number of interactions is taken into account while determining whether the software performs the desired function. Simple interfaces also provide support for reusability of components which reduces the cost to a greater extent. It not only decreases the time involved in design, coding, and testing but the overall software development cost is also liquidated gradually with several projects. A number of studies so far have proven that the reusability of software design is the most valuable way of reducing the cost involved in software development.
2. **Ensure information hiding:** The benefits of modularity cannot be achieved merely by decomposing a program into several modules; rather each module should be designed and developed in such a way that the information hiding is ensured. It implies that the implementation details of one module should not be visible to other modules of the program. The concept of information hiding helps in reducing the cost of subsequent design changes.

Modularity has become an accepted approach in every engineering discipline. With the introduction of modular design, complexity of software design has considerably reduced; change in the program is facilitated that has encouraged parallel development of systems. To achieve effective modularity, design concepts like functional independence are considered to be very important.

Functional Independence

Functional independence is the refined form of the design concepts of modularity, abstraction, and information hiding. Functional independence is achieved by developing a module in such a way that it uniquely performs given sets of function without interacting with other parts of the

system. The software that uses the property of functional independence is easier to develop because its functions can be categorized in a systematic manner. Moreover, independent modules require less maintenance and testing activity, as secondary effects caused by design modification are limited with less propagation of errors. In short, it can be said that functional independence is the key to a good software design and a good design results in high-quality software. There exist two qualitative criteria for measuring functional independence, namely, coupling and cohesion.

5. **Interface design:** This depicts how the software communicates with the system that interoperates with it and with the end-users.

Pattern-Based Software Design

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages, some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Working with design patterns during software development can be tricky at times. Their purpose is to provide guidelines for dealing with particular problems that might arise during development. However, the sheer quantity of potential design patterns, combined with the intricacies of each pattern's inner workings, can often make it challenging to select the correct pattern for your particular project, component, or programming language.
we'll focus on the most relevant patterns that are likely to be used in modern development.

Creational Patterns

Creational patterns emphasize the automatic creation of objects within code, rather than requiring you to instantiate objects directly. In most cases, this means that a function or method can call the code necessary to instantiate new objects on your behalf, so you only need to explicitly modify that object creation when it is necessary, and allow default behaviors to take over otherwise.

- **Abstract Factory**: Encapsulates groups of factories based on common themes. Often uses polymorphism, the concept in object-oriented programming that allows one interface to serve as a basis for multiple functions of different types.

- **Builder**: Splits up the construction of an object from its representation. This is usually done by defining a Builder object that presents methods to update the object, without directly interacting with the object itself.
- **Factory**: Creates objects without the need to specify the exact class or type of object to be created. As the name suggests, this object instantiation is performed through a secondary Factory class, again using polymorphism.
- **Prototype**: Creates new objects by prototyping or cloning a prototypical instance of an object. Effectively, an abstract Prototype class is created, and from that base prototype, new secondary inherited classes are defined.
- **Singleton**: Restricts the total number of instances of a particular class to only one at a time. This is commonly used when global access to the object is required across the system, and any changes or queries to the object must be consistent and identical.

Structural patterns

Structural patterns focus on the composition of classes and objects. By using inheritance and interfaces, these patterns allow objects to be composed in a manner that provides new functionality. In most cases, an interface in object-oriented programming is an abstract type or class which has no logical code, but instead is used to define method signatures and behaviors for other classes that will implement the interface.

- **Adapter**: Allows for an interface, which is otherwise incompatible, to be *adapted* to fit a new class. Typically, this is performed by creating a new ClassNameAdapter class that implements the interface, allowing for compatibility across the system.
- **Bridge**: Distinguishes between implementation and abstraction. Or, put another way, it's a pattern that separates the "look and feel" of code from the "logical behavior" of it, which we often see in websites and other visual applications.
- **Composite**: Groups of objects should behave the same as individual objects from within that group. Primarily useful when creating a collection of objects that inherit from the same type, yet are uniquely different types themselves. Since they are of the same composition type, their behavior should be identical when combined into a collective group.
- **Decorator**: Dynamically modifies the behavior of an object at run time, typically by wrapping the object in a decorator class. This pattern is commonly used when an object is instantiated, but as code execution progresses, modifications must be made to the object before it is finalized.
- **Facade**: Creates a front-end (facade) object that obfuscates and simplifies interactions between it and the more complicated interface behind it. Commonly used when a complex series of actions must take place to perform a task, where executing each and every task, in order, is too complicated. Instead, a simple facade replaces that series of tasks with a single task to be executed.
- **Flyweight**: Reduces memory and resource usage by sharing data with other, similar objects. Often relies heavily on Factory-style patterns to access and store already generated data during future executions.
- **Proxy**: Defines a wrapper class for an object, which acts as an interface for the wrapped object. Typically, the proxy class attaches additional behavior onto the wrapped object, without the need to modify the base object class behavior.

Behavioral Patterns

Behavioral patterns are concerned with communication and assignment *between* objects.

- **Chain of Responsibility**: Forces execution to follow a specific chain of command during execution, such that the first object is used, then the second, and so on. Often used as a failsafe in applications, checking the validity of the primary object, before moving onto the secondary object if the primary fails, and so forth.
- **Command**: Decouples the actions of the client from the behavior of the receiver. Often through the use of an interface, an object can specify individual behavior when a particular command is invoked, while a different object type can use that same command, but invoke its own unique behavior instead.
- **Interpreter**: Defines a series of classes used to interpret language syntax from a provided sentence. Typically, each symbol is defined by one class, and then a syntax tree is used to parse (interpret) the overall sentence.
- **Iterator**: Allows access to underlying elements of an object, without exposing those elements or their respective logic. A very commonly used pattern, often as a simple means of fetching the next item in a list or array of objects.
- **Mediator**: Generates a third party object (mediator) that acts as a go-between for interactions between two other similar objects (colleagues). Commonly, this is used when multiple objects need to communicate, but do not (or should not) be aware of the others respective implementation or behavior.
- **Memento**: Stores the state of an object, allowing for restoration (rollback) of the object to a previous state. This behavior is well-known when using word processors that implement the undo feature.
- **Observer**: Creates an event-based dependency between objects, such that an update to the observed object causes the observer objects to be notified. Typically, this is found in many languages that utilize asynchronous functionality, which requires events to be observed and responded to outside of typical execution order.
- **State**: Allows for the behavior of a class to change based on the current state. While these states are often changed throughout execution, the implementation of each possible state is typically defined by a unique class interface.
- **Strategy**: Defines a pattern where logical strategy changes based on the current situation. This is merely an object-oriented extension of common if-else statements, by altering the execution of code based on the outcome of previous code.
- **Template**: Allows for a skeletal template to be used as the basis for execution, without defining the inner-workings of any individual class or object. This is commonly seen in web applications, where the visual interface of the application is generated using templates, which are created using underlying data, but neither the template nor the underlying data are aware of the implementation of the other.
- **Visitor**: Allows for new operations to be added to objects without modifying their original implementation structures. Typically, the visitor class defines unique methods that are shared between it and other objects, without the need for the other object to be aware of the additional functionality.

While this is just a brief description of each design pattern, we hope this serves as a good basis for understanding just how varied design patterns can be, and how useful as well. In future articles, we'll dive deeper into specific design patterns, and examine how they might be implemented in real-world scenarios, using actual code examples.

Mapping data flow into software architecture

In architectural design, the mapping method uses data flow characteristics to derive a commonly used architectural style. A data flow diagram is mapped into program structure using one of the two mapping approaches:
- Transform Mapping
- Transaction Mapping

Structured design is often characterized as a data flow oriented design method because it provides a convenient transition from a data flow diagram. This type of information flow is the driver for the mapping approach.
Transform Flow: The overall flow of data occurs in a sequential manner and follows one, or only a few straight line paths. When a segment of a data flow diagram exhibits these characteristics, transform flow is present.

Transaction Flow: Information flow which is characterized by single data item called a transaction that triggers other data flow along one of many paths. When a data flow diagram takes this kind of form then transaction flow is present.

TRANSFORM MAPPING

It is a set of design steps that allows a data flow diagram with transform flow characteristics to be mapped into a specific architectural style.
- Review the fundamental system model.
- Review and refine data flow diagrams for the software.
- Determine whether the data flow diagram has transform or transaction flow characteristics.
- Isolate the transform center by specifying incoming and outgoing flow boundaries.
- Perform first level factoring.
- Perform second level factoring.
- Refine the first iteration architecture using design heuristics for improved software quality.

Transform Mapping

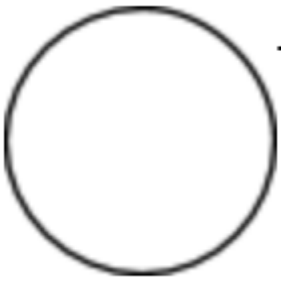
Transform mapping is a technique in which Data Flow Diagrams (DFD's) are mapped to a specific scenario. It is a data flow-oriented mapping technique that uses DFDs to map real life scenarios to a software architecture. These real life scenarios are converted to what we call DFDs which can be applied to a software architecture. This process of converting a real-life situation (termed as system in software engineering) with flow of data to a DFD is called transform

mapping. In this lesson, transform mapping has been described using the scenario of an airline reservation system.

Data Flow Diagram

A Data Flow Diagram(DFD) shows the flow of data through the system. It is also used for modeling the requirements. DFD is often called as a data flow graph. The data flow diagram is created with the help of various symbols which represent a process, data repository etc.

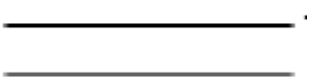
Symbols used in DFD



This symbol denotes a process which transforms data input into data output

This symbol depicts the data flow, it shows flow of data into or out of a process or data store.

This symbol depicts a source or sink. It is an external entity that acts as a source of system input or system output

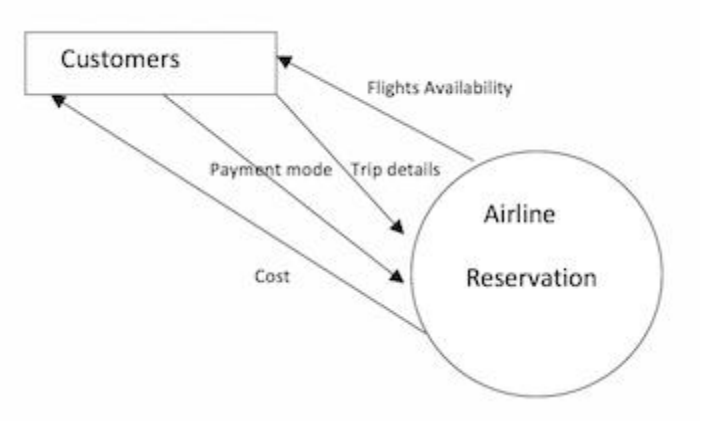


This symbol depicts a data repository, a place where collection of data items are stored.

Alternatively used for data repository

Example of Transform Mapping

See the figure below. This is a depiction of DFD level 0 where an architecture of an airline reservation system has been shown. The customer supplies trip details and payment mode as an input to the Airline reservation system. Airline reservation is a process which provides flight availability as well cost as an output to the customers which is a source in this case. This is the level 0 hierarchy of the DFD's.



DFD level-0

The next figure shows DFD level-1. You can see a repository named Flight database where the details related to seat availability are stored. This information is displayed on the website of the reservation system which informs the customer how many seats are available on the flight. Customer retrieves the details from the website after providing the specification of the trip details during the 'Retrieve and Display' process. 'Seat Reservation' is a process which is indirectly connected to another process called 'Total Cost Calculation'. This process helps in calculating the cost for the number of seats requested. The cost details are then provided to the customer based on the number of seats reserved which is indirectly provided by the process 'Seat Reservation'. 'Ticket Generation' is a process which generates tickets once the customer provides the payment details. The input for this process is payment and the output of this process is the E-ticket. 'Cancellation' is another such process which cancels the ticket of a customer when the customer provides the E-ticket, the output from the process being an acknowledgment sent back to the customer.

TRANSACTION MAPPING

In transaction mapping, information flow along two of the three action paths accommodates additional incoming flow. Each action path flows into a single transform, display messages and status. The design steps for transaction mapping is similar with a major difference lies in mapping of data flow diagram to software structure.

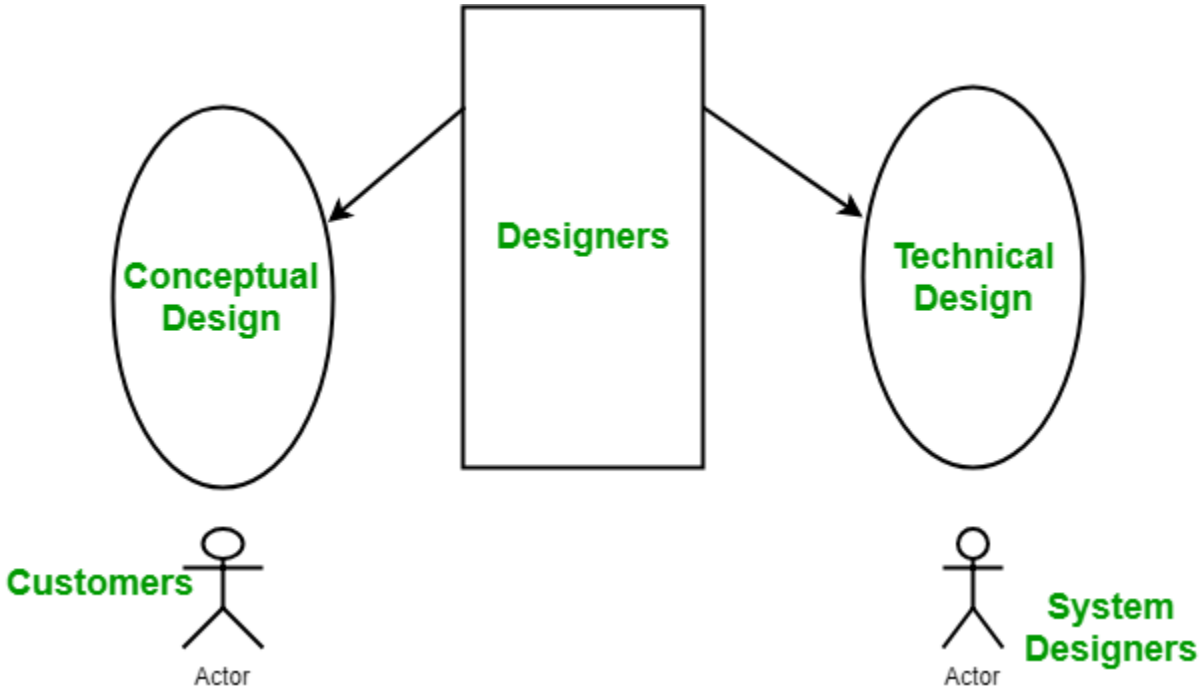
- Review the fundamental system model.
- Review and refine data flow diagrams for the software.
- Determine whether the data flow diagram has transform or transaction flow characteristics.
- Identify the transaction center and the flow characteristics along each of the action paths.
- Map the data flow diagram in a program structure amenable to transaction processing.
- Factor and refine the transaction structure and the structure of each action path.
- Refine the first iteration architecture using design heuristics for improved software quality.

Once an architecture has been derived, it is elaborated and then analyzed against quality criteria.

Cohesion, Coupling

The purpose of Design phase in the Software Development Life Cycle is to produce a solution to a problem given in the SRS(Software Requirement Specification) document. The output of the design phase is Sofware Design Document (SDD).

Basically, design is a two-part iterative process. First part is Conceptual Design that tells the customer what the system will do. Second is Technical Design that allows the system builders to understand the actual hardware and software needed to solve customer’s problem.



Conceptual design of system:

- Written in simple language i.e. customer understandable language.
- Detail explaination about system characteristics.
- Describes the functionality of the system.
- It is independent of implementation.
- Linked with requirement document.

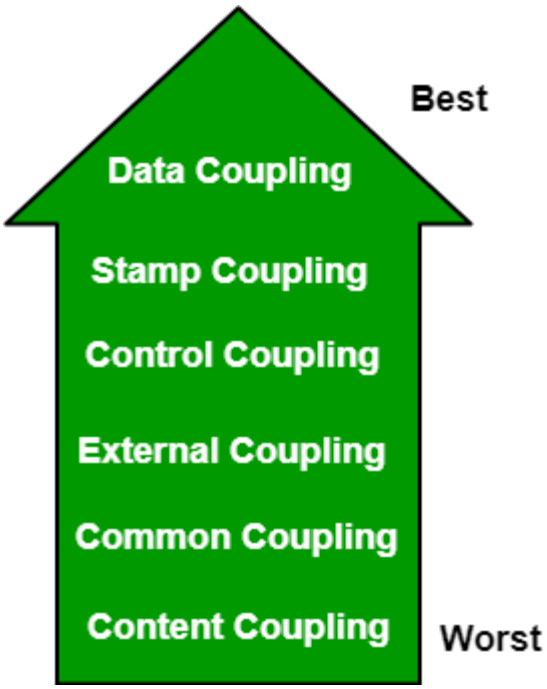
Technical Design of system:

- Hardware component and design.
- Functionality and hierarchy of software component.
- Software architecture
- Network architecture
- Data structure and flow of data.
- I/O component of the system.
- Shows interface.

Modularization: Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

- Easy to understand the system.
- System maintenance is easy.
- A module can be used many times as their requirements. No need to write it again and again.

Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.



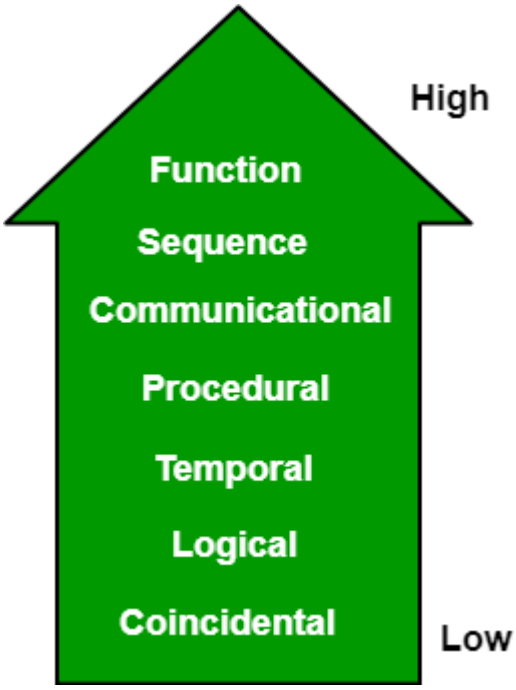
Types of Coupling:

- Data Coupling: If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain tramp data. Example-customer billing system.
- Stamp Coupling In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice made by the insightful designer, not a lazy programmer.
- Control Coupling: If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
- External Coupling: In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
- Common Coupling: The modules have shared data such as global data structures.The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.
- Content Coupling: In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Cohesion may be represented as a "spectrum." We always

strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is nonlinear.

Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



Types of Cohesion:

- Functional Cohesion: Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
- Sequential Cohesion: An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
- Communicational Cohesion: Two elements operate on the same input data or contribute towards the same output data. Example- update record int the database and send it to the printer.
- Procedural Cohesion: Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
- Temporal Cohesion: The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time-span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at init time.
- Logical Cohesion: The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.

- Coincidental Cohesion: The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

User interface analysis and Design

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction.

UI can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interfacing screens

UI is broadly divided into two categories:

- Command Line Interface
- Graphical User Interface

Command Line Interface (CLI)

CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate.

CLI uses less amount of computer resource as compared to GUI.

CLI Elements

A text-based command line interface can have the following elements:

- Command Prompt - It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.
- Cursor - It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.
- Command - A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

Graphical User Interface

Graphical User Interface provides the user graphical means to interact with the system. GUI can be combination of both hardware and software. Using GUI, user interprets the software.

Typically, GUI is more resource consuming than that of CLI. With advancing technology, the programmers and designers create complex GUI designs that work with more efficiency, accuracy and speed.

GUI Elements

GUI provides a set of components to interact with software or hardware.

Every graphical component provides a way to work with the system. A GUI system has following elements such as:

- Window - An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure. It is easier for a user to navigate in the file system in an exploring window. Windows can be minimized, resized or maximized to the size of screen. They can be moved anywhere on the screen. A window may contain another window of the same application, called child window.
- Tabs - If an application allows executing multiple instances of itself, they appear on the screen as separate windows. Tabbed Document Interface has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers use this feature.

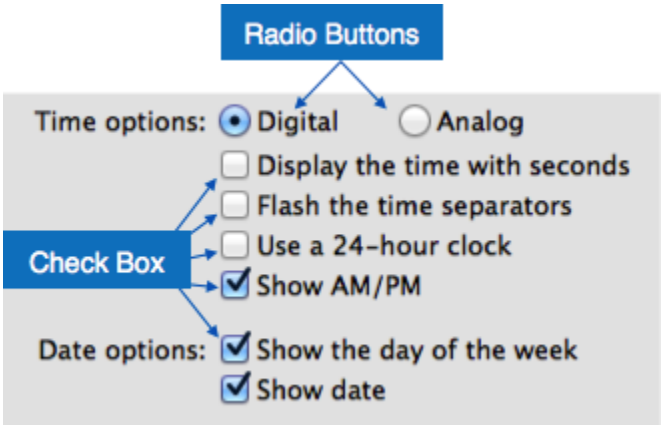
- Menu - Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window. The menu can be programmed to appear or hide on mouse clicks.
- Icon - An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened. Icon displays application and programs installed on a system in the form of small pictures.
- Cursor - Interacting devices such as mouse, touch pad, digital pen are represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.

Application specific GUI components

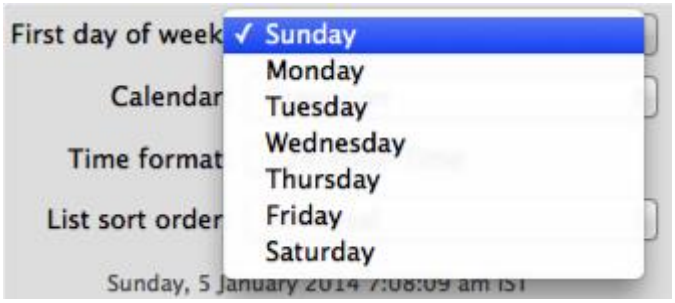
A GUI of an application contains one or more of the listed GUI elements:

- Application Window - Most application windows uses the constructs supplied by operating systems but many use their own customer created windows to contain the contents of application.
- Dialogue Box - It is a child window that contains message for the user and request for some action to be taken. For Example: Application generate a dialogue to get confirmation from user to delete a file.

- Text-Box - Provides an area for user to type and enter text-based data.
- Buttons - They imitate real life buttons and are used to submit inputs to the software.



- Radio-button - Displays available options for selection. Only one can be selected among all offered.
- Check-box - Functions similar to list-box. When an option is selected, the box is marked as checked. Multiple options represented by check boxes can be selected.
- List-box - Provides list of available items for selection. More than one item can be selected.



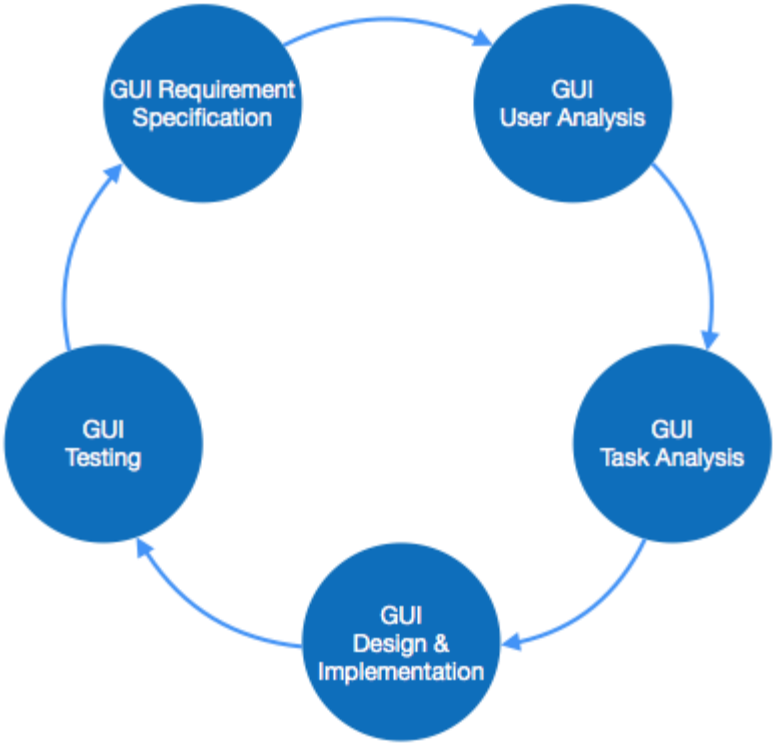
Other impressive GUI components are:

- Sliders
- Combo-box
- Data-grid
- Drop-down list

User Interface Design Activities

There are a number of activities performed for designing user interface. The process of GUI design and implementation is alike SDLC. Any model can be used for GUI implementation among Waterfall, Iterative or Spiral Model.

A model used for GUI design and development should fulfill these GUI specific steps.



- GUI Requirement Gathering - The designers may like to have list of all functional and non-functional requirements of GUI. This can be taken from user and their existing software solution.
- User Analysis - The designer studies who is going to use the software GUI. The target audience matters as the design details change according to the knowledge and competency level of the user. If user is technical savvy, advanced and complex GUI can be incorporated. For a novice user, more information is included on how-to of software.
- Task Analysis - Designers have to analyze what task is to be done by the software solution. Here in GUI, it does not matter how it will be done. Tasks can be represented in hierarchical manner taking one major task and dividing it further into smaller sub-tasks. Tasks provide goals for GUI presentation. Flow of information among sub-tasks determines the flow of GUI contents in the software.
- GUI Design & implementation - Designers after having information about requirements, tasks and user environment, design the GUI and implements into code and embed the GUI with working or dummy software in the background. It is then self-tested by the developers.
- Testing - GUI testing can be done in various ways. Organization can have in-house inspection, direct involvement of users and release of beta version are few of them. Testing may include usability, compatibility, user acceptance etc.

GUI Implementation Tools

There are several tools available using which the designers can create entire GUI on a mouse click. Some tools can be embedded into the software environment (IDE).

GUI implementation tools provide powerful array of GUI controls. For software customization, designers can change the code accordingly.

There are different segments of GUI tools according to their different use and platform.

Example

Mobile GUI, Computer GUI, Touch-Screen GUI etc. Here is a list of few tools which come handy to build GUI:

- FLUID
- AppInventor (Android)
- LucidChart
- Wavemaker
- Visual Studio

User Interface Golden rules

The following rules are mentioned to be the golden rules for GUI design, described by Shneiderman and Plaisant in their book (Designing the User Interface).

- Strive for consistency - Consistent sequences of actions should be required in similar situations. Identical terminology should be used in prompts, menus, and help screens. Consistent commands should be employed throughout.
- Enable frequent users to use short-cuts - The user's desire to reduce the number of interactions increases with the frequency of use. Abbreviations, function keys, hidden commands, and macro facilities are very helpful to an expert user.
- Offer informative feedback - For every operator action, there should be some system feedback. For frequent and minor actions, the response must be modest, while for infrequent and major actions, the response must be more substantial.
- Design dialog to yield closure - Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and this indicates that the way ahead is clear to prepare for the next group of actions.
- Offer simple error handling - As much as possible, design the system so the user will not make a serious error. If an error is made, the system should be able to detect it and offer simple, comprehensible mechanisms for handling the error.
- Permit easy reversal of actions - This feature relieves anxiety, since the user knows that errors can be undone. Easy reversal of actions encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.

- Support internal locus of control - Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.
- Reduce short-term memory load - The limitation of human information processing in short-term memory requires the displays to be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

Metrics for Design Models

The success of a software project depends largely on the quality and effectiveness of the software design. Hence, it is important to develop software metrics from which meaningful indicators can be derived. With the help of these indicators, necessary steps are taken to design the software according to the user requirements. Various design metrics such as architectural design metrics, component-level design metrics, user-interface design metrics, and metrics for object-oriented design are used to indicate the complexity, quality, and so on of the software design.

Architectural Design Metrics

These metrics focus on the features of the program architecture with stress on architectural structure and effectiveness of components (or modules) within the architecture. In architectural design metrics, three software design complexity measures are defined, namely, structural complexity, data complexity, and system complexity.

In hierarchical architectures (call and return architecture), say module ‘j’, structural complexity is calculated by the following equation.

$S(j) = f^2_{out}(j)$

Where

$f_{out}(j)$ = fan-out of module ‘j’ [Here, fan-out means number of modules that are subordinating module j].

Complexity in the internal interface for a module ‘j’ is indicated with the help of data complexity, which is calculated by the following equation.

$D(j) = V(j) / [f_{out}(j)+l]$

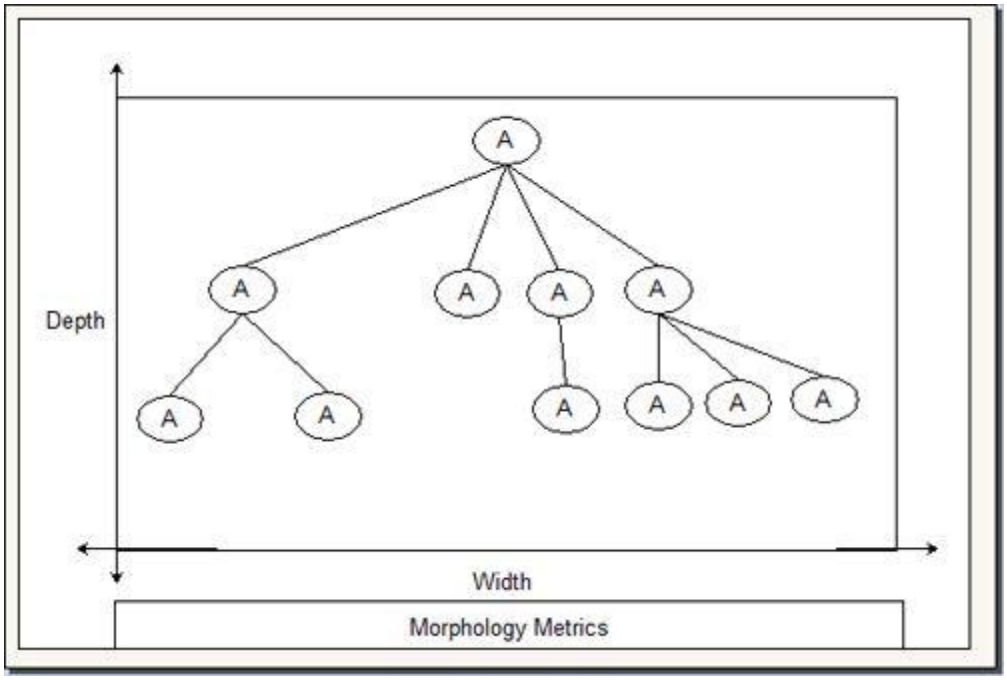
Where

V(j) = number of input and output variables passed to and from module ‘j’.

System complexity is the sum of structural complexity and data complexity and is calculated by the following equation.

$C(j) = S(j) + D(j)$

The complexity of a system increases with increase in structural complexity, data complexity, and system complexity, which in turn increases the integration and testing effort in the later stages.



In addition, various other metrics like simple morphology metrics are also used. These metrics allow comparison of different program architecture using a set of straightforward dimensions. A metric can be developed by referring to call and return architecture. This metric can be defined by the following equation.

Size = n+a

Where

n = number of nodes

a= number of arcs.

For example, there are 11 nodes and 10 arcs. Here, Size can be calculated by the following equation.

Size = n+a = 11+10+21.

Depth is defined as the longest path from the top node (root) to the leaf node and width is defined as the maximum number of nodes at any one level.

Coupling of the architecture is indicated by arc-to-node ratio. This ratio also measures the connectivity density of the architecture and is calculated by the following equation.

r=a/n

Quality of software design also plays an important role in determining the overall quality of the software. Many software quality indicators that are based on measurable design characteristics of a computer program have been proposed. One of them is Design Structural Quality Index (DSQI), which is derived from the information obtained from data and architectural design. To calculate DSQI, a number of steps are followed, which are listed below.

1. To calculate DSQI, the following values must be determined.

- Number of components in program architecture (S_1)
- Number of components whose correct function is determined by the Source of input data (S_2)
- Number of components whose correct function· depends on previous processing (S_3)
- Number of database items (S_4)
- Number of different database items (S_5)
- Number of database segments (S_6)
- Number of components having single entry and exit (S_7).

2. Once all the values from S_1 to S_7 are known, some intermediate values are calculated, which are listed below.

Program structure (D_1): If discrete methods are used for developing architectural design then D_1= 1, else $D_1 = 0$

Module independence (D_2): $D_2 = 1-(S_2/S_1)$

Modules not dependent on prior processing (D_3): $D_3 = 1-(S_3/S_1)$

Database size (D_4): $D_4 = 1-(S_5/S_4)$

Database compartmentalization (D_5):$D_5 = 1-(S_6/S_4)$

Module entrance/exit characteristic (D_6): $D_6 = 1-(S_7/S_1)$

3. Once all the intermediate values are calculated, OSQI is calculated by the following equation.

$DSQI = \sum W_i D_i$

Where

i = 1 to 6

$\sum W_i = 1$ (W_i is the weighting of the importance of intermediate values).

In conventional software, the focus of component – level design metrics is on the internal characteristics of the software components; The software engineer can judge the quality of the component-level design by measuring module cohesion, coupling and complexity; Component-level design metrics are applied after procedural design is final. Various metrics developed for component-level design are listed below.

- Cohesion metrics: Cohesiveness of a module can be indicated by the definitions of the following five concepts and measures.
- Data slice: Defined as a backward walk through a module, which looks for values of data that affect the state of the module as the walk starts
- Data tokens: Defined as a set of variables defined for a module
- Glue tokens: Defined as a set of data tokens, which lies on one or more data slice
- Superglue tokens: Defined as tokens, which are present in every data slice in the module
- Stickiness: Defined as the stickiness of the glue token, which depends on the number of data slices that it binds.
- Coupling Metrics: This metric indicates the degree to which a module is connected to other modules, global data and the outside environment. A metric for module coupling has been proposed, which includes data and control flow coupling, global coupling, and environmental coupling.

- Measures defined for data and control flow coupling are listed below.

d_i = total number of input data parameters

c_i = total number of input control parameters

d_o= total number of output data parameters

c_o= total number of output control parameters

\$1§ Measures defined for global coupling are listed below.

g_d= number of global variables utilized as data

g_c = number of global variables utilized as control

\$1§ Measures defined for environmental coupling are listed below.

w = number of modules called

r = number of modules calling the modules under consideration

By using the above mentioned measures, module-coupling indicator (m_c) is calculated by using the following equation.

$m_c = K/M$

Where

K = proportionality constant

$M = d_i + (a*c_i) + d_o + (b*c_o) + g_d + (c*g_c) + w + r$.

Note that K, a, b, and c are empirically derived. The values of m_c and overall module coupling are inversely proportional to each other. In other words, as the value of m_c increases, the overall module coupling decreases.

Complexity Metrics: Different types of software metrics can be calculated to ascertain the complexity of program control flow. One of the most widely used complexity metrics for ascertaining the complexity of the program is cyclomatic complexity.

Many metrics have been proposed for user interface design. However, layout appropriateness metric and cohesion metric for user interface design are the commonly used metrics. Layout Appropriateness (LA) metric is an important metric for user interface design. A typical Graphical User Interface (GUI) uses many layout entities such as icons, text, menus, windows, and so on. These layout entities help the users in completing their tasks easily. In to complete a given task with the help of GUI, the user moves from one layout entity to another.

Appropriateness of the interface can be shown by absolute and relative positions of each layout entities, frequency with which layout entity is used, and the cost of changeover from one layout entity to another.

Cohesion metric for user interface measures the connection among the onscreen contents. Cohesion for user interface becomes high when content presented on the screen is from a single major data object (defined in the analysis model). On the other hand, if content presented on the screen is from different data objects, then cohesion for user interface is low.

In addition to these metrics, the direct measure of user interface interaction focuses on activities like measurement of time required in completing specific activity, time required in recovering from an error condition, counts of specific operation, text density, and text size. Once all these measures are collected, they are organized to form meaningful user interface metrics, which can help in improving the quality of the user interface.

Metrics for Object-oriented Design

In order to develop metrics for object-oriented (OO) design, nine distinct and measurable characteristics of OO design are considered, which are listed below.

- Complexity: Determined by assessing how classes are related to each other
- Coupling: Defined as the physical connection between OO design elements
- Sufficiency: Defined as the degree to which an abstraction possesses the features required of it
- Cohesion: Determined by analyzing the degree to which a set of properties that the class possesses is part of the problem domain or design domain
- Primitiveness: Indicates the degree to which the operation is atomic
- Similarity: Indicates similarity between two or more classes in terms of their structure, function, behavior, or purpose
- Volatility: Defined as the probability of occurrence of change in the OO design
- Size: Defined with the help of four different views, namely, population, volume, length, and functionality. Population is measured by calculating the total number of OO entities, which can be in the form of classes or operations. Volume measures are collected dynamically at any given point of time. Length is a measure of interconnected designs such as depth of inheritance tree. Functionality indicates the value rendered to the user by the OO application.

Metrics for Coding

Halstead proposed the first analytic laws for computer science by using a set of primitive measures, which can be derived once the design phase is complete and code is generated. These measures are listed below.

n_1 = number of distinct operators in a program

n_2 = number of distinct operands in a program

N_1 = total number of operators

N_2= total number of operands.

By using these measures, Halstead developed an expression for overall program length, program volume, program difficulty, development effort, and so on.

Program length (N) can be calculated by using the following equation.

$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$.

Program volume (V) can be calculated by using the following equation.

$V = N \log_2 (n_1 + n_2)$.

Note that program volume depends on the programming language used and represents the volume of information (in bits) required to specify a program. Volume ratio (L)can be calculated by using the following equation.

$$L = \frac{\text{Volume of the most compact form of a program}}{\text{Volume of the actual program}}$$

Where, value of L must be less than 1. Volume ratio can also be calculated by using the following equation.

$L = (2/n_1)* (n_2/N_2)$.

Program difficulty level (D) and effort (E)can be calculated by using the following equations.

$D = (n_1/2)*(N_2/n_2)$.

$E = D * V$.

Metrics for Software Testing

Majority of the metrics used for testing focus on testing process rather than the technical characteristics of test. Generally, testers use metrics for analysis, design, and coding to guide them in design and execution of test cases.

Function point can be effectively used to estimate testing effort. Various characteristics like errors discovered, number of test cases needed, testing effort, and so on can be determined by estimating the number of function points in the current project and comparing them with any previous project.

Metrics used for architectural design can be used to indicate how integration testing can be carried out. In addition, cyclomatic complexity can be used effectively as a metric in the basis-path testing to determine the number of test cases needed.

Halstead measures can be used to derive metrics for testing effort. By using program volume (V) and program level (PL),Halstead effort (e)can be calculated by the following equations.

$e = V/ PL$

Where

$PL = 1/ [(n_1/2) * (N_2/n_2)]$ … (1)

For a particular module (z), the percentage of overall testing effort allocated can be calculated by the following equation.

Percentage of testing effort (z) = $e(z)/\sum e(i)$

Where, e(z) is calculated for module z with the help of equation (1). Summation in the denominator is the sum of Halstead effort (e) in all the modules of the system.

For developing metrics for object-oriented (OO) testing, different types of design metrics that have a direct impact on the testability of object-oriented system are considered. While developing metrics for OO testing, inheritance and encapsulation are also considered. A set of metrics proposed for OO testing is listed below.

- Lack of cohesion in methods (LCOM): This indicates the number of states to be tested. LCOM indicates the number of methods that access one or more same attributes. The value of LCOM is 0, if no methods access the same attributes. As the value of LCOM increases, more states need to be tested.
- Percent public and protected (PAP): This shows the number of class attributes, which are public or protected. Probability of adverse effects among classes increases with increase in value of PAP as public and protected attributes lead to potentially higher coupling.
- Public access to data members (PAD): This shows the number of classes that can access attributes of another class. Adverse effects among classes increase as the value of PAD increases.
- Number of root classes (NOR): This specifies the number of different class hierarchies, which are described in the design model. Testing effort increases with increase in NOR.
- Fan-in (FIN): This indicates multiple inheritances. If value of FIN is greater than 1, it indicates that the class inherits its attributes and operations from many root classes. Note that this situation (where FIN> 1) should be avoided.

Metrics for Software Maintenance

For the maintenance activities, metrics have been designed explicitly. IEEE have proposed Software Maturity Index (SMI), which provides indications relating to the stability of software product. For calculating SMI, following parameters are considered.

- Number of modules in current release (M_T)
- Number of modules that have been changed in the current release (F_e)
- Number of modules that have been added in the current release (F_a)
- Number of modules that have been deleted from the current release (F_d)

Once all the parameters are known, SMI can be calculated by using the following equation.

$SMI = [M_T - (F_a + F_e + F_d)]/M_T$.

Note that a product begins to stabilize as 8MI reaches 1.0. SMI can also be used as a metric for planning software maintenance activities by developing empirical models in order to know the effort required for maintenance.