**Software design**

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes “good” software, and an overall commitment to quality are critical success factors for a competent design. The design model is the equivalent of an architect’s plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software.

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

## ****Software Design Concepts:****

Concepts are defined as a principal idea or invention that comes into our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system or software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:

**Points should be considered while Designing Software:**

1. **Abstraction-**(**Hide Irrelevant data** )  
   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.
2. **Modularity- (subdivide the system)**   
   Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays, there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components, then it would mean the system is complex which requires a lot of effort (cost) but if we are able to divide the system into components then the cost would be small.
3. **Architecture- (design a structure of something )**  
   Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
4. **Refinement- (removes impurities)**  
   Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
5. **Pattern- (a Repeated form)**   
   The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
6. **Information Hiding – Hide the Information**   
   Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can’t be accessed by any other modules.
7. **Refactoring-( Reconstruct something )**  
   Refactoring simply means reconstructing something in such a way that it does not affect the behaviour of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without impacting the behaviour or its functions. Fowler has defined refactoring as “the process of changing a software system in a way that it won’t impact the behavior of the design and improves the internal structure”.

# **Effective Modular Design**

**The role of effective modular design in software engineering:** Any software comprises of many systems which contains several sub-systems and those sub-systems further contains their sub-systems. So, designing a complete system in one go comprising of each and every required functionality is a hectic work and the process can have many errors because of its vast size. Thus in order to solve this problem the developing team breakdown the complete software into various modules. A module is defined as the unique and addressable components of the software which can be solved and modified independently without disturbing ( or affecting in very small amount ) other modules of the software. Thus every software design should follow modularity. The process of breaking down a software into multiple independent modules where each module is developed separately is called **Modularization**. Effective modular design can be achieved if the partitioned modules are separately solvable, modifiable as well as compilable. Here separate compilable modules means that after making changes in a module there is no need of recompiling the whole software system. In order to build a software with effective modular design there is a factor **“Functional Independence”** which comes into play. The meaning of Functional Independence is that a function is atomic in nature so that it performs only a single task of the software without or with least interaction with other modules. Functional Independence is considered as a sign of growth in modularity i.e., presence of larger functional independence results in a software system of good design and design further affects the quality of the software. **Benefits of Independent modules/functions in a software design:** Since the functionality of the software have been broken down into atomic levels, thus developers get a clear requirement of each and every functions and hence designing of the software becomes easy and error free. As the modules are independent they have limited or almost no dependency on other modules. So, making changes in a module without affecting the whole system is possible in this approach. Error propagation from one module to another and further in whole system can be neglected and it saves time during testing and debugging. Independence of modules of a software system can be measured using 2 criteria : [Cohesion, and Coupling](https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/). These are explained as following below.

Cohesion

Cohesion is a natural extension of the information hiding. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. 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. That is, low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module. Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed. At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed coincidentally cohesive. A module that performs tasks that are related logically (e.g., a module that produces all output regardless of type) is logically cohesive. When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits temporal cohesion. As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed prespecified bounds.

It performs the following tasks:

(1) computes supplementary data based on original computed data,

(2) produces an error report (with graphical content) on the user's workstation,

(3) performs follow-up calculations requested by the user,

(4) updates a database, and

(5) enables menu selection for subsequent processing.

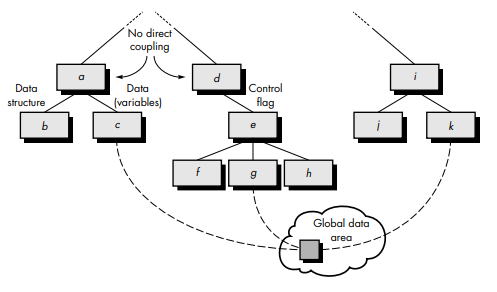
Although the preceding tasks are loosely related, each is an independent functional entity that might best be performed as a separate module. Combining the functions into a single module can serve only to increase the likelihood of error propagation when a modification is made to one of its processing tasks. Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, procedural cohesion exists. When all processing elements concentrate on one area of a data structure, communicational cohesion is present. High cohesion is characterized by a module that performs one distinct procedural task. As we have already noted, it is unnecessary to determine the precise level of cohesion. Rather it is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.(A1—A2—A3—A4)
5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.(PERFORM ACTIVITY RELATED IN SAME TIME)(BOOT SETUP)
6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

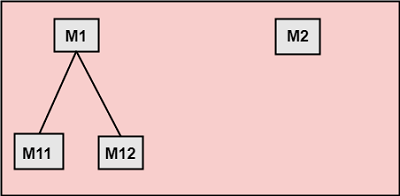
**Coupling**

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" [STE74], caused when errors occur at one location and propagate through a system. Modules a and d are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module c is subordinate to module a and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called data coupling) is exhibited in this portion of structure. A variation of data coupling, called stamp coupling, is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules b and a.

Relatively high levels of coupling occur when modules are tied to an environment external to software. For example, I/O couples a module to specific devices, formats, and communication protocols. External coupling is essential, but should be limited to a small number of modules with a structure. High coupling also occurs when a number of modules reference a global data area. Common coupling, as this mode is called, is shown in Figure 13.6. Modules c, g, and k each access a data item in a global data area (e.g., a disk file or a globally accessible memory area). Module c initializes the item. Later module g re-compute and updates the item. Let's assume that an error occurs and g updates the item incorrectly. Much later in processing module, k reads the item, attempts to process it, and fails, causing the software to abort. The apparent cause of abort is module k; the actual cause, module g. Diagnosing problems in structures with considerable common coupling is time consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them. The highest degree of coupling, content coupling, occurs when one module makes use of data or control information maintained within the boundary of another module. Secondarily, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

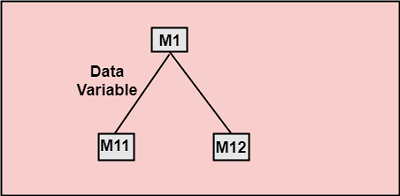


1. **No Direct Coupling:** There is no direct coupling between M1 and M2.



In this case, modules are subordinates to different modules. Therefore, no direct coupling.

**2. Data Coupling:** When data of one module is passed to another module, this is called data coupling.



**3. Stamp Coupling:** Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

**4. Control Coupling:** Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

**5. External Coupling:** External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

**6. Common Coupling:** Two modules are common coupled if they share information through some global data items.



1. **Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.(MODULE MODIFIED ANOTHER MODULE)

# **User Interface Design**

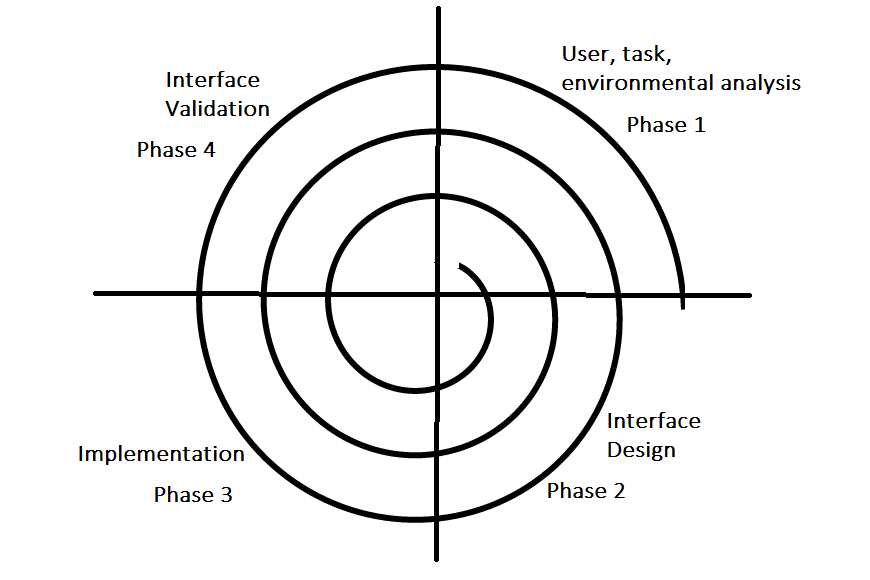
The user interface is the front-end application view to which the user interacts to use the software. The software becomes more popular if its user interface is:

1. **Attractive**
2. **Simple to use**
3. **Responsive in a short time**
4. **Clear to understand**
5. **Consistent on all interface screens**

## Types of User Interface

1. **Command Line Interface:** The Command Line Interface provides a command prompt, where the user types the command and feeds it to the system. The user needs to remember the syntax of the command and its use.
2. **Graphical User Interface:** Graphical User Interface provides a simple interactive interface to interact with the system. GUI can be a combination of both hardware and software. Using GUI, the user interprets the software.

## User Interface Design Process



*User Interface Design Process*

The analysis and design process of a user interface is iterative and can be represented by a spiral model. The analysis and design process of user interface consists of four framework activities.

### 1. **User, Task, Environmental Analysis, and Modeling**

Initially, the focus is based on the profile of users who will interact with the system, i.e., understanding, skill and knowledge, type of user, etc., based on the user’s profile users are made into categories. From each category requirements are gathered. Based on the requirement’s developer understand how to develop the interface. Once all the requirements are gathered a detailed analysis is conducted. In the analysis part, the tasks that the user performs to establish the goals of the system are identified, described and elaborated. The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are:

1. Where will the interface be located physically?
2. Will the user be sitting, standing, or performing other tasks unrelated to the interface?
3. Does the interface hardware accommodate space, light, or noise constraints?
4. Are there special human factors considerations driven by environmental factors?

### 2. **Interface Design**

The goal of this phase is to define the set of interface objects and actions i.e., control mechanisms that enable the user to perform desired tasks. Indicate how these control mechanisms affect the system. Specify the action sequence of tasks and subtasks, also called a user scenario. Indicate the state of the system when the user performs a particular task. Always follow the three golden rules stated by Theo Mandel. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. This phase serves as the foundation for the implementation phase.

### **3. Interface Construction and Implementation**

The implementation activity begins with the creation of a prototype (model) that enables usage scenarios to be evaluated. As iterative design process continues a User Interface toolkit that allows the creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment can be used for completing the construction of an interface.

### **4. Interface Validation**

This phase focuses on testing the interface. The interface should be in such a way that it should be able to perform tasks correctly, and it should be able to handle a variety of tasks. It should achieve all the user’s requirements. It should be easy to use and easy to learn. Users should accept the interface as a useful one in their work.

## User Interface Design Golden Rules

The following are the golden rules stated by Theo Mandel that must be followed during the design of the interface. **Place the user in control:**

1. **Define the interaction modes in such a way that does not force the user into unnecessary or undesired actions:**The user should be able to easily enter and exit the mode with little or no effort.
2. **Provide for flexible interaction:** Different people will use different interaction mechanisms, some might use keyboard commands, some might use mouse, some might use touch screen, etc., Hence all interaction mechanisms should be provided.
3. **Allow user interaction to be interruptible and undoable:** When a user is doing a sequence of actions the user must be able to interrupt the sequence to do some other work without losing the work that had been done. The user should also be able to do undo operation.
4. **Streamline interaction as skill level advances and allow the interaction to be customized:** Advanced or highly skilled user should be provided a chance to customize the interface as user wants which allows different interaction mechanisms so that user doesn’t feel bored while using the same interaction mechanism.
5. **Hide technical internals from casual users:**The user should not be aware of the internal technical details of the system. He should interact with the interface just to do his work.
6. **Design for direct interaction with objects that appear on-screen:**The user should be able to use the objects and manipulate the objects that are present on the screen to perform a necessary task. By this, the user feels easy to control over the screen.

## ****Reduce the User’s Memory Load****

1. **Reduce demand on short-term memory:** When users are involved in some complex tasks the demand on short-term memory is significant. So the interface should be designed in such a way to reduce the remembering of previously done actions, given inputs and results.
2. **Establish meaningful defaults:** Always an initial set of defaults should be provided to the average user, if a user needs to add some new features then he should be able to add the required features.
3. **Define shortcuts that are intuitive:**Mnemonics should be used by the user. Mnemonics means the keyboard shortcuts to do some action on the screen.
4. **The visual layout of the interface should be based on a real-world metaphor:** Anything you represent on a screen if it is a metaphor for a real-world entity then users would easily understand.
5. **Disclose information in a progressive fashion:**The interface should be organized hierarchically i.e., on the main screen the information about the task, an object or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

## ****Make the Interface Consistent****

1. **Allow the user to put the current task into a meaningful context:**Many interfaces have dozens of screens. So it is important to provide indicators consistently so that the user know about the doing work. The user should also know from which page has navigated to the current page and from the current page where it can navigate.
2. **Maintain consistency across a family of applications:**in The development of some set of applications all should follow and implement the same design, rules so that consistency is maintained among applications.
3. If past interactive models have created user expectations do not make changes unless there is a compelling reason.

User interface design is a crucial aspect of software engineering, as it is the means by which users interact with software applications. A well-designed user interface can improve the usability and user experience of an application, making it easier to use and more effective.

## Key Principles for Designing User Interfaces

1. **User-centered design:** User interface design should be focused on the needs and preferences of the user. This involves understanding the user’s goals, tasks, and context of use, and designing interfaces that meet their needs and expectations.
2. **Consistency:** Consistency is important in user interface design, as it helps users to understand and learn how to use an application. Consistent design elements such as icons, color schemes, and navigation menus should be used throughout the application.
3. **Simplicity:**User interfaces should be designed to be simple and easy to use, with clear and concise language and intuitive navigation. Users should be able to accomplish their tasks without being overwhelmed by unnecessary complexity.
4. **Feedback:** Feedback is significant in user interface design, as it helps users to understand the results of their actions and confirms that they are making progress towards their goals. Feedback can take the form of visual cues, messages, or sounds.
5. **Accessibility:** User interfaces should be designed to be accessible to all users, regardless of their abilities. This involves considering factors such as colour contrast, font size, and assistive technologies such as screen readers.
6. **Flexibility:**User interfaces should be designed to be flexible and customizable, allowing users to tailor the interface to their own preferences and needs.