UNIT 3:

Elements of the Analysis Model

The specific elements of the analysis model are dictated by the analysis modeling method that is to be used. However, a set of generic elements is common to most analysis models.

Scenario-based elements: The system is described from the user's point of view using a scenario-based approach.

It always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use-cases that describe how the software engineering models will be used.

Functional—processing narratives for software functions

Use-case- descriptions of the interaction between an "actor" and the system.

Class-based elements: Each usage scenario implies a set of "objects" that are manipulated as an actor interacts with the system.

These objects are categorized into classes- a collection of things that have similar attributes and common behaviors.

One way to isolate classes is to look for descriptive nouns in a use-case script. At least some of the nouns will be candidate classes.

Flow-oriented elements: Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies functions to transform it; and produces output in a variety of forms.

Behavioral elements: The state diagram is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state.

A *state* is any observable mode of behavior. Moreover, the state diagram indicates what actions are taken as a consequence of a particular event.

Scenario Based Modeling

A use case diagram is a graphic depiction of the interactions among the elements of a system.

Use cases are simply an aid to defining what exists outside the system (actors) and what should be performed by the system.

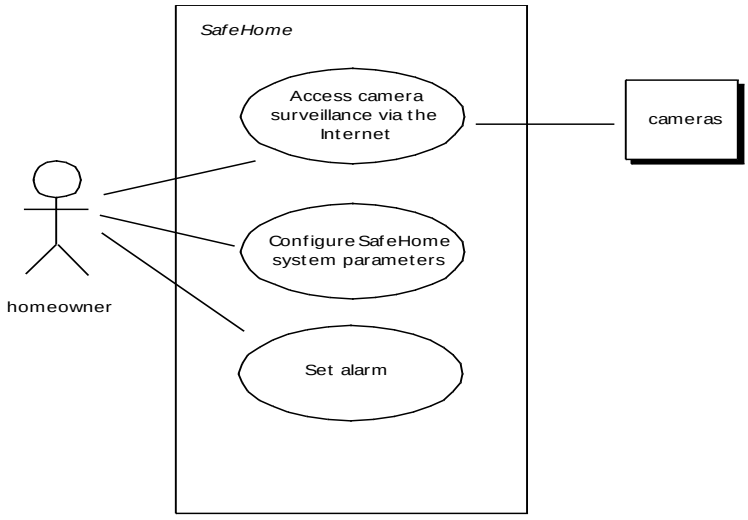
The concept is relatively easy to understand- describe a specific usage scenario in straightforward language from the point of view of a defined actor.

Use case diagram:

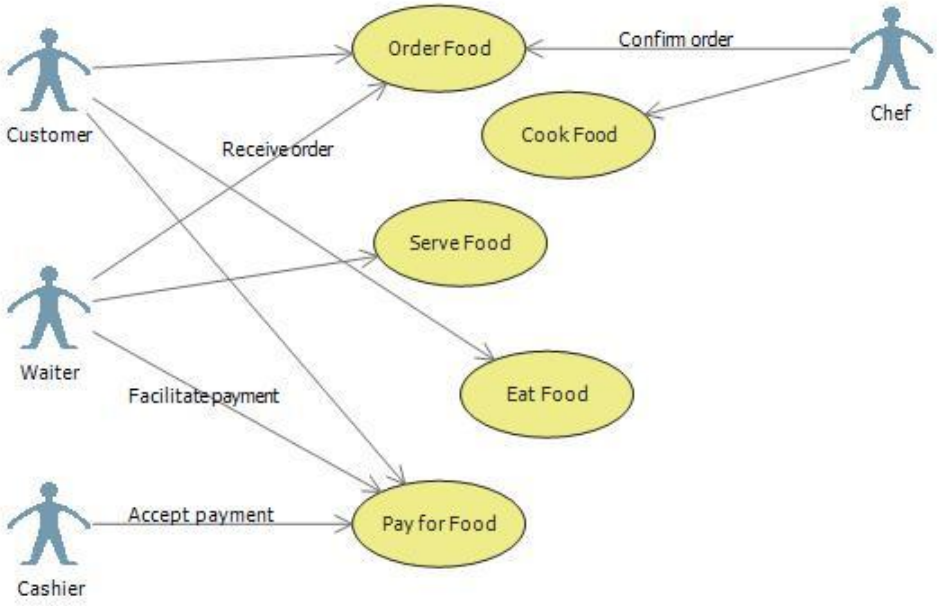
A use case diagram contains four components.

- The boundary, which defines the system of interest in relation to the world around it.
- The actors, usually individuals involved with the system defined according to their roles.
- The use cases, which are the specific roles played by the actors within and around the system.
- The relationships between and among the actors and the use cases.

- A scenario that describes a "thread of usage" for a system.
- ***Actors*** represent roles people or devices play as the system functions.
- ***Users*** can play a number of different roles for a given scenario.



Example 1: use case diagram for hotel management system:



Example 2: use case diagram for ATM system:

Class based modelling:

Class:

A class is represented by a rectangle having three sections:

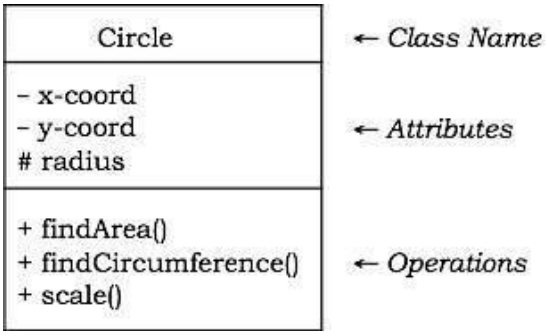
- the top section containing the name of the class
- the middle section containing class attributes
- the bottom section representing operations of the class

The visibility of the attributes and operations can be represented in the following ways:

Public : A public member is visible from anywhere in the system. In class diagram, it is prefixed by the symbol '+'.

Private : A private member is visible only from within the class. It cannot be accessed from outside the class. A private member is prefixed by the symbol '−'.

Protected : A protected member is visible from within the class and from the subclasses inherited from this class, but not from outside. It is prefixed by the symbol '#'.



An object is represented as a rectangle with two sections: The top section contains the name of the object with the name of the class or package of which it is an instance of. The name takes the following forms:

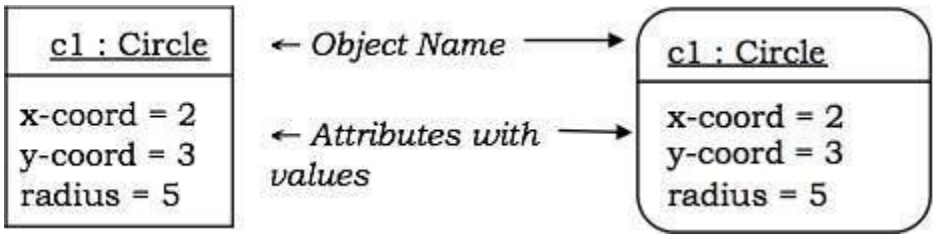
object-name : class-name

object-name : class-name :: package-name

class-name : in case of anonymous objects

The bottom section represents the values of the attributes. It takes the form attribute-name = value. Sometimes objects are represented using rounded rectangles.

Example : Let us consider an object of the class Circle named c1. We assume that the center of c1 is at (2, 3) and the radius of c1 is 5. The following figure depicts the object.



Class Responsibility Collaborator (CRC) Modeling

Class-Responsibility-Collaborator (CRC) Modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirement.

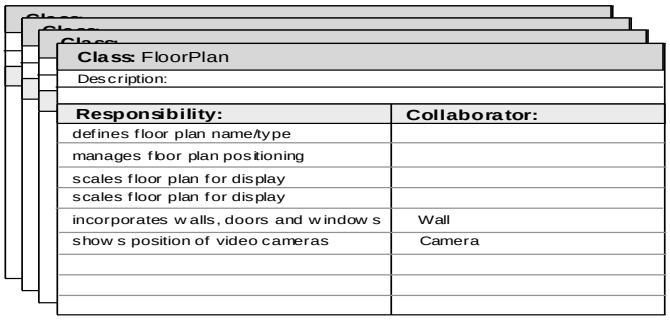
CRC modeling is described as follows:

"A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right."

Responsibilities are the attributes and operations that are relevant for the class. "Anything the class knows or does."

Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility.

In general, collaboration implies either a request for information or a request for some action.



Example 1:class diagram for railway management system:

Data Modeling:

1.data object

A *data object* is a representation of almost any composite information that must be processed by software. By composite, we mean something that has a number of different properties and attributes.

– "Width" (a single value) would not be a valid data object, but **dimensions** (incorporating height, width and depth) could be defined as object.
A data object encapsulates data only – there is no reference within a data object to operations that act on the data. Therefore, the data can be represented as a table below.

For example: if FAN is data object then it having different attributes as name,weight,height...etc

2.Data Attributes

Data attributes define the properties of a data object and take one of three different characteristics. They can be used to:

1. Name an instance of the data object.
2. Describe the instance, or
3. Make reference to another instance in another table.

In addition, one or more of the attributes, must be defined as an identifier, i.e., the identifier attribute becomes a "key" when we want to find an instance of the data object. Values for the identifier(s) are unique, although this is not a requirement.

Referring to the data object **car**, a reasonable identifier might be the ID number.

3.Relationships

Indicates "connectedness"; a "fact" that must be "remembered" by the system and cannot or is not computed or derived mechanically

- several instances of a relationship can exist
- objects can be related in many different ways

We can define a set of object/relationship pairs that define the relevant relationships. For example:

- A person *owns* a car.
- A person is *insured to drive* a car.

The relationship *owns* and *insured to drive* define the relevant connections between **person** and **car**.

Flow Oriented Modeling:

Represents how data objects are transformed as they move through the system.

A *data flow diagram* (DFD) is the diagrammatic form that is used to complement UML diagrams.

Considered by many to be an 'old school' approach, flow-oriented modeling continues to provide a view of the system that is unique.

The DFD takes an input-process-output insight into system requirements and flow.

Data objects are represented by labeled arrows and transformations are represented by circles (called *bubbles).*

Creating a Data Flow Model

The DFD diagram enables the software engineer to develop models of the information domain and functional domain at the same time.

As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system.

The flow Model

External Entity

A producer or consumer of data

Example: computer-based system

Data must always originate somewhere and must always be sent to something

Process

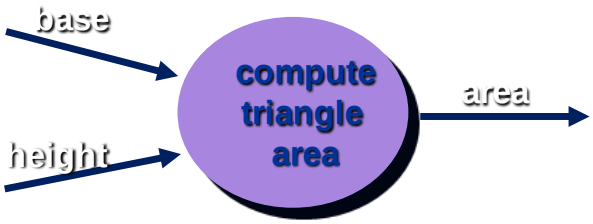
A data transformer (changes input to output)

Examples: compute taxes, determine area, format report, display graph

Data must always be processed in some way to achieve system function

Data Flow

Data flows through a system, beginning as input and be transformed into output.

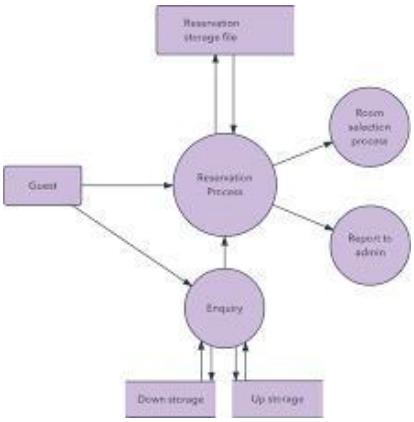


Data Flow Diagramming:

Constructing a DFD

- Review the data model to isolate data objects and use a grammatical parse to determine "operations"
- Determine external entities (producers and consumers of data)
- Create a level 0 DFD

Example:



Activity Diagram:

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc

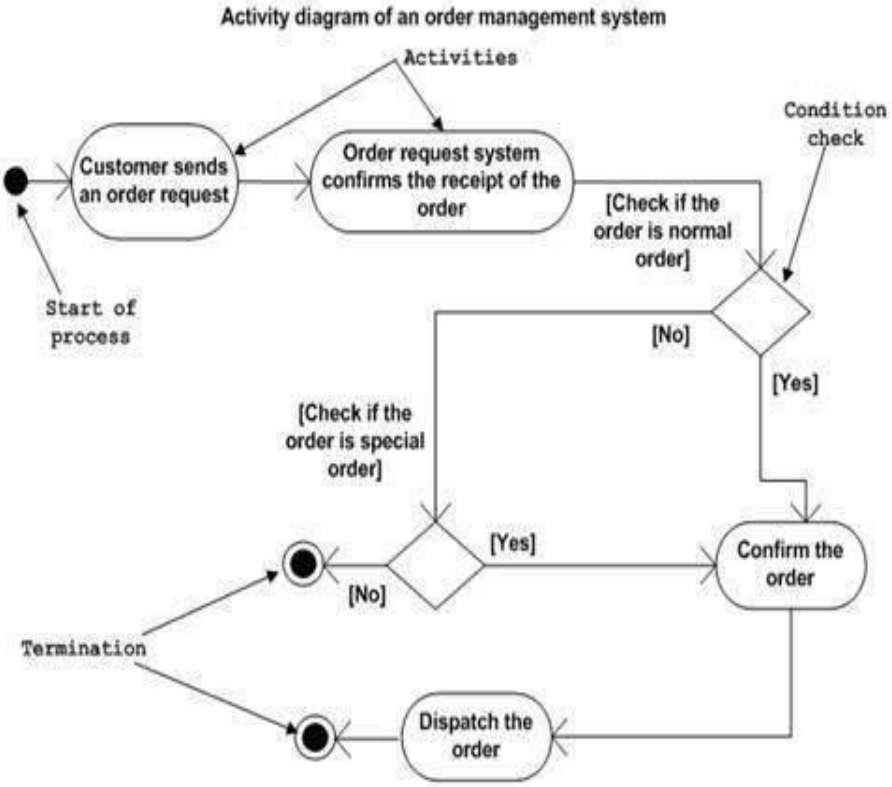
Purpose of Activity Diagrams

The basic purposes of activity diagrams is similar to other four diagrams. It captures the dynamic behavior of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques.

Before drawing an activity diagram, we should identify the following elements −

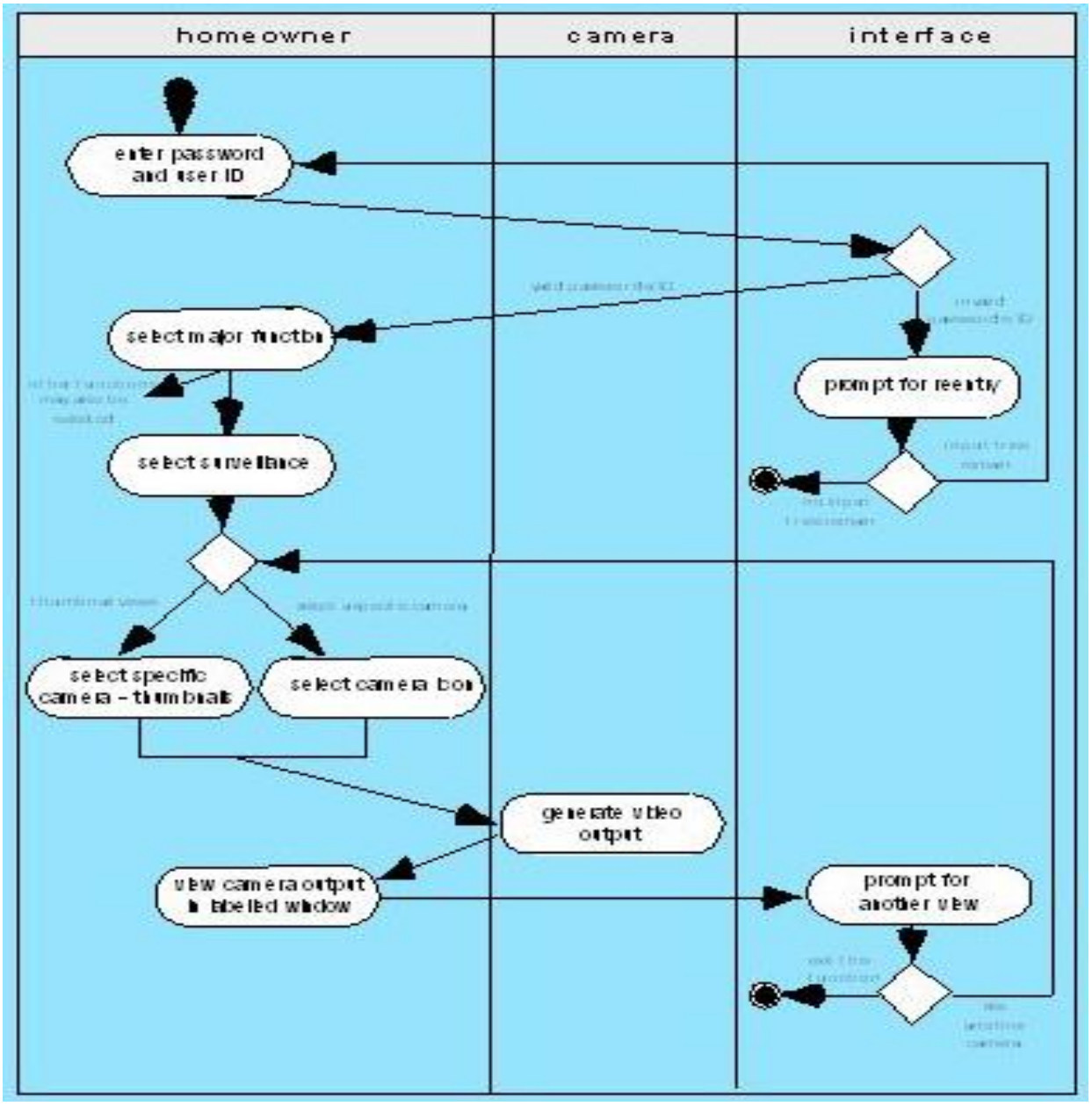
- Activities
- Association
- Conditions
- Constraints



Swimlane Diagrams

The UML *swimlane diagram* is a useful variation of the activity diagram and allows the modeler to represent the flow of activities described by the user-case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle.

Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.


homeowner
camera
interface
enter password and user ID
prompt for reentry
select surveillance
select specific camera - thumbnails
generate video output
view camera output in labelled window
prompt for another view

State Chart Diagram:

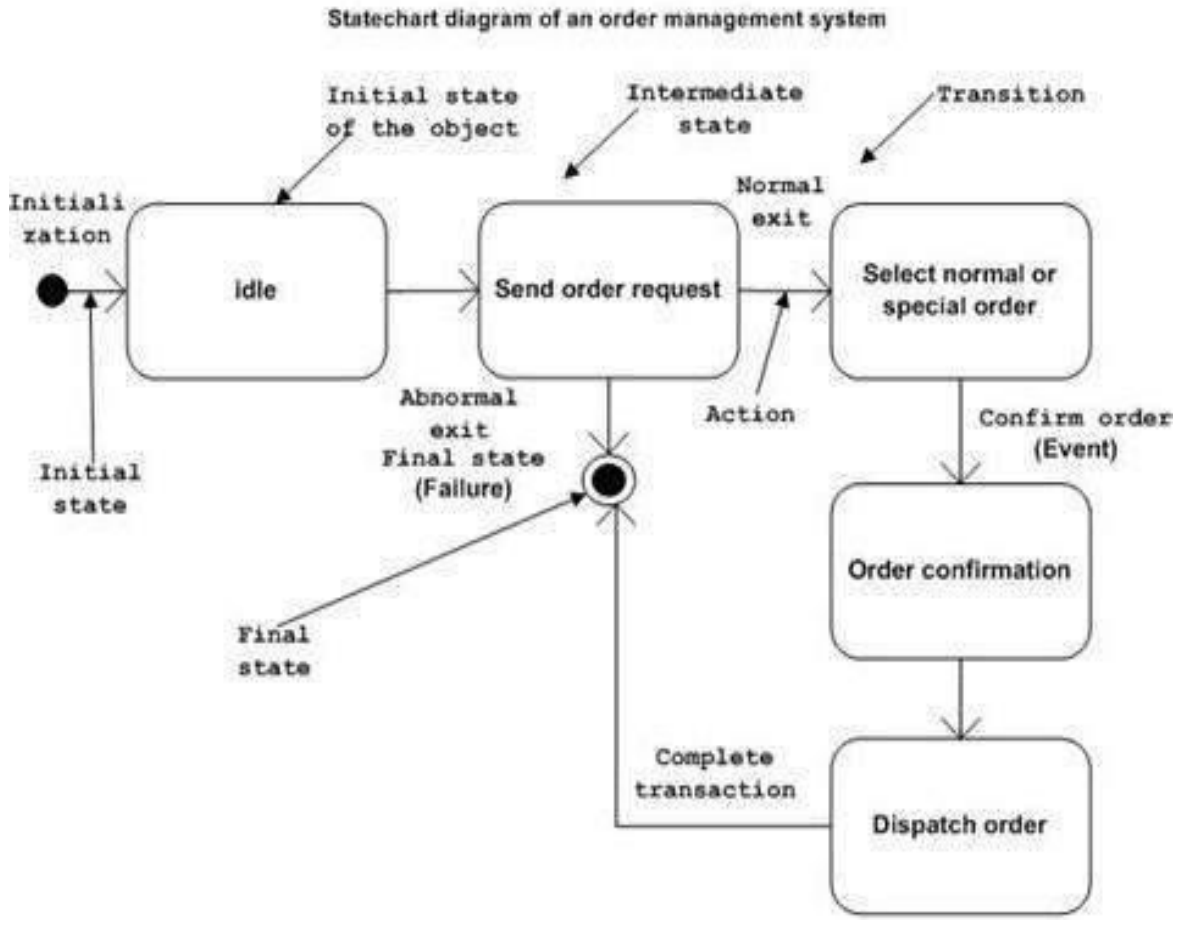
A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system

Following are the main purposes of using Statechart diagrams

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.



1.Design Model

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.

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.

Principles of Software Design

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

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

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

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

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

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

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

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

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

2.Software Engineering Design

Data/Class design - created by transforming the analysis model class-based elements (class diagrams, analysis packages, CRC models, collaboration diagrams) into classes and data structures required to implement the software

Architectural design - defines the relationships among the major structural elements of the software, it is derived from the class-based elements and flow-oriented elements (data flow diagrams, control flow diagrams, processing narratives) of the analysis model

Interface design - describes how the software elements, hardware elements, and end-users communicate with one another, it is derived from the analysis model scenario-based elements (use-case text, use-case diagrams, activity diagrams, swim lane diagrams), flow-oriented elements, and behavioral elements (state diagrams, sequence diagrams)

Component-level design - created by transforming the structural elements defined by the software architecture into a procedural description of the software components using information obtained from the analysis model class-based elements, flow-oriented elements, and behavioral elementsSoftware Design Concepts

---

3.Software Design Concepts.

1.Abstraction

Abstraction means hiding of information without providing internal details. It 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.

1.1 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.

1.2 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.

1.3 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.

2.Architecture

Software architecture refers to the structure of the system, which is composed of various components of a program/ system, the attributes (properties) of those components and the relationship among them. The software architecture enables the software engineers to analyze the software design efficiently. In addition, it also helps them in decision-making and handling risks. The software architecture does the following.

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

3.Patterns

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

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

Types of Design Patterns

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

3.1.Architectural patterns: These patterns are high-level strategies that refer to the overall structure and organization of a software system. That is, they define the elements of a software system such as subsystems, components, classes, etc. **In** addition, they also indicate the relationship between the elements along with the rules and guidelines for specifying these relationships. Note that architectural patterns are often considered equivalent to software architecture.

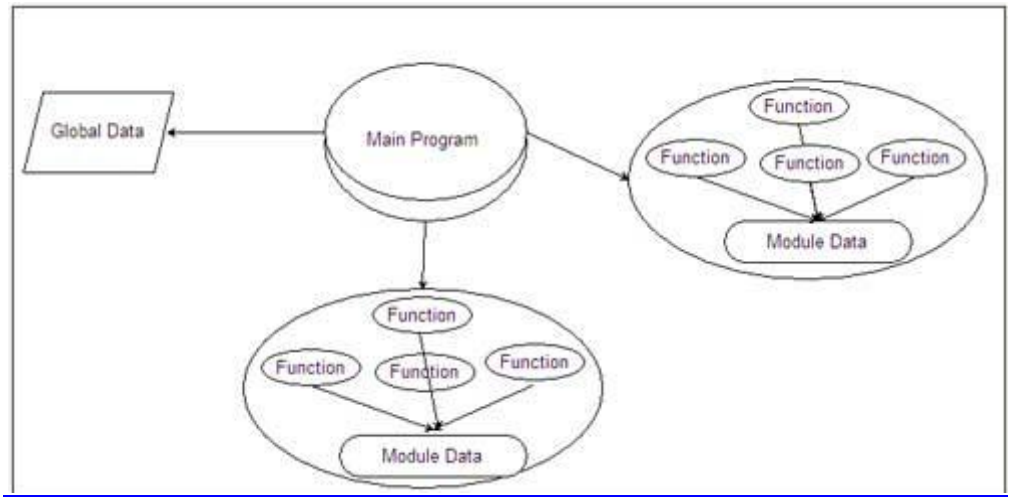
3.2.Design patterns: These patterns are medium-level strategies that are used to solve design problems. They provide a means for the refinement of the elements (as defined by architectural pattern) of a software system or the relationship among them. Specific design elements such as relationship among components or mechanisms that affect component-to-component interaction are addressed by design patterns. Note that design patterns are often considered equivalent to software components.

3.3.Idioms: These patterns are low-level patterns, which are programming-language specific. They describe the implementation of a software component, the method used for interaction among software components, etc., in a specific programming language. Note that idioms are often termed as coding patterns.

4.Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

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.

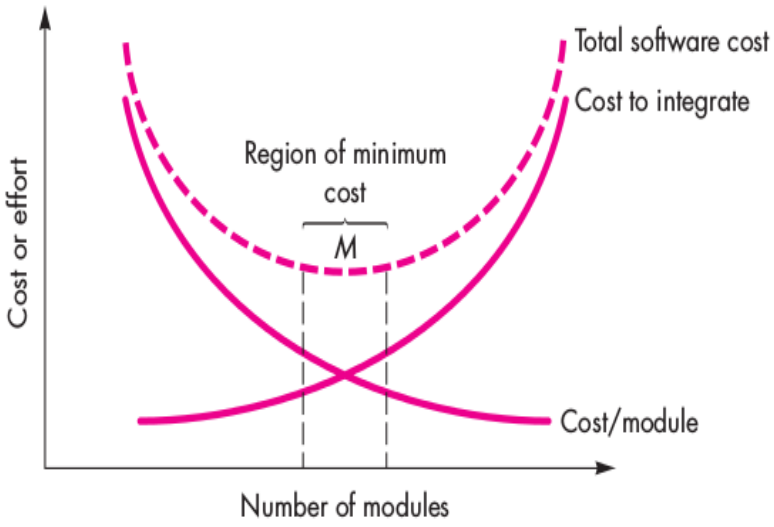


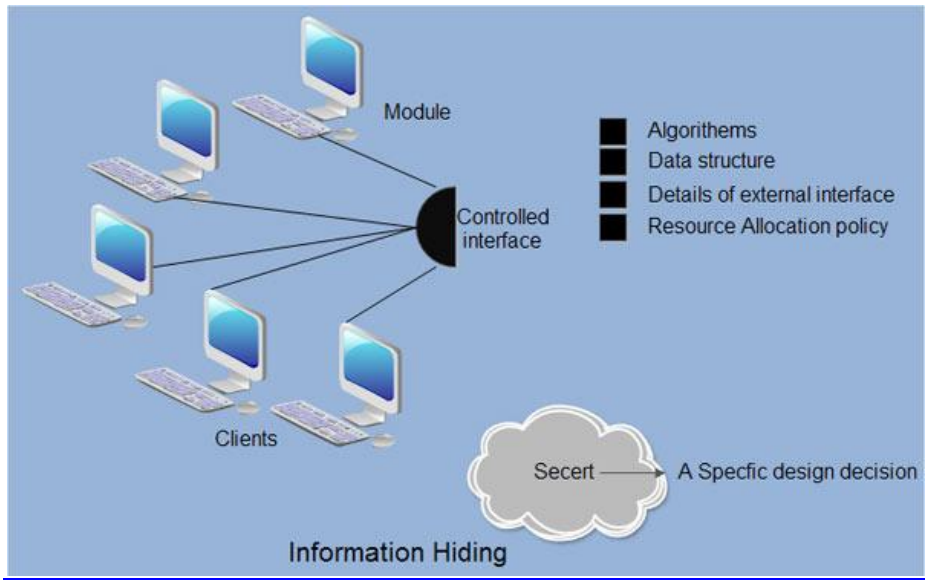
Fig: Modularity and software cost

The curves shown in Figure 8.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. **Under modularity** or **over modularity** should be avoided.

5.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.



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.

6.Refactoring

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

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

Structural Partitioning

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

- The testing and maintenance of software becomes easier.
- The negative impacts spread slowly.
- The software can be extended easily.

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

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

7.Functional Independence:

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with “single- minded” function and an “aversion” to excessive interaction with other modules.

Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and

test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.

Independence is assessed using two qualitative criteria: cohesion and coupling. Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules.

Cohesion is a qualitative indication of the degree to which a module focuses on just one thing.

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world.

8.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.

4.Developing a Design Model

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

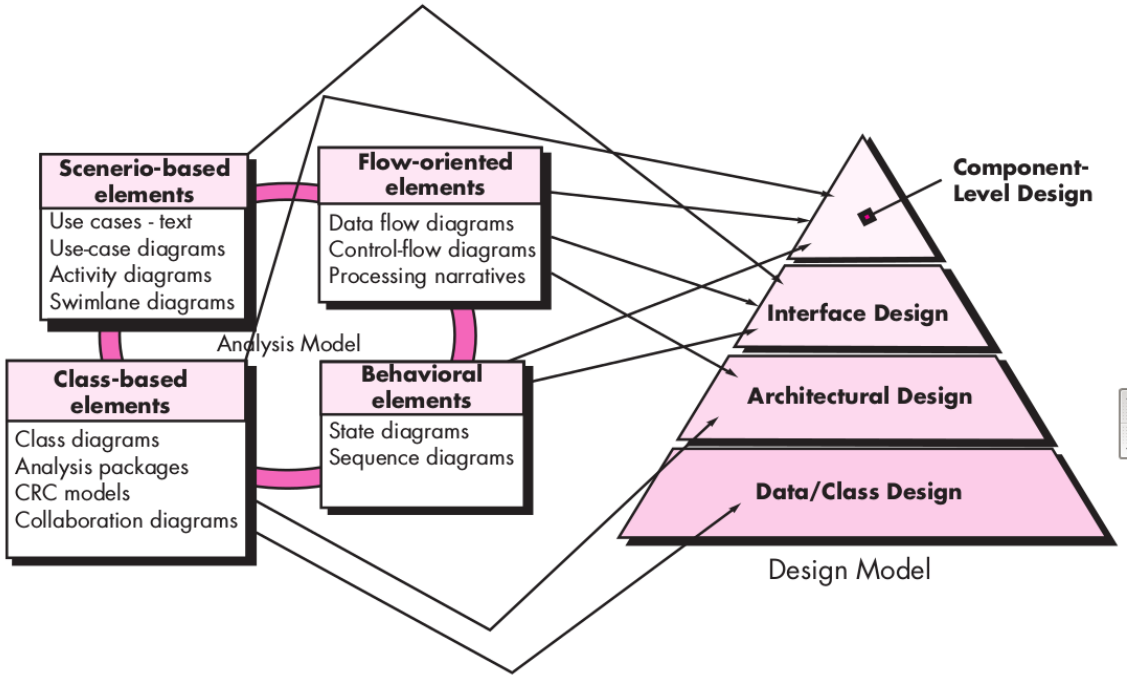
1.Data design: This specifies the data structures for implementing the software by converting data objects and their relationships identified during the analysis phase. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design 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 inter operates with it and with the end-users.

5.Translating Analysis model into Software Design



Each element of the analysis model provides information that is necessary to create the four design models

1. The data-class design transforms analysis classes into design classes along with the data structures required to implement the software .

2. The architectural design defines the relationship between ma(or structural elements of the software. architectural styles and design patterns help achieve the requirements defined for the system.

3. The interface design describes how the software communicates with systems that interoperate with it and with humans that use it.

4. The component/level design transforms structural elements of the software architecture into a procedural description of software component.

6.Pattern-Based Software Design

In software development, a pattern (or design pattern) is a written document that **describes a general solution to a design problem** that recurs repeatedly in many projects. Software designers adapt the pattern solution to their specific project.

Patterns use a formal approach to describing a design problem, its proposed solution, and any other factors that might affect the problem or the solution.

In object-oriented programming, a pattern can contain the description of certain objects and object classes to be used, along with their attributes and dependencies, and the general approach to how to solve the problem.

Programmers can use more than one pattern to address a specific problem. A collection of patterns is called a **pattern framework**

Design patterns include the following types of information:

5. Name that describes the pattern

6. Problem to be solved by the pattern

7. Context, or settings, in which the problem occurs

8. Forces that could influence the problem or its solution

9. Solution proposed to the problem

10. Rationale behind the solution (examples and stories of past successes or failures often go here)

11. Known uses and related patterns

12. Author and date information

13. References and keywords used or searching

14. Sample code related to the solution, if it helps

Describing Design Patterns

• Design patterns try to formally capture intuition or experience

– Reduce the need for super-star programmers: engineering not art

– Increase productivity: don't re-invent the wheel

– Increase reliability: record both the patterns and when they apply

Elements of a design pattern

–Name: we need a common name so people can talk about them

–Problem: simple description of what problem the pattern solves

–Context: where does the problem occur, and any background info

–Forces: any intrinsic tradeoffs that are being addressed

–Solution: how the pattern is applied

–Examples: application of the pattern to a real-world example

---

7.Coupling and Cohesion

Coupling:

An indication of the strength of interconnections between program units.

Highly coupled have program units dependent on each other. Loosely coupled are made up of units that are independent or almost independent.

In general, modules tightly coupled if they use shared variables or if they exchange control info.

Loose coupling if info held within a unit and interface with other units via parameter lists. Tight coupling if shared global data.

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

There are five levels of coupling, namely -

1. Content coupling - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
2. Common coupling- When multiple modules have read and write access to some global data, it is called common or global coupling.
3. Control coupling- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
4. Stamp coupling- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
5. Data coupling- Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Cohesion:

A component should implement a single logical function or single logical entity. All the parts should contribute to the implementation.

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

1. **Co-incidental cohesion** - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.

2. **Logical cohesion** - When logically categorized elements are put together into a module, it is called logical cohesion.

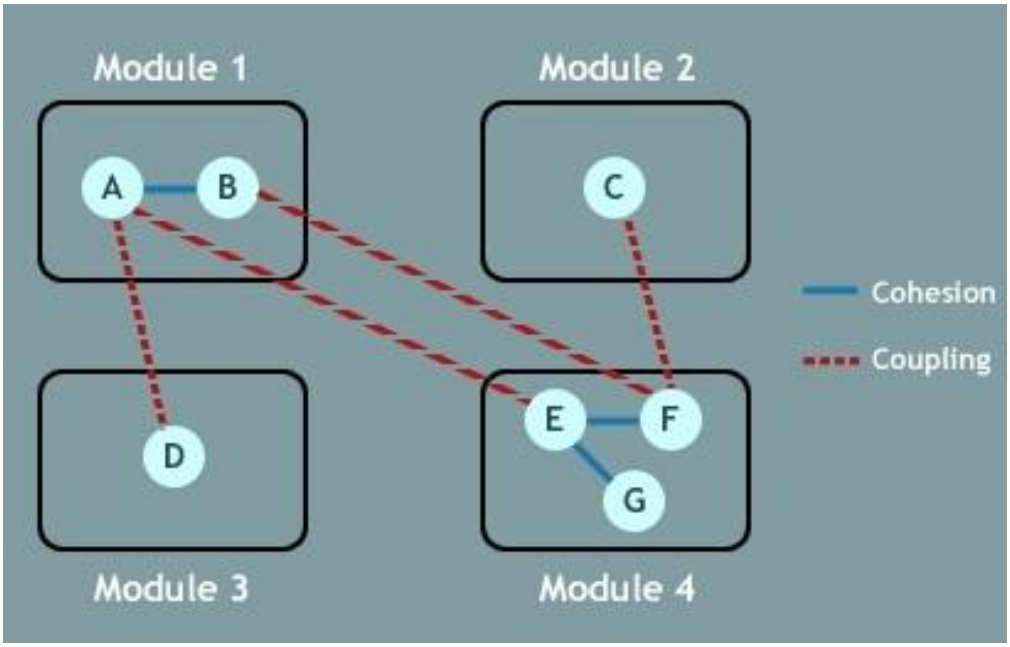
3. **Temporal Cohesion** - When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.

4. **Procedural cohesion** - When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.

5. **Communicational cohesion** - When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

6. **Sequential cohesion** - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

7. **Functional cohesion** - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It also be reused.



Cohesion	**Coupling**
Cohesion is the indication of the relationship within **module**.	**Coupling** is the indication of the relationships between modules.
Cohesion shows the module's relative **functional** strength.	Coupling shows the relative **independence** among the modules.
Cohesion is a degree (quality) to which a component / module focuses on the **single** thing.	Coupling is a degree to which a component / module is connected to the **other** modules.

While designing you should strive for **high cohesion** i.e. a cohesive component/ module focus on a single task (i.e., **single-mindedness**) with little interaction with other modules of the system.	While designing you should strive for **low coupling** i.e. **dependency** between modules should be less.
Cohesion is the kind of natural extension of data hiding for example, **class** having all members visible with a package having default visibility.	Making private fields, private methods and non public classes provides loose coupling.
Cohesion is **Intra – Module** Concept.	Coupling is **Inter -Module** Concept.

---

8. User interface analysis and Design

A user interface is the part of the system with which the users interact. It includes the screen displays that provide navigation through the system, the screens and forms that capture data, and the reports that the system produces.

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.

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:

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

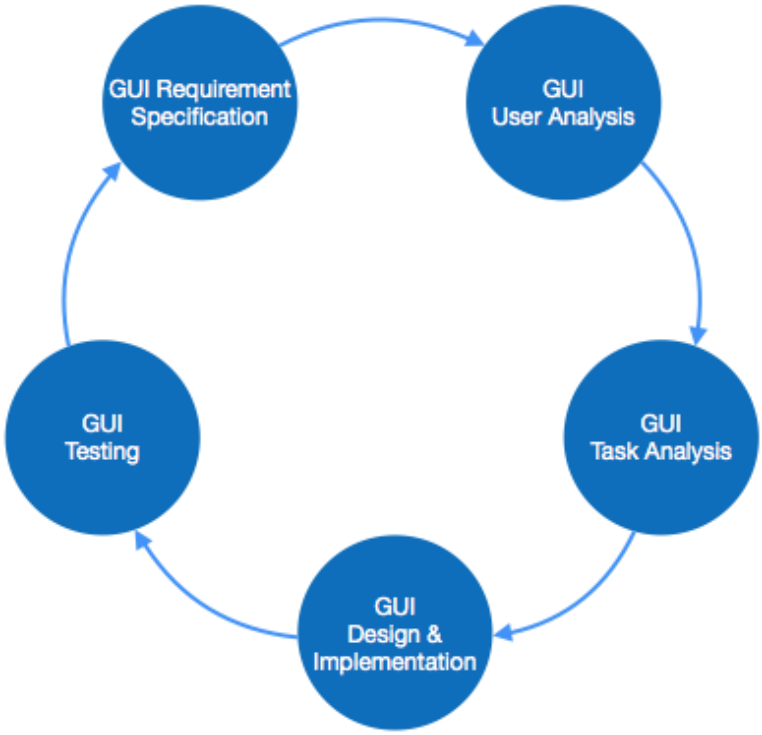
UI is broadly divided into two categories:

1. Command Line Interface
2. Graphical User Interface

8.1User 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.



1.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.

2.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.

3.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.

4.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.

5.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.

8.2User Interface Golden rules

The following rules are mentioned to be the golden rules for GUI design

1.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.

2.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.

3.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.

4.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.

5.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.

6.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.

7.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.

8.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.