SOFTWARE DESIGN

During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document.

Definition: The activities carried out during the design phase (called as design process) transform the SRS document into the design document.

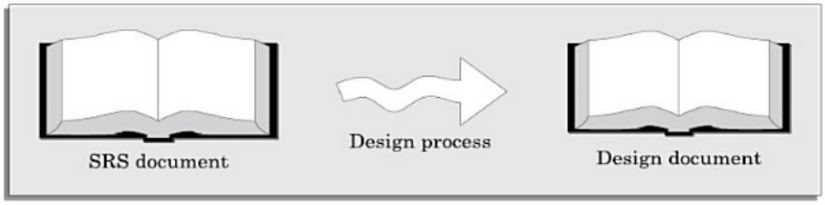


Figure 5.1: The design process.

Classification of Design Activities:

We can broadly classify into two important stages.

- Preliminary (or high-level) design, and
- Detailed design.

Preliminary (or high-level) design:

- A problem is decomposed into a set of modules. The control relationships among the modules are identified, and also the interfaces among various modules are identified.
- The outcome of high-level design is called the **program structure** or the **software architecture**

Detailed design:

- Once the high-level design is complete, detailed design is undertaken.
- During detailed design each module is examined carefully to design its **data structures** and the **algorithms**.

Characteristics GOOD SOFTWARE DESIGN

In fact, the definition of a "good" software design can vary depending on the exact application being designed. For example, "**memory size** used up by a program"

Characteristics of good Software Design are listed below:

- **Correctness**: A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.
- **Understandability**: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.
- **Efficiency**: A good design solution should adequately address resource, time, and cost optimization issues.
- **Maintainability**: A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

Understandability of a Design

- While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one. Obviously all incorrect designs have to be discarded first.
- Out of the correct design solutions, how can we identify the best one?

Understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design

NOTE:

A design solution should be modular and layered to be understandable

A good design follows:

(a). Modularity design principle

(b). Layered design principle

(a). Modularity:

1) A modular design is an effective decomposition of a problem.
2) It is a basic characteristic of any good design solution.
3) A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.
4) Decomposition of a problem into modules facilitates taking advantage of the divide and conquers principle.
5) If different modules have either no interactions or little interactions with each other, then each module can be understood separately.

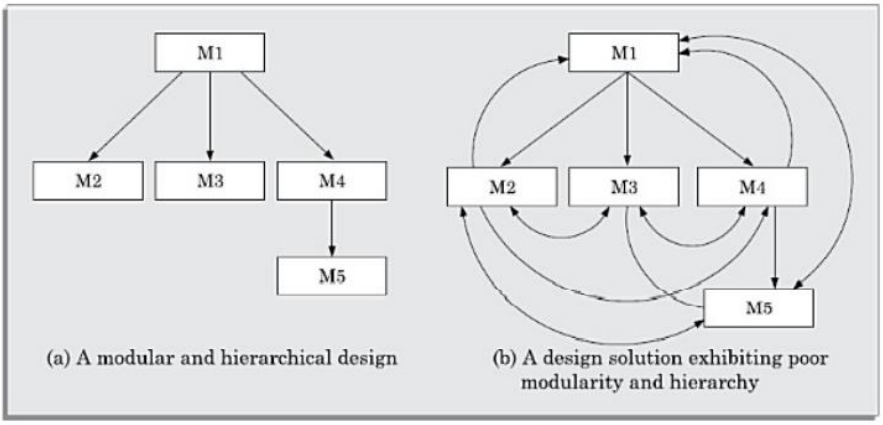


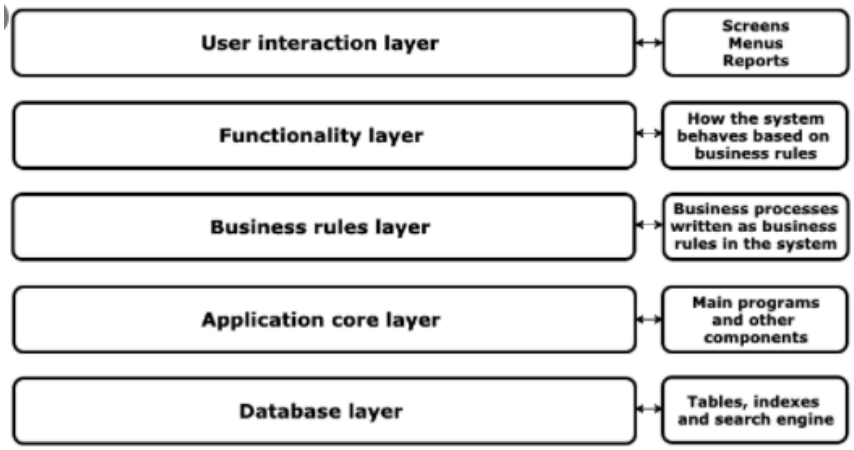
Figure 5.2: Two design solutions to the same problem.

Note:

A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.

(b). Layered design:

A layered design is one in which when the call **relations** among different modules are represented **graphically**, it would result in a tree-like diagram with clear layering. In a layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it.

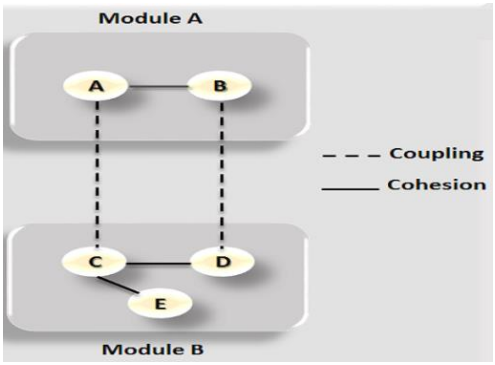


COHESION AND COUPLING

Cohesion: is a measure of the functional strength of a module.

Coupling: is is a measure of the degree of interaction (or **interdependence**) between the two modules

A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.



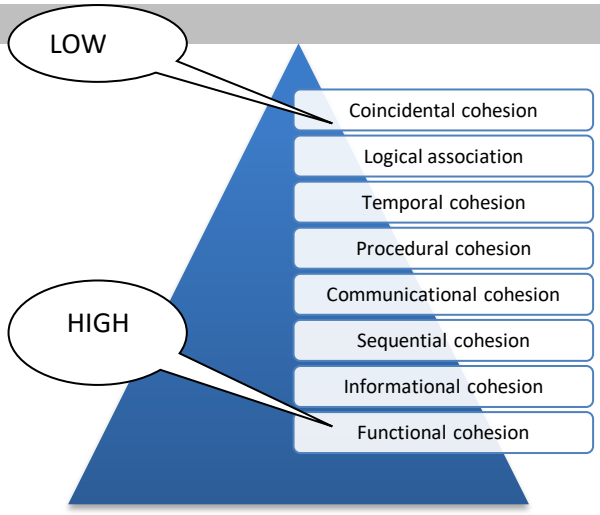
Difference between **COHESION AND COUPLING**

COHESION	COUPLING
[1]. Cohesion is the concept of **intra** module.	1. Coupling is the concept of **inter** module.
[2]. Cohesion represents the relationship **within** module.	2. Coupling represents the relationships **between** modules.
[3]. **Increasing** in cohesion is **good** for software.	3. **Increasing** in coupling is **avoided** for software.
[4]. Cohesion represents the **functional strength** of modules.	4. Coupling represents the **independence** among modules.
[5]. **Highly** cohesive gives the best software.	5. Whereas **loosely** coupling gives the best software.

Types/levels of Cohesion:

There are many levels of cohesion.

1. Coincidental cohesion
2. Logical association
3. Temporal cohesion
4. Procedural cohesion
5. Communicational cohesion
6. Sequential cohesion
7. Informational cohesion
8. Functional cohesion



1. Coincidental cohesion (worst)

Coincidental cohesion is when parts of a module are grouped arbitrarily;

Example:

Transaction Processing System: In a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.

2. Logical cohesion

Logical cohesion is when parts of a module are grouped because they are logically categorized to do the same thing, even if they are different by nature.

Examples:

Input handling routines: Grouping all mouse and keyboard handling routines

3. Temporal cohesion

Temporal cohesion is when parts of a module are grouped by when they are processed – the parts are processed at a particular time in program execution.

Examples:

The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion

4. Procedural cohesion

A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective.

Examples:

The algorithm for decoding a message

5. Communicational/informational cohesion

A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure.

Examples:

Module determines customer details like use customer account no to find and return customer name and loan balance.

6. Sequential cohesion

Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).

Examples:

In a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

7. Functional cohesion (best)

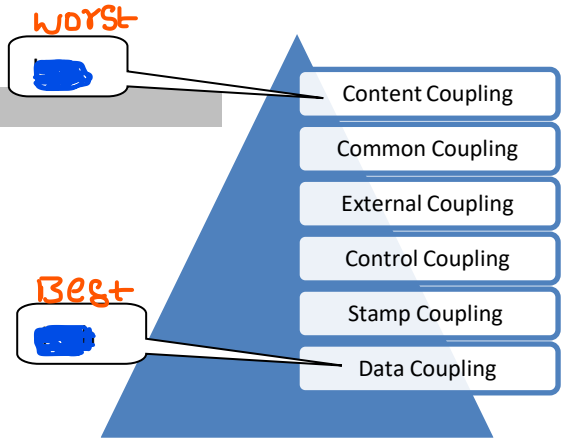
Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module (e.g. lexical analysis of an XML string).Focused (strong, single minded purpose) and no element doing unrelated activities

Examples:

Read transaction record

Types/levels of Coupling:

1. Content Coupling
2. Common Coupling
3. External Coupling
4. Control Coupling
5. Stamp Coupling
6. Data Coupling



1. **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.
2. **Common Coupling:** The modules have shared data such as global data structures.

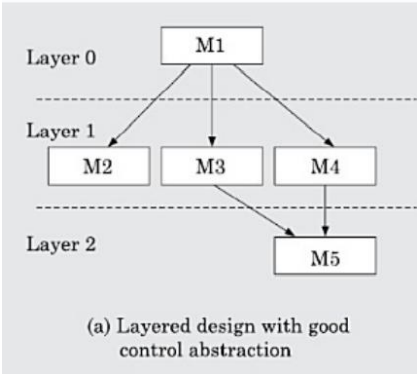
3. **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware.
   Ex- protocol, external file, device format, etc.
4. **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled.
   Example- sort function that takes comparison function as an argument
5. **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module..
6. **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.
   Example-customer billing system

Control Hierarchy

Control Hierarchy: The control hierarchy represents the organization of program components in terms of their call relationships. Thus we can say that the control hierarchy of a design is determined by **the order in which different modules call** each other.

Layering:

- In a layered design solution, the modules are arranged into several layers based on their call relationships. A module is allowed to call only the modules that are **at a lower layer**. That is, a module should not call a module that is either at a higher layer or even in the same layer.
- An important characteristic feature of a good design solution is layering of the modules. A layered design achieves **control abstraction** and is easier to understand and debug.
- In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower level module to discharge its responsibility.



(a) Layered design with good control abstraction

Terminologies associated with a layered design:

- **Superordinate and subordinate modules**: In a control hierarchy, a module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller.
- **Visibility**: A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.
- **Control abstraction**: In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.
- **Depth and width**: Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is **3** and width is also **3**.

- **Fan-out**: Fan-out is a measure of the **number of modules** that are **directly controlled** by a given module. In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very **high fan-out** numbers is **not a good design**. The reason for this is that a very high fan-out is an indication that the module **lacks cohesion**. A module having a large fan-out (greater than **7**) is likely to implement several different functions and not just a single cohesive function.
- **Fan-in**: Fan-in indicates the number of modules that **directly invoke a given module**. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

Software design approaches

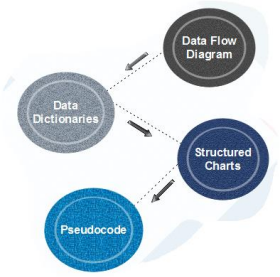
There are two fundamentally different approaches to software design that are in use today—

- function-oriented design,
- and object-oriented design

Function-oriented Design: A system is viewed as something that performs a set of functions. Starting at this high-level view of the system, each function is successively refined into more detailed functions. Example: consider a function create-new library-member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge.

Function Oriented Design Strategies:

1. Data Flow Diagram (DFD)
2. Data Dictionaries
3. Structure Charts
4. Pseudo Code



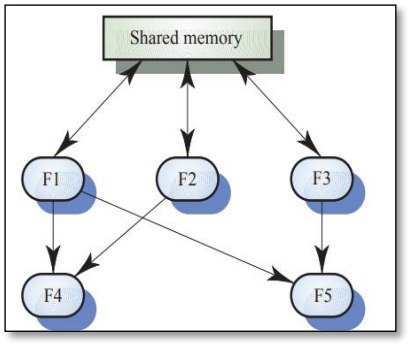
Function-Oriented Approach:

```
/* Global data (system state) accessible by various
functions */

BOOL detector_status[MAX_ROOMS];
int detector_locs[MAX_ROOMS];
BOOL alarm_status[MAX_ROOMS];
/* alarm activated when status is set */
int alarm_locs[MAX_ROOMS];
/* room number where alarm is located */
int neighbor-alarm[MAX_ROOMS][10];
/* each detector has at most 10 neighboring locations
*/

The functions which operate on the system state are:

interrogate_detectors();
get_detector_location();
determine_neighbor();
ring_alarm();
reset_alarm();
report_fire_location();
```

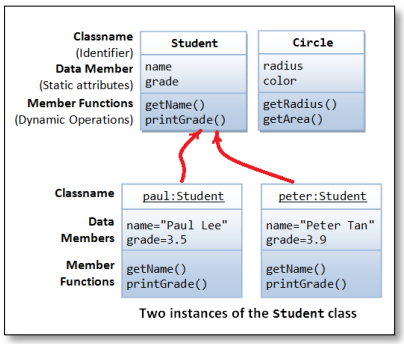


The following are the salient features of the function-oriented design approach:

Top-down decomposition: In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.

Centralized system state: The system state is centralized and shared among different functions.

Object-oriented design: In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information.



For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data.

Object-Oriented Approach:

```
class detector
attributes:
     status, location, neighbors

operations:
     create, sense_status, get_location,
     find_neighbors

class alarm
attributes:
     location, status

operations:
     create, ring_alarm, get_location,
     reset_alarm
```

Object Oriented Design Strategies

1. Class
2. Attributes
3. Objects
4. Methods (Behaviour)
5. Message

Function-oriented vs. object-oriented design approach

COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions, which are given to the user, are real world **functions**.	The basic abstractions are **not** the real world functions but are the **data abstraction** where the real world entities are represented.
Function	Functions are grouped together by which a higher level function is obtained.	Function are grouped together on **the basis of the data they operate** since the classes are associated with their methods.
State information	In this approach the state information is often represented in a centralized **shared memory**.	In this approach the state information is **not represented is not represen**ted in a centralized memory but is implemented or distributed among the objects of the system.
Approach	It is a top down approach.	It is a bottom up approach.
Begins basis	Begins by considering the **use-case diagrams** and the scenarios.	Begins by identifying **objects and classes**.
Decompose	In function oriented design we decompose in **function**/procedure level.	We decompose in **class** level.
Use	This approach is mainly used for **computation** sensitive application.	This approach is mainly used for evolving system which mimics **business case**.

Overview of SA/SD methodology

The Structured Analysis (SA)/Structured Design (SD) technique can b e used to perform the high-level design of software.

- SA: Structured analysis is to capture the detailed structure of the system as perceived by the user.
- SD: Structured design is to define the structure of the solution that is suitable for implementation in some programming language.
- ➢ During structured analysis, the SRS document is transformed into a data flow diagram (DFD).
- ➢ During structured design, the DFD model is transformed into a structure chart.

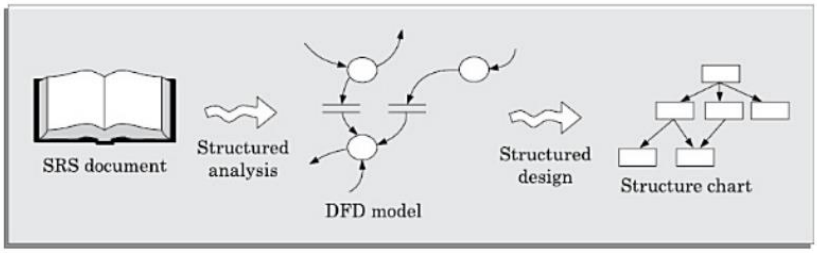


Figure 6.1: Structured analysis and structured design methodology.

As shown in Figure 6.1, the structured analysis activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved. On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure.

Structured Analysis

The structured analysis technique is based on the following underlying principles:

1. Top-down decomposition approach.
2. Application of divide and conquer principle. Through this each high level function is independently decomposed into detailed functions.
3. Graphical representation of the analysis results using data flow diagrams (**DFDs**).

Data Flow Diagrams (DFDs)

1. A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.
2. DFD model only represents the **data flow** aspects and **does not** show the *sequence* of execution of the different functions and the *conditions* based on which a function may or may not be executed.
3. In the DFD terminology, each function is called a **process** or a **bubble**. It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.
4. Starting with a set of high-level functions that a system performs, a DFD model represents the sub-functions performed by the functions using a hierarchy of diagrams.

Data Dictionaries

Overview:

Data Dictionary is the major component in the structured analysis model of the system. A data dictionary in Software Engineering means a file or a set of files that includes a database's **metadata** (hold records about other objects in the database), like data ownership, relationships of the data to another object, and some other data**.**

The data dictionary, in general, includes information about the following:

- Name of the data item
- Aliases
- Description/purpose
- Related data items
- Range of values
- Data structure definition/Forms

Composite data items can be defined in terms of primitive data items using the following data definition operators.

Notations	Meaning
+	Denotes composition of two data items, e.g. a+b represents data a **and** b
[,,]	Represents **selection**, i.e. **any one** of the data items listed inside the square bracket can occur For example, [a,b] represents either a occurs **or** b occurs.
()	The contents inside the bracket represent **optional data** which **may or may not** appear. a+(b) represents either a or a+b occurs.
{}	Represents iterative data definition, e.g. {name}5 represents **five name data**. {name}* represents zero or more instances of name data.
=	represents equivalence, e.g. a=b+c means that a is a **composite** data item comprising of both b and c
/* */	Anything appearing within /* and */ is considered as **comment**.

Primitive symbols used for constructing DFDs

There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure 6.2. The meanings of these symbols are explained as follows:

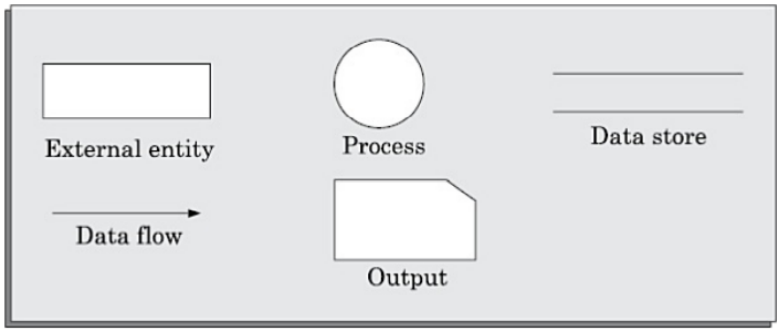


Figure 6.2: Symbols used for designing DFDs.

- **Function symbol**: A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions.

- **External entity symbol:** An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system.
- **Data flow symbol:** A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.
- **Data store symbol:** A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk.
- **Output symbol:** The output symbol is used when a hard copy is produced.

Synchronous and asynchronous operations:

- **Synchronous:** If two bubbles are **directly** connected by a data flow arrow, then they are synchronous.
  Example: Here, the validate-number bubble can start processing only after the read number bubble has supplied data to it; and the read-number bubble has to wait until the validate-number bubble has consumed its data.
- **Asynchronous:** if two bubbles are connected through a data store.

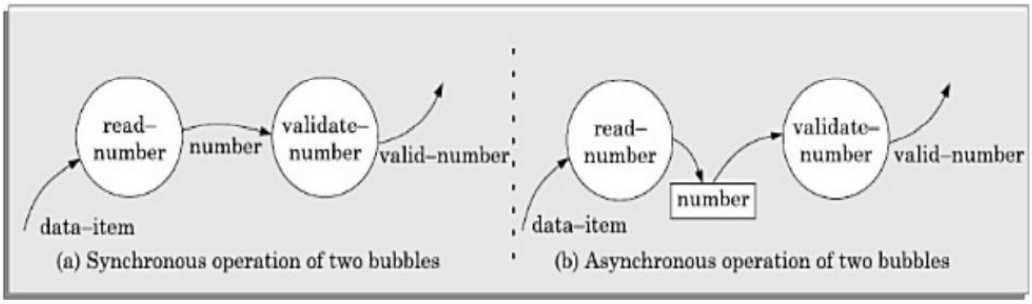


Figure 6.3: Synchronous and asynchronous data flow.

Levels in DFD

Levels in DFD are numbered 0, 1, 2 or beyond. Here, we will see primarily three levels in the data flow diagram, which are:

- 0-level DFD,
- 1-level DFD, and
- 2-level DFD.

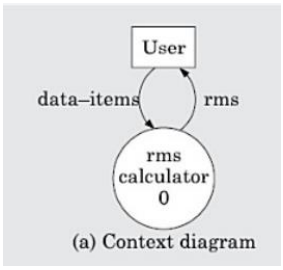
- **0-level DFDM (Context diagram)**
  The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble. The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). Ccontext diagram represents the entire software requirement as a single bubble with input and output data denoted by **incoming** and **outgoing** arrows.
- **Level 1 DFD**
  The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions.
- **Note**: What if a system has more than seven high-level requirements identified in the SRS document? In this case, some of the related requirements have to be combined and represented as a single bubble in the level 1 DFD.

- **Numbering of bubbles**

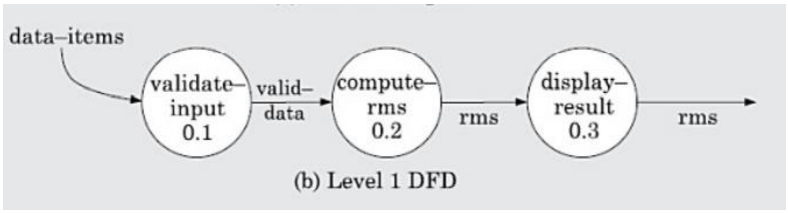
  It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc.

Ex:

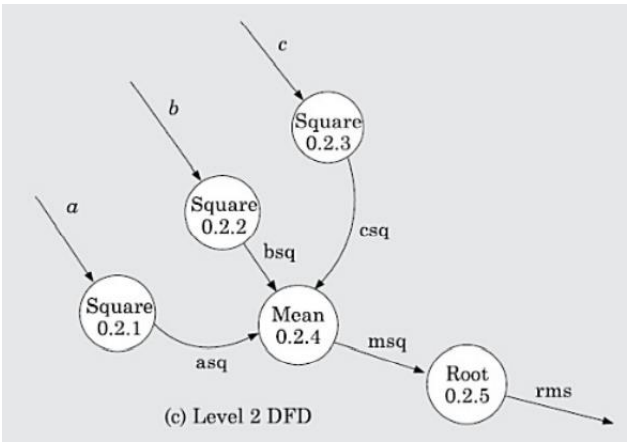
(RMS Calculating Software) A software system called RMS calculating software would read three integral numbers from the user in the range of –1000 and +1000 and would determine the root mean square (RMS) of the three input numbers and display it.



(a) Context diagram



(b) Level 1 DFD



(c) Level 2 DFD

Commonly made errors while constructing a DFD model

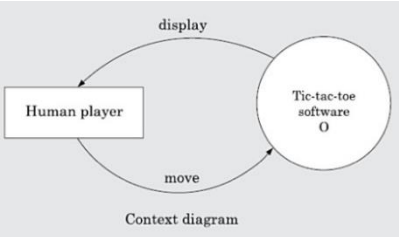
- Many beginners commit the mistake of drawing more than one bubble in the context diagram. Context diagram should depict the system as a single bubble.
- Many beginners create DFD models in which external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level.
- It is a common oversight to have either too few or too many bubbles in a DFD. Only three to seven bubbles per diagram should be allowed. This also means that each bubble in a DFD should be decomposed three to seven bubbles in the next level.
- Many beginners leave the DFDs at the different levels of a DFD model unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD.

Data dictionary:

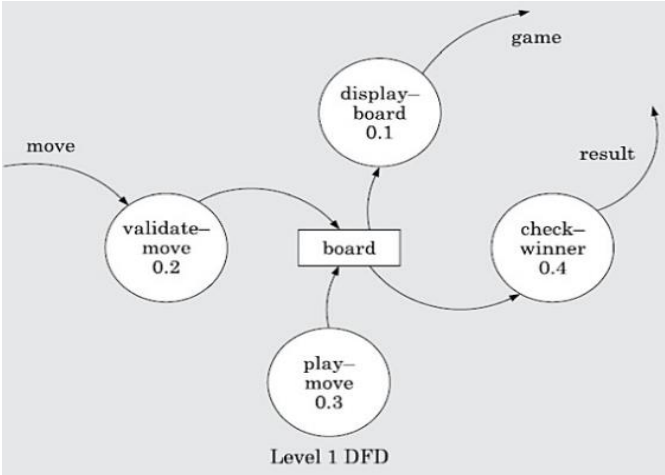
data-items: {integer}3 rms: float valid-data:data-items a: integer b: integer c: integer asq: integer bsq: integer csq: integer msq: integer

Example: Tic-Tac-Toe Computer Game:

Tic-tac-toe is a computer game in which a human player and the computer make alternate moves on a 3 × 3 square. A move consists of marking a previously unmarked square. The player who is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins. As soon as either of the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game. The context diagram and the level 1 DFD are shown in Figure 6.9.



Context diagram

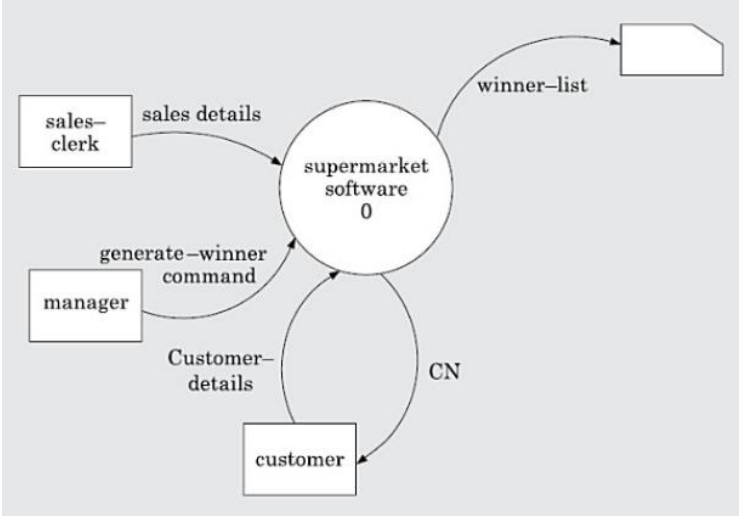


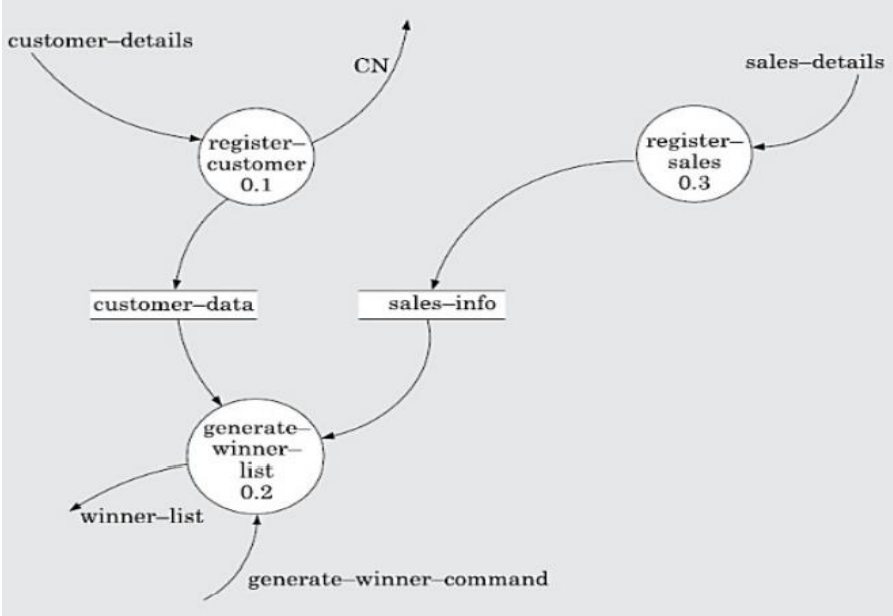
Level 1 DFD

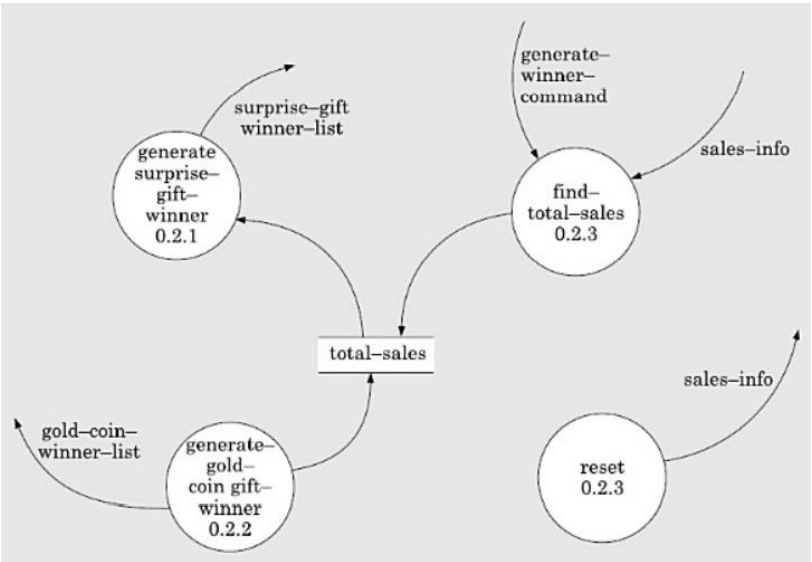
Data dictionary

1. move: integer /* number between 1 to 9 */
2. display: game+result
3. game: board
4. board: {integer}9
5. result: [“computer won”, “human won”, “drawn”]

Example: Supermarket Prize Scheme







Example 6.5 (Personal Library Software) Perform structured analysis for the personal library software of Example 6.5.

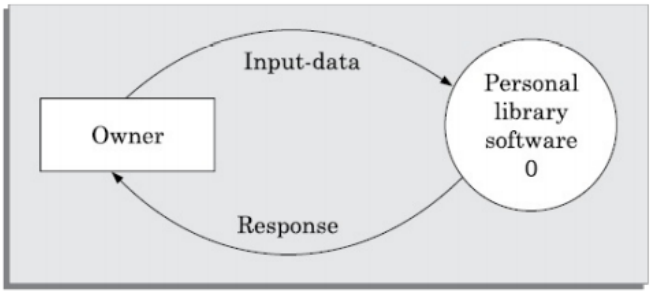


Figure 6.15: Context diagram for Example 6.5.

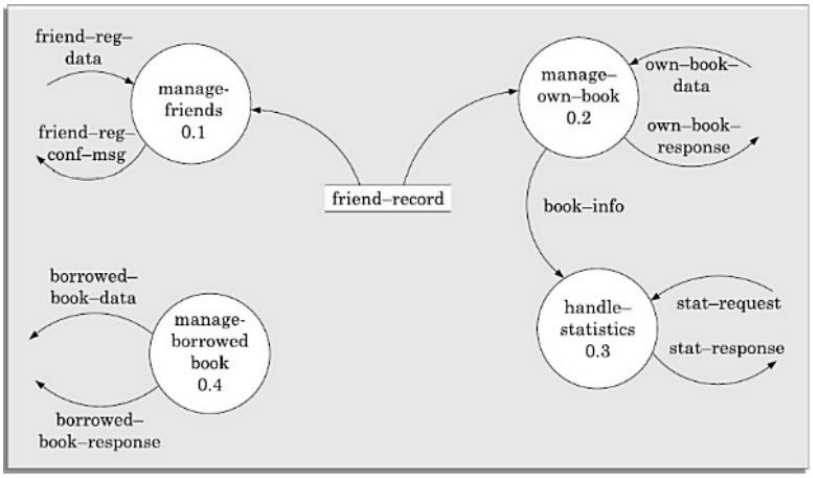


Figure 6.16: Level 1 DFD for Example 6.5.

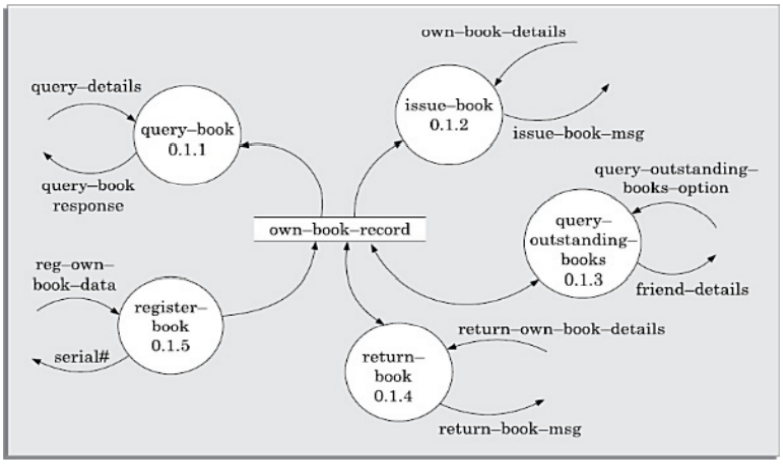


Figure 6.17: Level 2 DFD for Example 6.5.

Shortcomings of the DFD model

- We judge the function performed by a bubble from its label. However, a short label may not capture the entire functionality of a bubble
- Not-well defined control aspects are not defined by a DFD.
- The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst.
- For the same problem, several alternative DFD representations are possible

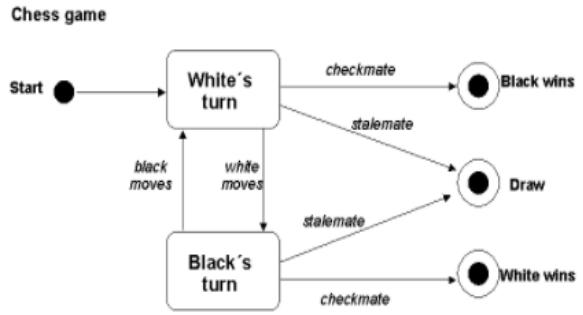
Extending DFD Technique to Make it Applicable to Real-time Systems

- In a real-time system, some of the high-level functions are associated with deadlines. Therefore, a function must not only produce correct results but also should produce them by some pre-specified time.
- For real-time systems, execution time is an important consideration for arriving at a correct design. Therefore, explicit representation of control and event flow aspects is essential. One of the widely accepted techniques for extending the DFD technique to real-time system analysis is the **Ward and Mellor technique**.
- In the Ward and Mellor notation, a type of process that handles only **Control flows** is introduced. These processes representing control processing are denoted using dashed bubbles. Control flows are shown using dashed lines/arrows.

Control specifications represent the behavior of the system in two different ways:

- It contains a **State Transition Diagram** (STD). The STD is a sequential specification of behavior.
- It contains a **Program Activation Table** (PAT). The PAT is a combinatorial specification of behavior.
- PAT represents invocation sequence of bubbles in a DFD.

State Transition Diagram (STD): Chess Game



Program Activation Table (PAT). : Vending Machine

Process Activated / Control Action	Dispense Change 2	Dispense Product 6	Get Valid Selection 5
Accept Customer Request	0	0	1
Return Payment	1	0	0
Accept New Coin	0	0	0
Despense Product	1	1	0

STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (that i s, the DFD model) into a structure chart.

The basic building blocks using which structure charts are designed are as following:

- **Rectangular boxes**: A rectangular box represents a **module**. Usually, every rectangular box is annotated with the name of the module it represents
- **Module invocation arrows**: An arrow **connecting two modules** implies that during program execution control is passed from one module to the other in the direction of the connecting arrow
- **Data flow arrows**: These are **small arrows** appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name.

- **Library modules**: A library module is usually represented by a rectangle with **double edges**. Libraries comprise the frequently called modules. Usually, when a module is invoked by many other modules, it is made into a library module.
- **Selection**: The **diamond** symbol represents the fact that one module of several modules connected with the diamond symbol i s invoked depending on the outcome of the condition attached with the diamond symbol.
- **Repetition**: A **loop around the control flow arrows** denotes that the respective modules are invoked repeatedly.

Flow chart versus structure chart

1. A structure chart differs from a flow chart in three principal ways:
2. It is usually difficult to identify the different modules of a program from its flow chart representation. Data interchange among different modules is not represented in a flow chart.
3. Sequential ordering of tasks that i s inherent to a flow chart is suppressed in a structure chart.

Transformation of a DFD Model into Structure Chart

Structured design provides two strategies to guide transformation of a DFD into a structure chart:

1. Transform analysis
2. Transaction analysis

✓ As in transform analysis, first **all data entering** into the DFD need to be identified.

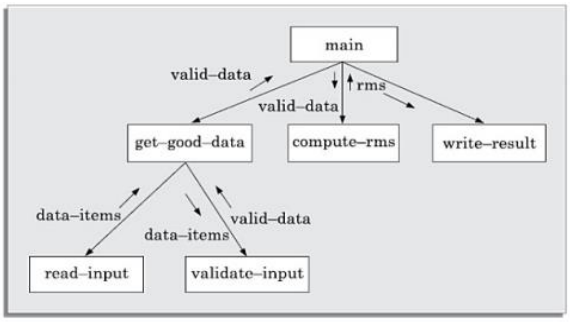
✓ In a transaction-driven system, different data items may pass through **different computation** paths through the DFD.

Normally, one would start with the level 1 DFD, transform it into module representation using either the transform or transaction analysis and then proceed toward the lower level DFDs.

1. Transform analysis:

Transform analysis identifies the primary functional components (modules) and the input and output data for these components. The first step in transform analysis is to divide the DFD into three types of parts:

• Input.
• Processing.
• Output

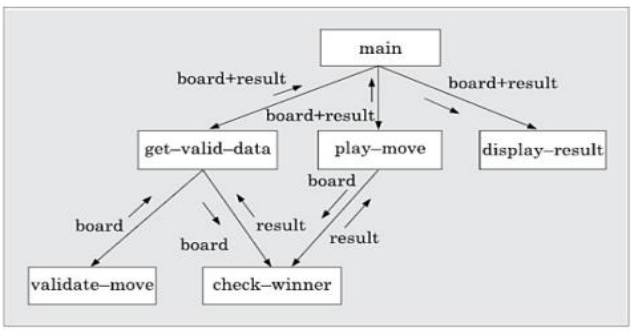


Draw the structure chart for the RMS software

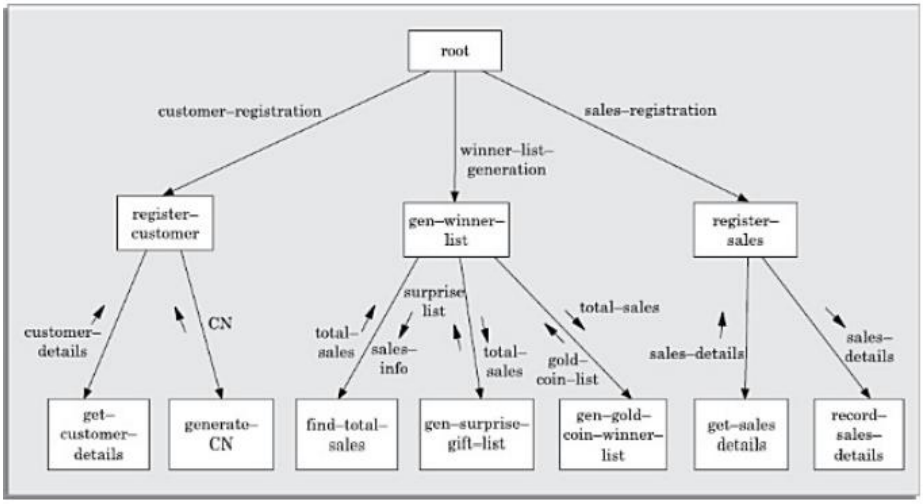
2. Transaction analysis

Transaction analysis is an alternative to transform analysis and is useful while designing **transaction processing programs**. A transaction allows the user to perform some specific type of work by using the software. For example, 'issue book', 'return book', 'query book', etc., are transactions.

Structure chart for the Tic-tac-toe software



Draw the structure chart for the Supermarket Prize:



DETAILED DESIGN:

During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart. These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English.

DESIGN REVIEW:

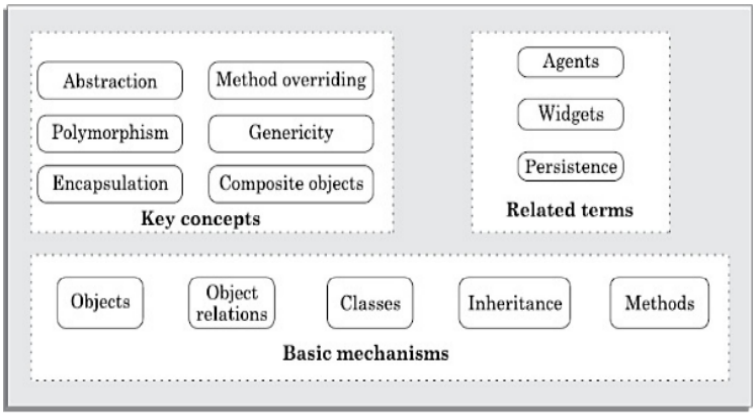
After a design is complete, the design is required to be reviewed. After a design is complete, the design is required to be reviewed.

- ✓ **Traceability**: Whether each bubble of the DFD can be traced to some module in the structure chart and vice versa. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model and vice versa.
- ✓ **Correctness**: Whether all the algorithms and data structures of the detailed design are correct.
- ✓ **Maintainability**: Whether the design can be easily maintained in future.
- ✓ **Implementation**: Whether the design can be easily and efficiently be implemented. After the points raised by the reviewers are addressed by the designers, the design document becomes ready for implementation.

BASIC OBJECT-ORIENTATION (OO) CONCEPTS

The following diagrams depict basics of Object-Orientation (OO) Concepts.

(a). Object: An object-oriented program usually represents a tangible real-world entity such as a library member, a book, an issue register, etc. Each object essentially consists of some data that is private to the object and a set of functions (termed as operations or methods) that operate on those data.



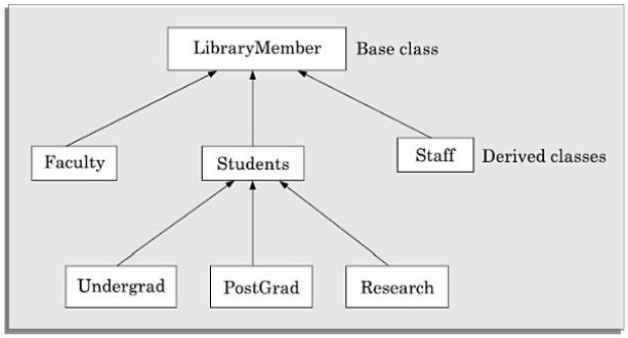
A key advantage of considering a system as a set of objects is the following:
When the system is analyzed, developed, and implemented in terms of objects, it becomes easy to understand the design and the implementation of the system, since objects provide an excellent decomposition of a large problem into small parts.

(b). Class: *Collection of objects* is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space. Class is an abstract data type (ADT).

Class Relationships: Classes in a programming solution can be related to each other in the Following four ways:

- Inheritance
- Association and link
- Aggregation and composition
- Dependency

- **Inheritance**:
  - When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism. . In Figure, observe that the *classes Faculty*, *Students*, and *Staff* have been derived from the base class LibraryMember through an inheritance relationship.



  - Each derived class can be considered as a **specialisation** of its base class because it modifies or extends the basic properties of the base class in certain ways. Therefore, the inheritance relationship can be viewed as a **generalisation-specialisation** relationship.
  - When a new definition of a method that existed in the base class is provided in a derived class, the method is said to be **overridden** in the derived class.

- **Association and link**:
  - Association is a common type of relation among classes.
  - When two classes are associated, the relationship between two objects of the corresponding classes is called a **link**. An association describes a group of similar links.
  - Consider the following example. A Student can register in **one** Elective subject. In this example, the class Student is associated with the class Elective Subject.
  - In **unary association**, two (or more) different objects of the same class are linked by the association relationship.

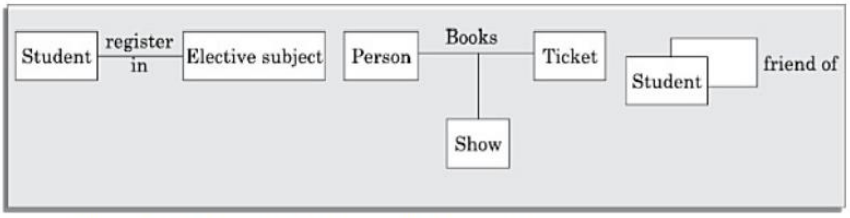


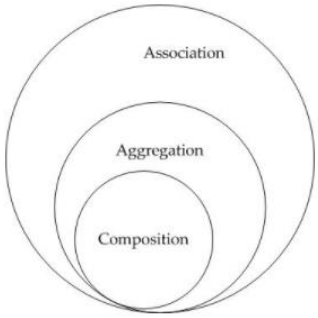
Figure 7.5: Example of (a) binary (b) ternary (c) unary association.

- **Composition and Aggregation:**
  - Composition and aggregation represent part/whole relationships among objects. Objects which contain other objects are called composite objects.
  - Example: A Book object can have upto ten Chapters. In this case, a Book object is said to be composed of upto ten Chapter objects. The composition/aggregation relationship can also be read as follows—"A Book **has** upto ten Chapter objects". The composition/aggregation relationship is also known as **has a relationship.**

Figure 7.6: Example of aggregation relationship.

  - **Dependency**: A class is said to be dependent on another class, if any changes to the latter class necessitates a change to be made to the dependent class.
  - **Abstract class:** Classes that are not intended to produce instances of them are called abstract classes. In other words, an abstract class cannot be instantiated.

Difference among Aggregation and Composition are Association



- In both aggregation and composition **object of one class "owns" object of another class**.
- But there is a subtle difference. In **Composition** the object of class that is owned by the object of it's owning class **cannot live on it's own** (Also called "death relationship").
- It will always live as a part of it's owning object where as in **Aggregation** the dependent object is **standalone** and can exist even if the object of owning class is dead.
- **Trick to remember the difference** : "has **A**" -**A**ggregation and "**O**wn" - c**O**mpositoin

	Aggregation	Composition
Life time	Have their own lifetime	Owner's life time
Child object	Child objects belong to a single parent	Child objects belong to a single parent
Relation	Has-A	Owns
Example	**Car and Driver**	**Car and Wheels**

How to Identify Class Relationships:

In the following, we give examples of a few key words (shown in italics) that indicate the specific relationships among two classes A and B:

(i).Composition

- B is a permanent part of A
- A is made up of Bs
- A is a permanent collection of Bs

(ii).Aggregation

- B is a part of A
- A contains B
- A is a collection of Bs

(iii). **Inheritance**

- A is a kind of B
- A is a specialisation of B
- A behaves like B

(iv).Association

- A delegates to B
- A needs help from B
- A collaborates with B.

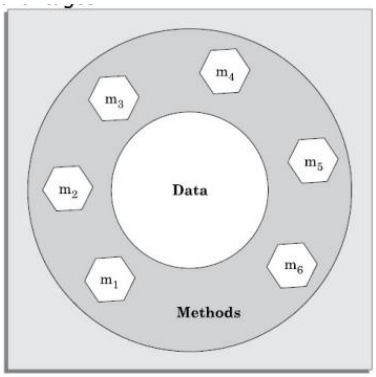
(c).Abstraction: The abstraction mechanism allows us to represent a problem in a simpler way by considering only those aspects that is relevant to some purpose and omitting all other details that are irrelevant. Abstraction is supported in two different ways in object-oriented designs (OODs). These are the following:

- **Feature abstraction:** inheritance mechanism can be thought of as providing feature abstraction
- **Data abstraction:** An object itself can be considered as a data abstraction entity, because it abstracts out the exact way in which it stores its various private data items and it merely provides a set of methods to other objects to access and manipulate these data items.

(d).Encapsulation: The data of an object is encapsulated within its methods. To access the data internal to an object, other objects have to invoke its methods, and cannot directly access the data.

Encapsulation offers the following three important advantages:

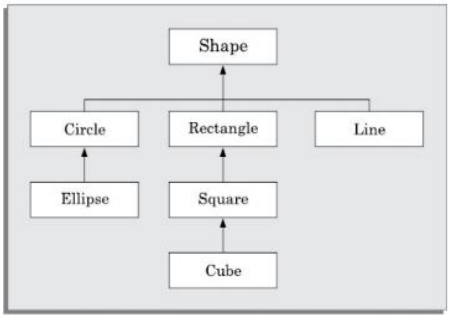
- Protection from unauthorised data access:
- Data hiding
- Weak coupling



(e).Polymorphism:

Polymorphism literally means poly (many) morphism (forms).

- **Static polymorphism**: Static polymorphism occurs when multiple methods implement the same operation but at compiled-time (statically).
- **Dynamic polymorphism**: Dynamic polymorphism is also called dynamic binding. In dynamic binding,the exact method that would be invoked (bound) on a method call can only be known at the **run time** (dynamically).



(f).Method overriding: When a new definition of a method that existed in the base class is provided in a derived class, the method is said to be **overridden** in the derived class.

Advantages of OOD:

The main reason for the popularity of OOD is that it holds out the following promises

1. Code and design reuse
2. Increased productivity
3. Ease of testing and maintenance

4. Better code and design understandability enabling development of large programs

Out of all the above mentioned advantages, it is usually agreed that the chief advantage of OOD is **improved productivity**—which comes about due to a variety of factors, such as the following:

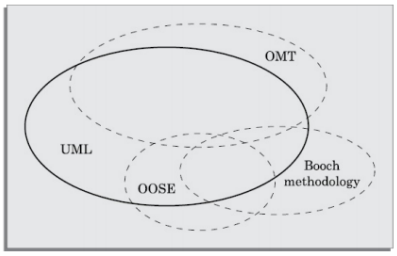
1. Code reuse by the use of predeveloped class libraries
2. Code reuse due to inheritance
3. Simpler and more intuitive abstraction, i.e., better management of inherent problem and code complexity
4. Better problem decomposition

Disadvantages of OOD:

1. Project-oriented program to run a little slower
2. Spatial locality of data becomes weak and this leads to higher cache miss ratios and consequently to larger memory access times

UML Diagrams

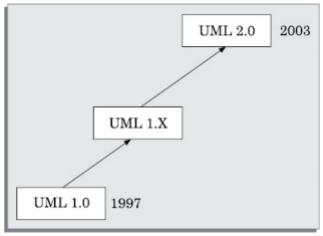
Unified Modeling Language (UML) can be used to document object-oriented analysis and design results that have been obtained using any methodology. UML was developed to standardize the large number of object-oriented modelling notations:



1. OMT [Rumbaugh 1991]
2. Booch's methodology [Booch 1991]
3. OOSE [Jacobson 1992]
4. Odell's methodology [Odell 1992]
5. Shlaer and Mellor methodology[Shlaer 1992]

As shown in Figure 7.12, OMT had the most profound influence on UML.

Evolution of UML:



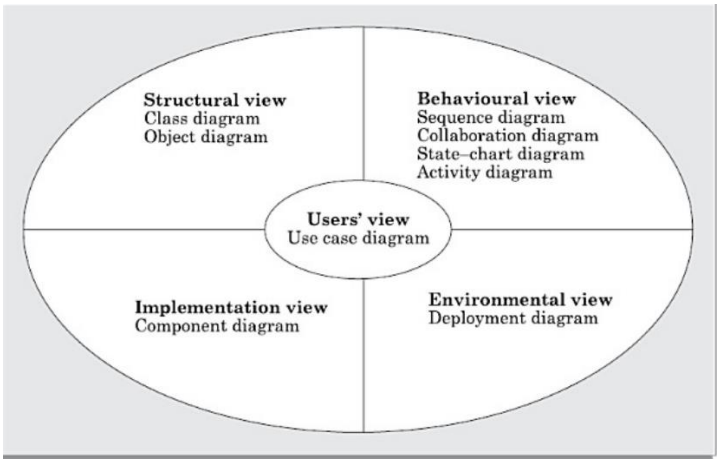
Model: A model is an abstraction of a real problem (or situation), and is constructed by leaving out unnecessary details. This reduces the problem complexity and makes it easy to understand the problem (or situation).

UML DIAGRAMS

UML diagrams can capture the following views (models) of a system:

1. User's view
2. Structural view
3. Behaviourial view
4. Implementation view
5. Environmental view

Figure 7.14 shows the different views that the UML diagrams can document.



1. **User View**: The users' view captures the view of the system in terms of the functionalities offered by the system to its users.
2. **Structural view**: The structural model is also called the **static** model, since the structure of a system does not change with time.
3. **Behavioral view:** The behavioral view captures how objects interact with each other in time to realise the system behavior. The system behavior captures the time-dependent (**dynamic**) behavior of the system. It therefore constitutes the dynamic model of the system.
4. **Implementation view:** This view captures the important components of the system and their interdependencies. For example, the implementation view might show the GUI part, the middleware, and the database part as the different parts and also would capture their interdependencies.
5. **Environmental view:** This view models how the different components are implemented on different pieces of hardware

USE CASE MODEL

- Intuitively, the use cases represent the different ways in which a system can be used by the users.
- A **use case** can be viewed as a set of related scenarios tied together by a common goal. The main line sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity.
- In contrast to all other types of UML diagrams, the use case model represents a **functional or process** model of a system.
- Both human users and external systems can be represented by **stick person** icons (Called **Actor**).
- When a stick person icon represents an external system, it is annotated by the stereotype <<**external system**>>.

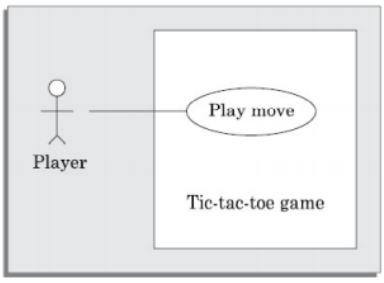
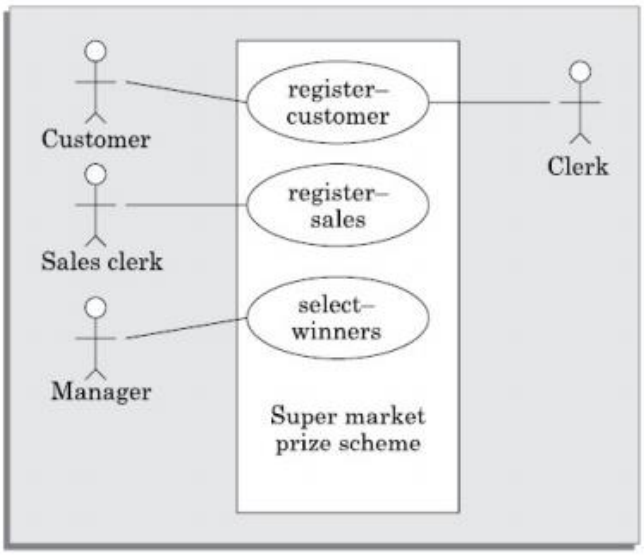


Figure 7.15: Use case model

Example: The use case model for the Tic-tac-toe game software.

The use case diagram of the Super market prize:



Text description

- **U1**: **register-customer**: Using this use case, the customer can register himself by providing the necessary details.
  - **Scenario 1**: Mainline sequence
    1. Customer: select register customer option
    2. System: display prompt to enter name, address, and telephone number.
    3. Customer: enter the necessary values
    4. System: display the generated id and the message that the customer has successfully been registered.
  - **Scenario 2**: At step 4 of mainline sequence

     4: System: displays the message that the customer has already registered.
  - **Scenario 3**: At step 4 of mainline sequence

     4: System: displays message that some input information have not been entered. The system displays a prompt to enter the missing values.
- **U2: register-sales**: Using this use case, the clerk can register the details of the purchase made by a customer.
  - **Scenario 1**: Mainline sequence
    1. Clerk: selects the register sales option.
    2. System: displays prompt to enter the purchase details and the id of the customer.
    3. Clerk: enters the required details.
    4. System: displays a message of having successfully registered the sale.
- **U3: select-winners**. Using this use case, the manager can generate the winner list.
  - **Scenario 1**: Mainline sequence
    1. Manager: selects the select-winner option.
    2. System: displays the gold coin and the surprise gift winner list.

Generalization: Use case generalisation can be used when you have one use case that is similar to another, but does something slightly differently or something more.

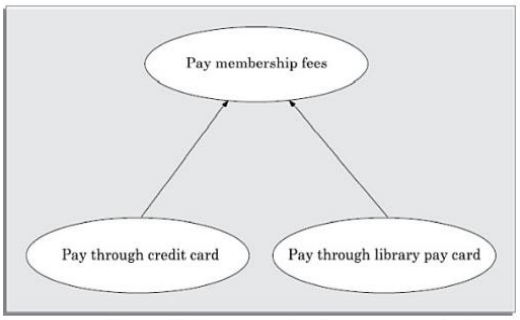
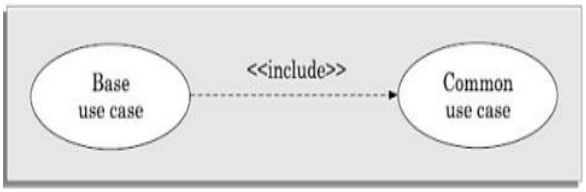


Figure 7.17: Representation of use case generalisation.

Includes: he includes relationship implies one use case includes the behaviour of another use case in its sequence of events and actions.



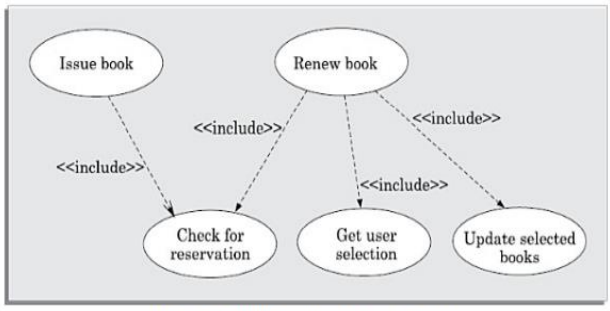
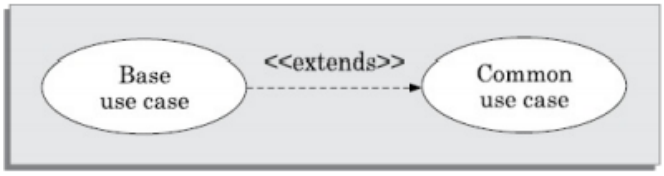
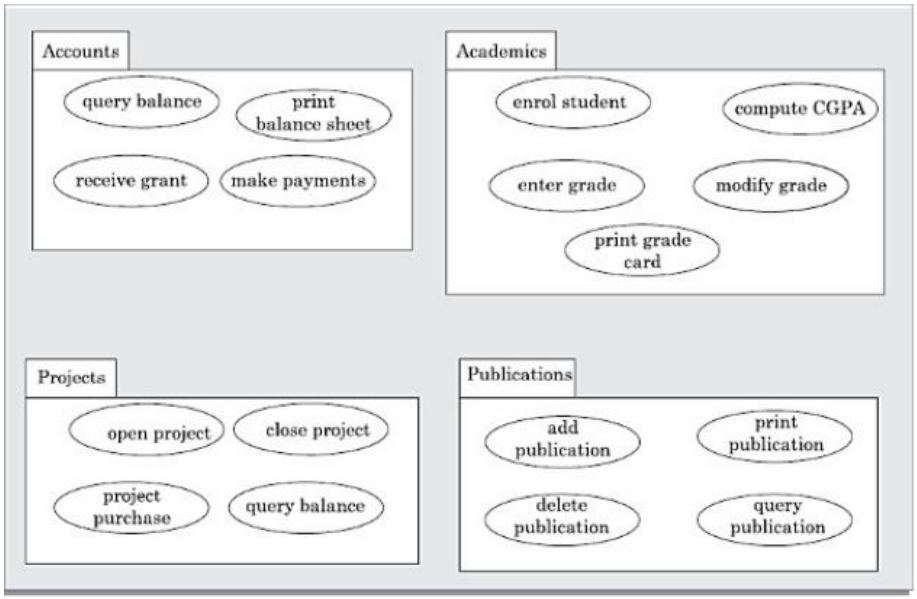


Figure 7.19: Example of use case inclusion.

Extends: The main idea behind the extends relationship among use cases is that it allows you show optional system behaviour. An optional system behaviour is executed only if certain conditions hold, otherwise the optional behaviour is not executed



USE CASE PACKAGING: Packaging is the mechanism provided by UML to handle complexity. When we have too many use cases in the top-level diagram, we can package the related use cases



Characteristics of a Good User Interface

The user interface portion of a software product is responsible for all interactions with the user.

1. **Speed of learning**:
    a. A good user interface should be easy to learn.
    b. Speed of learning is hampered by complex syntax and semantics of the command issue procedures.
    c. A good user interface should not require its users to memorise commands.
    d. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface.

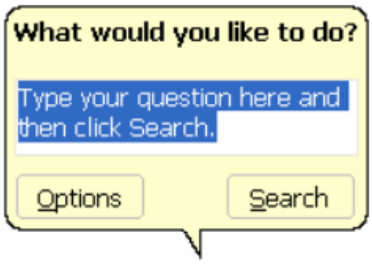
Besides, the following three issues are crucial to enhance the **Speed of learning**:

- **U s e of metaphors1 and intuitive command name**: Speed of learning an interface is greatly facilitated if these are based on some dayto-day real-life examples or some physical objects with which the users are familiar with.
    - (Ex: shopping cart).
- **Consistency**: Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions.
- **Component-based interface**: Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with.

2. **Speed of use**: Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is some times referred to as productivity support of the interface
3. **Speed of recall**: Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximised.
4. **Error prevention**: A good user interface should minimise the scope of committing errors while initiating different commands.
5. **Aesthetic and attractive**: A good user interface should be attractive to use. An attractive user interface catches user attention and fancy.
6. **Consistency**: The commands supported by a user interface should be consistent.
7. **Feedback**: A good user interface must provide feedback to various user actions.
8. **Support for multiple skill levels**: A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users.
9. **Error recovery (undo facility):** While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface.
10. **User guidance and on-line help**: Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software.

User Guidance and On-line Help

- ✓ Users may seek help about the operation of the software any time while using the software.
- ✓ This is provided by the on-line help system.



On-line help system: Users expect the on-line help messages to be tailored to the context in which they invoke the "help system". Therefore, a good online help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way.

Guidance messages: The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc.

Error messages: Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc.

Mode-based Vs Mode-less Interface

A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the software. Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software. On the other hand, in a mode-based interface, different set of commands can be invoked depending on the mode in which the system is, i.e. the mode at any instant is determined by the sequence of commands already issued by the user.

A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

Fig 9.2 shows the interface of a word processing program. The top-level menu provides the user with a gamut of operations like file open, close, save, etc. When the user chooses the open option, another frame is popped up which limits the user to select a name from one of the folders.

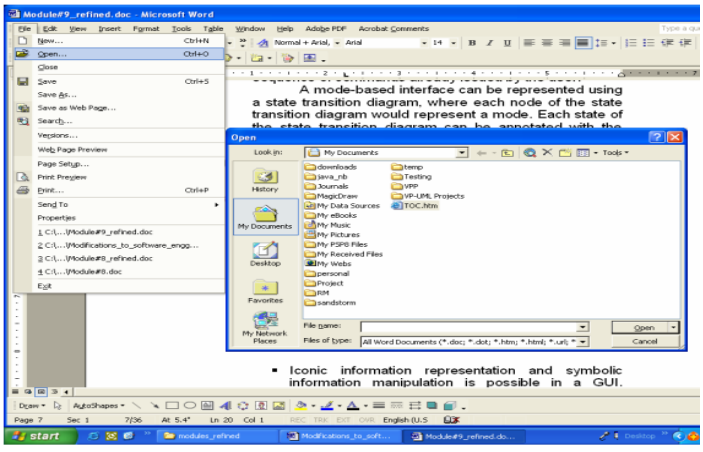


Fig 9.2. An example of mode-based interface

Types of user interfaces

Broadly speaking, user interfaces can be classified into the following three categories:

1. Command language-based interfaces
2. Menu-based interfaces
3. Direct manipulation interfaces

1.Command Language-based Interface:

- As the name itself suggests, is based on designing a command language which the user can use to issue the commands.
- The user is expected to frame the appropriate commands in the language and type them appropriately whenever required.
- Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.
- Further, a command language-based interface can be implemented even on cheap alphanumeric terminals.
- Usually, command language-based interfaces are difficult to learn and require the user to memorise the set of primitive commands

Issues in designing a command language-based interface:

Two overbearing command design issues are to:

- Reduce the number of primitive commands that a user has to remember and
- Minimize the total typing required.

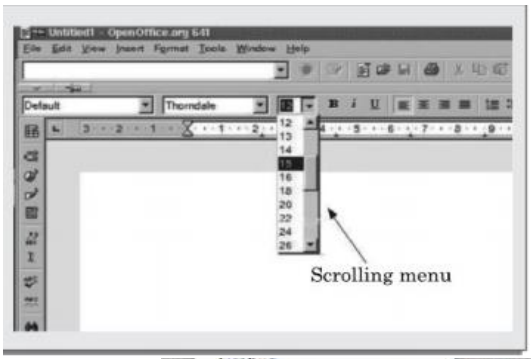
2. Menu-based Interface

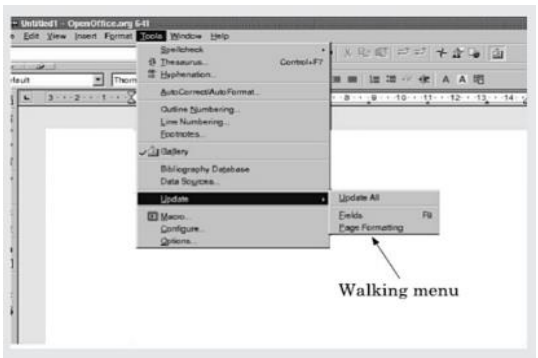
- An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands.
- A menu-based interface is based on recognition of the command names, rather than recollection.
- menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device

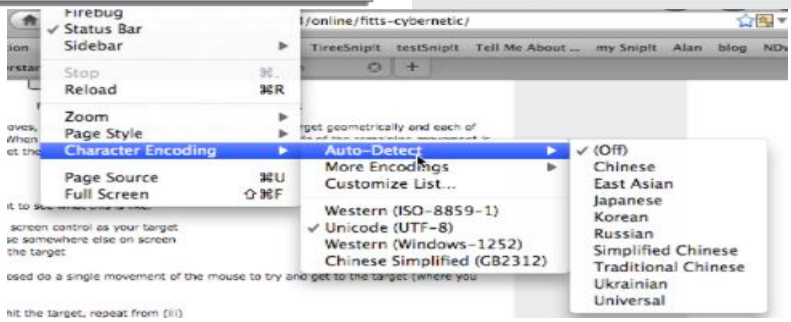
Scrolling menu: Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required.

Walking menu:. In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu.

Hierarchical menu: This type of menu is suitable for small screens with limited display area such as that in mobile phones. In a hierarchical menu, the menu items are organised in a hierarchy or tree structure







3. Direct Manipulation Interfaces:

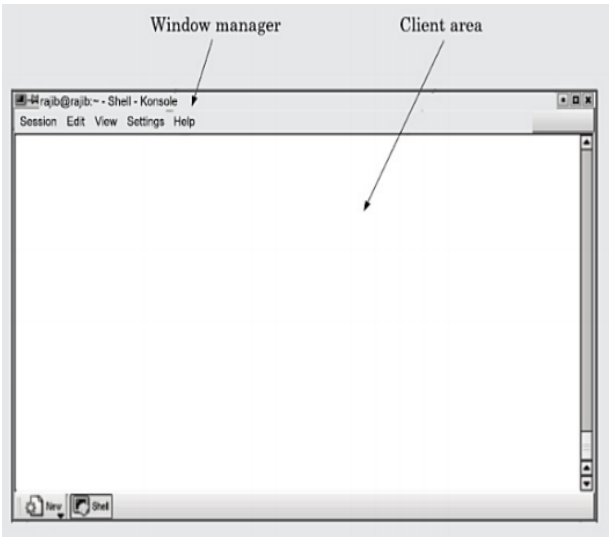
- Direct manipulation interfaces present the interface to the user in the form of visual models (i.e., icons2 or objects).
- For this reason, direct manipulation interfaces are sometimes called as iconic interfaces.
- In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g., pull an icon representing a file into an icon representing a trash box, for deleting the file.

Component-based GUI development

The current style of user interface development is component-based. It recognises that every user interface can easily be built from a handfuls of predefined components such as menus, dialog boxes, forms, etc.

Window System:

Most modern graphical user interfaces are developed using some window system. A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g., one window can be used for editing a program and another for drawing pictures, etc

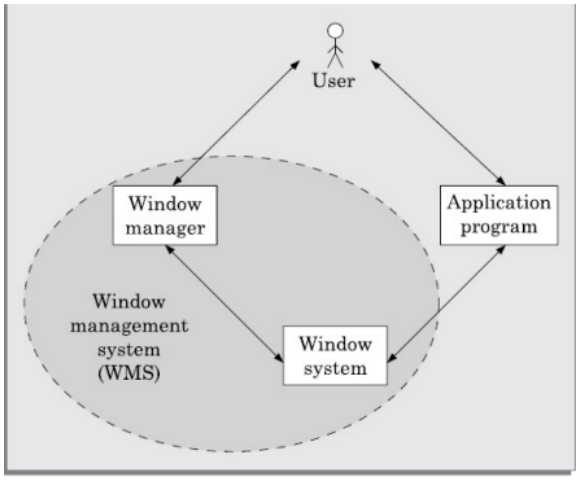


Window management system (WMS)

A graphical user interface typically consists of a large number of windows. Therefore, it is necessary to have some systematic way to manage these windows. Most graphical user interface development environments do this through a window management system (WMS).

A WMS consists of two parts (see Figure 9.4):

- A window manager, and
- A window system.



Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc.

The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the **window system**. The application programmer can also directly invoke the services of the window system to develop the user interface.

Component-based development

A development style based on **widgets** (A widget is the short form of a window object) is called component-based (or widget-based) GUI development style.

Types of Widgets

1. **Label widget**:. A label widget does nothing except to display a label, i.e., it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.
2. **Container widget**: These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget.
3. **Pop-up menu**: These are transient and task specific. A pop-up menu appears upon pressing the mouse button, irrespective of the mouse position.

4. **Pull-down menu** : These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.
5. **Dialog boxes**: We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user.
6. **Push button**: A push button contains key words or pictures that describe the action that is triggered when you activate the button.

7. **Radio buttons**: A set of radio buttons are used when only one option has to be selected out of many options.
8. **Combo boxes**: A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from.

A USER INTERFACE DESIGN METHODOLOG

A GUI Design Methodology

Interface design methodology consists of the following important steps:

- Examine the use case model of the software.
- Task and object modelling.
- Metaphor selection.
- Interaction design and rough layout.
- Detailed presentation and graphics design.
- GUI construction and Usability evaluation

Use case model: The Use-case model is defined as a model which is used to show how users interact with the system in order to solve a problem. As such, the use case model defines the user's objective, the interactions between the system and the user, and the system's behavior required to meet these objectives.

Task and object modeling: A task is a human activity intended to achieve some goals. Examples of task goals can be as follows:

- Reserve an airline seat
- Buy an item
- Transfer money from one account to another
- Book a cargo for transmission to an address

Metaphor selection: The first place one should look for while trying to identify the candidate metaphors is the set of parallels to objects, tasks, and terminologies of the use cases.
Ex: White board, Shopping cart,, Post box etc.

Interaction design and rough layout : The interaction design involves mapping the subtasks into appropriate controls, and other widgets such as forms, text box, etc. This involves making a choice from a set of available components that would best suit the subtask. Rough layout concerns how the controls, an other widgets to be organised in windows.

Detailed presentation and graphics design Each window should represent either an object or many objects that have a clear relationship to each other

GUI construction and Usability evaluation: Based on the usability and context proper GUI construction is important.

Ex: Some of the windows have to be defined as modal dialogs. When a window is a modal dialog, no other windows in the application are accessible until the current window is closed. When a modal dialog is closed, the user is returned to the window from which the modal dialog was invoked.