Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.
Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.
Software Design Levels
Software design yields three levels of results:

Architectural Design - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.

High-level Design- The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

Detailed Design- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Modularization
Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.
Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

Smaller components are easier to maintain

Program can be divided based on functional aspects

Desired level of abstraction ca n be brought in the program

Components with high cohesion can be re-used again.

Concurrent execution can be made possible

Desired from security aspect

Concurrency
Back in time, all softwares were meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.
In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.
It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.
Example
The spell check feature in word processor is a module of software, which runs alongside the word processor itself.
Coupling and Cohesion
When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.
Cohesion
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 –

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.

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

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

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

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

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.

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 can also be reused.

Coupling

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 -

Content coupling - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.

Common coupling- When multiple modules have read and write access to some global data, it is called common or global coupling.

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.

Stamp coupling- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

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.

Ideally, no coupling is considered to be the best.

SOFTWARE DESIGN STRATEGIES

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Let us study them briefly:

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Let us study them briefly:

Structured Design

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on ‘divide and conquer’ strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.
These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -
Cohesion - grouping of all functionally related elements.
Coupling - communication between different modules.
A good structured design has ***high*** cohesion and ***low*** coupling arrangements.

Function Oriented Design
In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.
Function oriented design inherits some properties of structured design where divide and conquer methodology is used.
This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.
Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process
The whole system is seen as how data flows in the system by means of data flow diagram.

DFD depicts how functions change the data and state of entire system.

The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.

Each function is then described at large.

Object Oriented Design
Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.
Let us see the important concepts of Object Oriented Design:
Objects - All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.

Classes - A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

Encapsulation - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.

Inheritance - OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

Polymorphism - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Software Design Approaches

There are two generic approaches for software designing:

Top down Design

We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their one set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-

system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.
Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.
Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Bottom-up Design

The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.
Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

SOFTWARE ANALYSIS & DESIGN TOOLS

Software analysis and design includes all activities, which help the transformation of requirement specification into implementation. Requirement specifications specify all functional and non-functional expectations from the software. These requirement specifications come in the shape of human readable and understandable documents, to which a computer has nothing to do.
Software analysis and design is the intermediate stage, which helps human-readable requirements to be transformed into actual code.
Let us see few analysis and design tools used by software designers:

Data Flow Diagram

Data flow diagram is a graphical representation of data flow in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.
There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

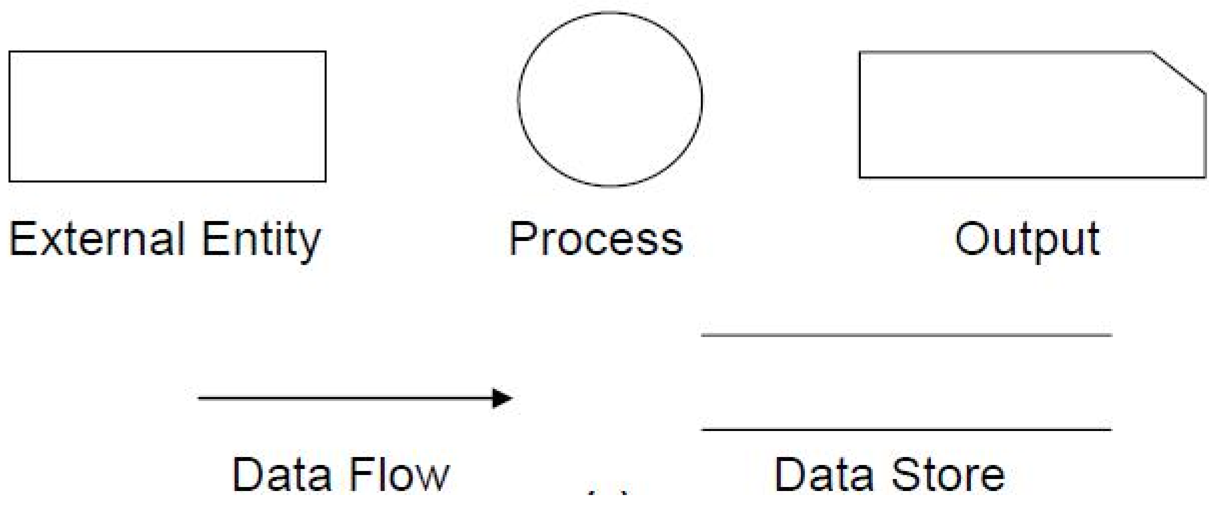
Types of DFD

Data Flow Diagrams are either Logical or Physical.

Logical DFD - This type of DFD concentrates on the system process and flow of data in the system. For example in a Banking software system, how data is moved between different entities.

Physical DFD - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

DFD Components
DFD can represent Source, destination, storage and flow of data using the following set of components -



Entities - Entities are source and destination of information data. Entities are represented by rectangles with their respective names.

Process - Activities and action taken on the data are represented by Circle or Round-edged rectangles.

Data Storage - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.

Data Flow - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

Importance of DFDs in a good software design

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism – it is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub-functions. In fact, any hierarchical model is simple to understand. Human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem, but also for several other applications such as showing the flow of documents or items in an organization.

Importance of Data Dictionary
A data dictionary plays a very important role in any software development process because of the following reasons:
• A data dictionary provides a standard terminology for all relevant data for use by the engineers working in a project. A consistent vocabulary for data items is very important, since in large projects different engineers of the project have a tendency to use different terms to refer to the same data, which unnecessary causes confusion.
• The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

Context Diagram
The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble. This bubble is labeled according to the main function of the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are also represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data names. The name 'context diagram' is well justified because it represents the context in which the system is to exist, i.e. the external entities who would interact with the system and the specific data items they would be supplying the system and the data items they would be receiving from the system. The context diagram is also called as the level 0 DFD.
To develop the context diagram of the system, it is required to analyze the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving the system. Here, the term "users of the system" also includes the external systems which supply data to or receive data from the system.
The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is in contrast with the bubbles in all other levels which are annotated with verbs. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality.

Commonly made errors while constructing a DFD model
Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is powerful thing, it is an expensive pedagogical technique in the business world. It is therefore helpful to understand the different types of mistakes that users usually make while constructing the DFD model of systems.

Many beginners commit the mistake of drawing more than one bubble in the context diagram. A context diagram should depict the system as a single bubble.

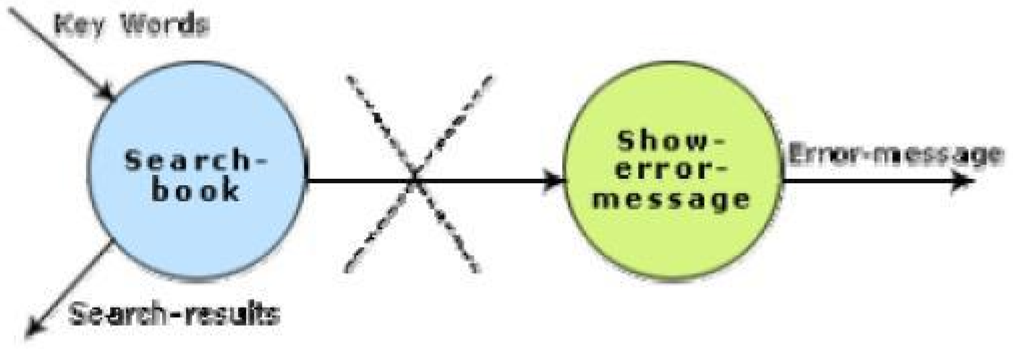
Many beginners have external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear at other levels of the DFD.

It is a common oversight to have either too less or too many bubbles in a DFD. Only 3 to 7 bubbles per diagram should be allowed, i.e. each bubble should be decomposed to between 3 and 7 bubbles.

Many beginners leave different levels of DFD unbalanced.

A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD. It is important to realize that a DFD is the data flow representation of a system, and it does not represent control information.

For an example mistake of this kind:
Consider the following example. A book can be searched in the library catalog by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalog, then an error message
is generated. While generating the DFD model for this simple problem, many beginners commit the mistake of drawing an arrow (as shown in fig. 10.10) to indicate the error function is invoked after the search book. But, this is control information and should not be shown on the DFD



Showing control information on a DFD - incorrect

Another error is trying to represent when or in what order different functions (processes) are invoked and not representing the conditions under which different functions are invoked.

If a bubble A invokes either the bubble B or the bubble C depending upon some conditions, we need only to represent the data that flows between bubbles A and B or bubbles A and C and not the conditions depending on which the two modules are invoked.

A data store should be connected only to bubbles through data arrows. A data store cannot be connected to another data store or to an external entity.

All the functionalities of the system must be captured by the DFD model. No function of the system specified in its SRS document should be overlooked.

Only those functions of the system specified in the SRS document should be represented, i.e. the designer should not assume functionality of the system not specified by the SRS document and then try to represent them in the DFD.

Improper or unsatisfactory data dictionary.

The data and function names must be intuitive. Some students and even practicing engineers use symbolic data names such a, b, c, etc. Such names hinder understanding the DFD model.

Shortcomings of a DFD model

DFD models suffer from several shortcomings. The important shortcomings of the DFD models are the following:

DFDs leave ample scope to be imprecise - In the DFD model, the function performed by a bubble is judged from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find-book-position has only intuitive meaning and does not specify several things, e.g. what happens when some input information are missing or are incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.

Control aspects are not defined by a DFD- For instance; the order in which inputs are consumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modeling real-time systems.

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. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one.

STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation) into a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart.

• Transform analysis
• Transaction analysis

Normally, one starts with the level 1 DFD, transforms it into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD. These are discussed in the subsequent sub-sections.

Structure Chart

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that

are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.
The basic building blocks which are used to design structure charts are the following:

Rectangular boxes: Represents a module.

Module invocation arrows: Control is passed from on one module to another module in the direction of the connecting arrow.

Data flow arrows: Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.

Library modules: Represented by a rectangle with double edges.

Selection: Represented by a diamond symbol.

Repetition: Represented by a loop around the control flow arrow.

Structure Chart vs. Flow Chart
We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

• It is usually difficult to identify the different modules of the software from its flow chart representation.
• Data interchange among different modules is not represented in a flow chart.
• Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart

Transform Analysis
Transform analysis identifies the primary functional components (modules) and the high level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:
• Input
• Logical processing
• Output

OBJECT MODELLING USING UML

Model

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following**:**

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model used for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed, along with the model.

Unified Modeling Language (UML)

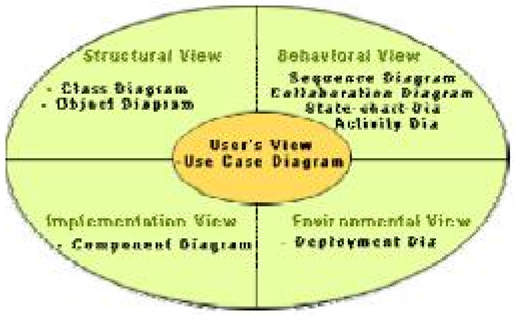
UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology

UML Diagrams

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system**:**

- User’s view
- Structural view
- Behavioral view
- Implementation view
- Environmental view



Different types of diagrams and views supported in UML

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.
Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.
Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.
Implementation view: This view captures the important components of the system and their dependencies.
Environmental view: This view models how the different components are implemented on different pieces of hardware.

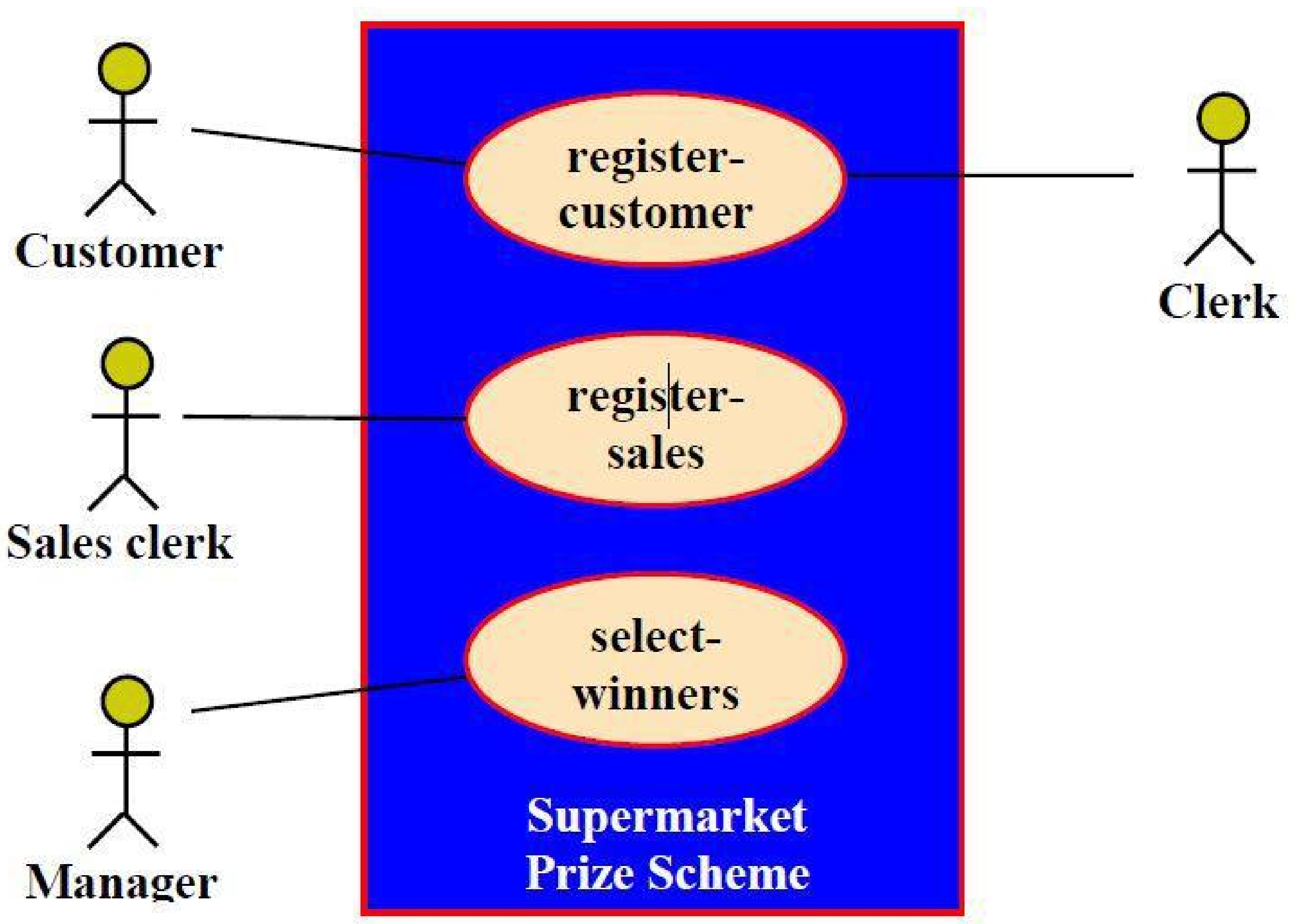
USE CASE DIAGRAM
Use Case Model
The use case model for any system consists of a set of "use cases". Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: "What the users can do using the system?" Thus for the Library Information System (LIS), the use cases could be**:**
• issue-book
• query-book
• return-book
• create-member
• add-book, etc

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user's point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.
Purpose of use cases
The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence. The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction. For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount. Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline 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 through the use case.

Representation of Use Cases
Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (such as Library Information System) appears inside the rectangle.

Text Description
Each ellipse on the use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc. The behavior description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.
Contact persons: This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting, etc.
Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.
Post-condition: This captures the state of the system after the use case has successfully completed.
Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.
Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.
Sample dialogs: These serve as examples illustrating the use case.
Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.
Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation
Example 2:
A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the checkout staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs.10,000. The entries against the CN are the reset on the day of every year after the prize winners' lists are generated.
The use case model for the Supermarket Prize Scheme is shown As discussed earlier, the use cases correspond to the high-level functional requirements. From the problem description, we can identify three use cases: "register-customer", "register-sales", and "select-winners". As a sample, the text description for the use case "register-customer" is shown.



Use case model for Supermarket Prize Scheme

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. Customer: enter the necessary values.

4. System: display the generated id and the message that the customer has been successfully registered.

Scenario 2: at step 4 of mainline sequence

1. System: displays the message that the customer has already registered.

Scenario 2: at step 4 of mainline sequence

1**.** System: displays the message that some input information has not been entered. The system displays a prompt to enter the missing value.

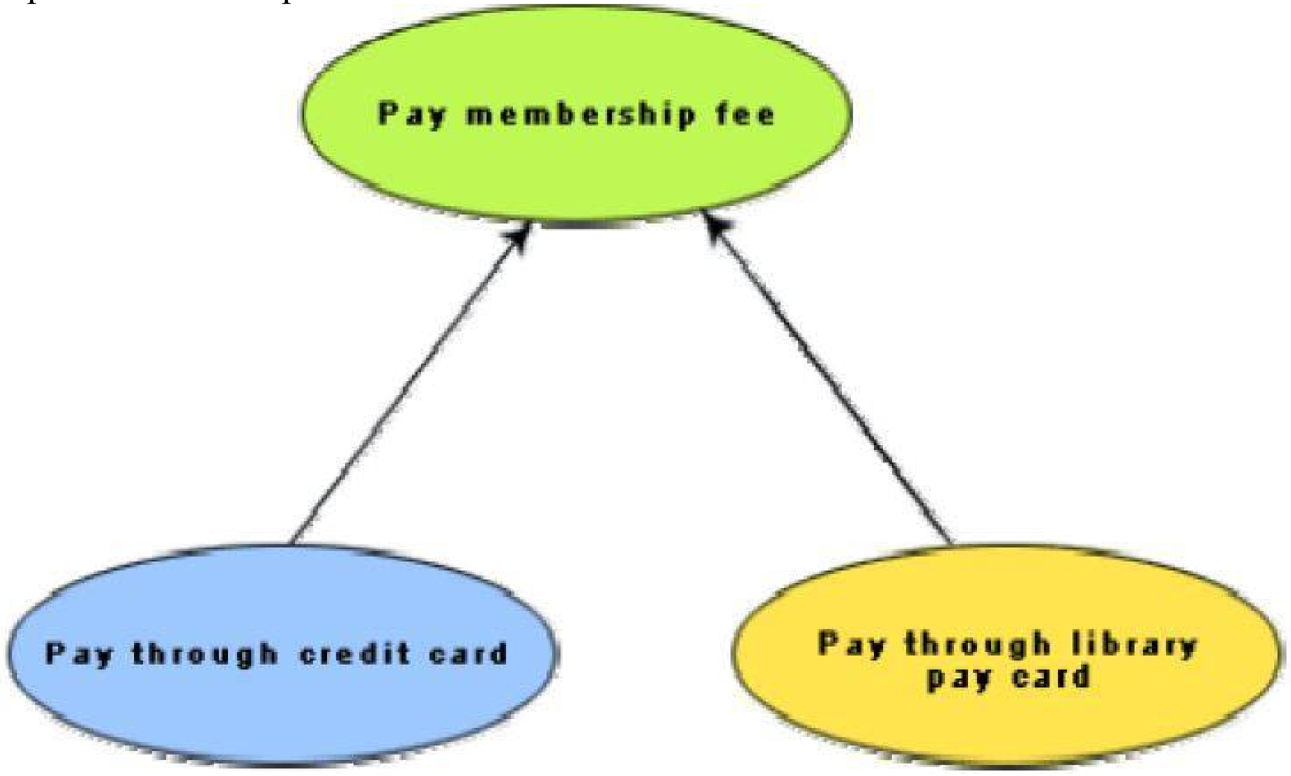
The description for other use cases is written in a similar fashion.

Factoring of use cases

It is often desirable to factor use cases into component use cases. Actually, factoring of use cases are required under two situations. First, complex use cases need to be factored into simpler use cases. This would not only make the behavior associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single sized (A4) paper. Secondly, use cases need to be factored whenever there is common behavior across different use cases. Factoring would make it possible to define such behavior only once and reuse it whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the sake of it

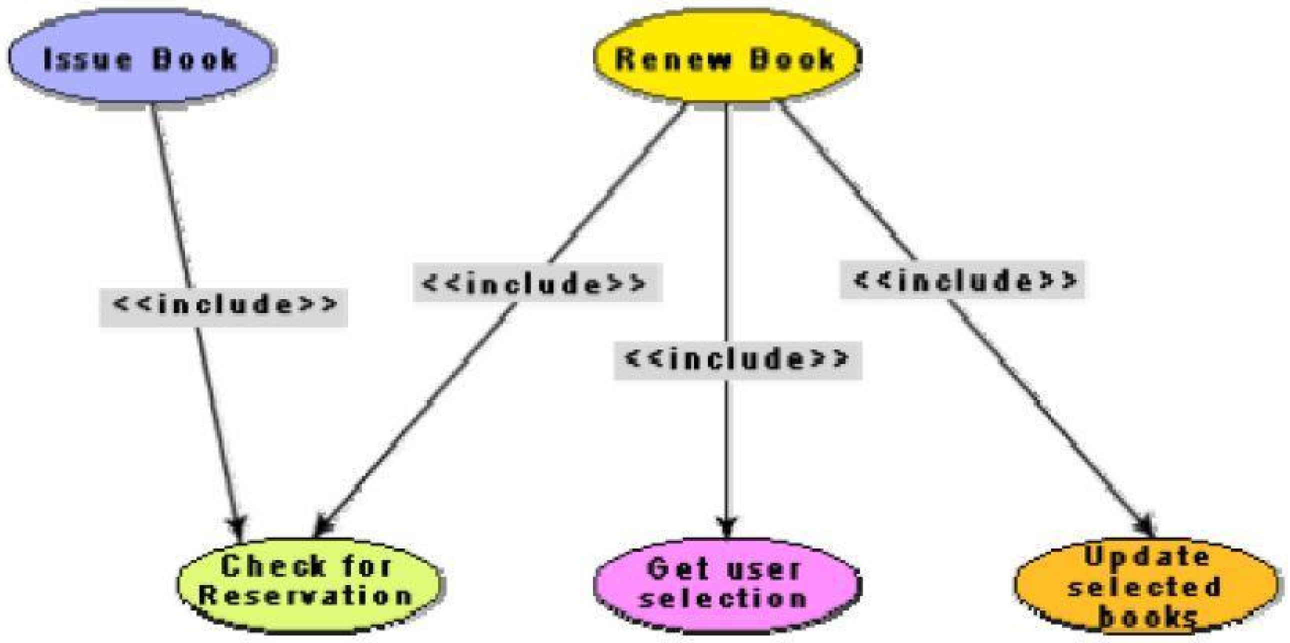
UML offers three mechanisms for factoring of use cases as follows:
1. **Generalization**

Use case generalization can be used when one use case that is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behavior and meaning of the parent use case. The notation is the same too (as shown in fig. 13.3). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.



Includes

The *includes* relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The *includes* relationship involves one use case including the behavior of another use case in its sequence of events and actions



Example use case inclusion

Extends
The main idea behind the *extends* relationship among the use cases is that it allows you to show optional system behavior. An optional system behavior is extended only under certain conditions.