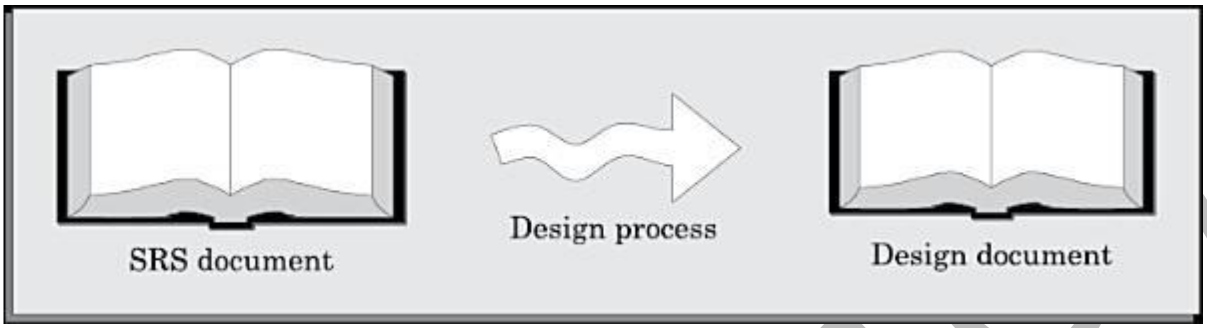
MODULE--3

Software Design ---

During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. We can state the main objectives of the design phase the design process starts using the SRS document and completes with the production of the design document. The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.



OVERVIEW OF THE DESIGN PROCESS The following items are designed and documented during the design phase.

Different modules required: The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs. For example, in academic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named handle student registration.

Control relationships among modules: A control relationship between two modules essentially arises due to function calls across the two modules. The control relationships existing among various modules should be identified in the design document.

Interfaces among different modules: The interfaces between two modules identify the exact data items that are exchanged between the two modules when one module invokes a function of the other module.

Data structures of the individual modules: Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for storing and managing the data of a module need to be properly designed and documented.

Algorithms required implementing the individual modules: Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

Classification of Design Activities A good software design is seldom realized by using a single step procedure, rather it requires iterating over a series of steps called the design activities. Let us first classify the design activities before discussing them in detail. Depending on the order in which various design activities are performed, we can broadly classify them into two important stages.

• Preliminary (or high-level) design, and

• Detailed design.

- Through 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. High-level design is a crucial step in the overall design of software. When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among them, and are arranged in a hierarchy.
- Many different types of notations have been used to represent a high-level design. A notation that is widely being used for procedural development is a tree-like diagram called the structure chart.
- During detailed design each module is examined carefully to design its data structures and the algorithms.

Classification of Design Methodologies The design activities vary considerably based on the specific design methodology being used. A large number of software design methodologies are available. We can roughly classify these methodologies into procedural and object-oriented approaches. These two approaches are two fundamentally different design paradigms.
Analysis versus design The analysis results are generic and do not consider implementation or the issues associated with specific platforms. The analysis model is usually documented using some graphical formalism. In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart.

HOW TO CHARACTERISE A 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" may be an important issue to characterize a good solution for embedded software development—since embedded
- applications are often required to work under severely limited memory sizes due to cost, space, or power consumption considerations.
- judge a design solution depend on the exact application being designed, but to make the matter worse, there is no general agreement among software engineers and researchers on the exact criteria to use for judging a design even for a specific category of application.

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: A Major Concern

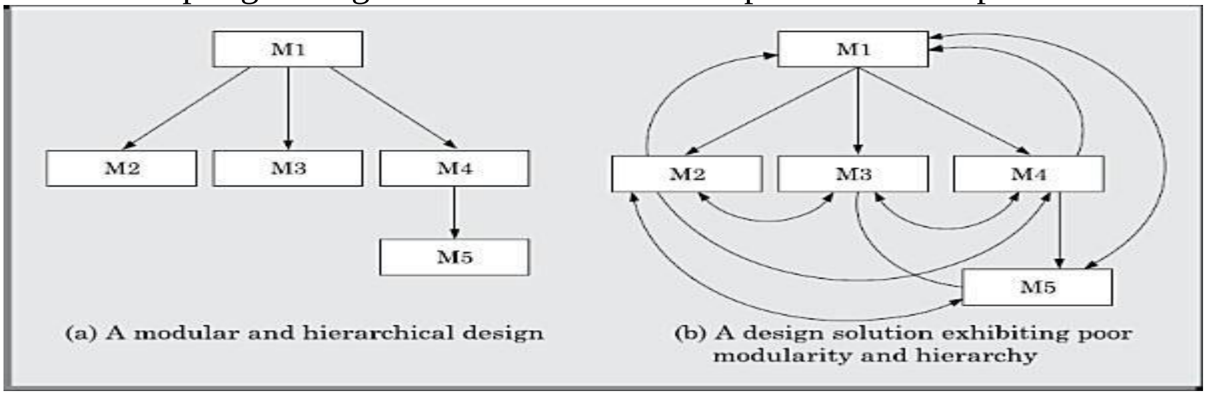
Given that we are choosing from only correct design solutions, understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design. a good design solution should be simple and easily understandable. A design that is easy to understand is also easy to develop and maintain. A complex design would lead to severely increased life cycle costs. Unless a design is easily understandable, it would require tremendous effort to implement, test, debug, and maintain it. We had already pointed out that about 60 per cent of the total effort in the life cycle of a typical product is spent on maintenance. If the software is not easy to understand, not only would it lead to increased development costs, the effort required to maintain the product would also increase manifold

An understandable design is modular and layered--

How can the understandability of two different designs be compared, so that we can pick the better one? To be able to compare the understandability of two design solutions, we should at least have an understanding of the general features that an easily understandable design should possess. A design solution should have the following characteristics to be easily understandable: It should assign consistent and meaningful names to various design components. It should make use of the principles of decomposition and abstraction in good measures to simplify the design. A design solution is understandable, if it is modular and the modules are arranged in distinct layers. A design solution should be modular and layered to be understandable. We now elaborate the concepts of modularity and layering of modules:

Modularity

A modular design is an effective decomposition of a problem. It is a It is a basic characteristic of any good design solution. 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. Decomposition of a problem into modules facilitates taking advantage of the divide and conquers principle. If different modules have either no interactions or little interactions with each other, then each module can be understood separately. This reduces the perceived complexity of the design solution greatly. To understand why this is so, remember that it may be very difficult to break a bunch of sticks which have been tied together, but very easy to break the sticks individually. A software design with high cohesion and low coupling among modules is the effective problem decomposition


M1
M2
M3
M4
M5
(a) A modular and hierarchical design
M1
M2
M3
M4
M5
(b) A design solution exhibiting poor modularity and hierarchy

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. The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done. A layered design can be considered to be implementing control abstraction, since a module at a lower layer is unaware of (about how to call) the higher layer modules.
- A layered design can make the design solution easily understandable, since to understand the working of a module, one would at best have to understand how the immediately lower layer modules work without having to worry about the functioning of the upper layer modules.
- When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error. This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error.

COHESION AND COUPLING Effective problem decomposition is an important characteristic of a good design. Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other.

- Cohesion is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.
- Cohesion is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

Coupling: Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled. If the interactions occur through some shared data, then also we say that they are highly coupled.

Cohesion: When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion. If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor cohesion.

Functional independence

A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules.

Error isolation: Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules. Further, once a failure is detected, error isolation makes it very easy to locate the error. On the other hand, when a module is not functionally independent, once a failure is detected in a functionality provided by the module, the error can be potentially in any of the large number of modules and propagated to the functioning of the module.

Scope of reuse: Reuse of a module for the development of other applications becomes easier. The reasons for this are as follows. A functionally independent module performs some well-defined and precise task and the interfaces of the module with other modules are very few and simple.

Understandability: When modules are functionally independent, complexity of the design is greatly reduced. This is because of the fact that different modules can be understood in isolation, since the modules are independent of each other.

Classification of Cohesiveness

Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective.

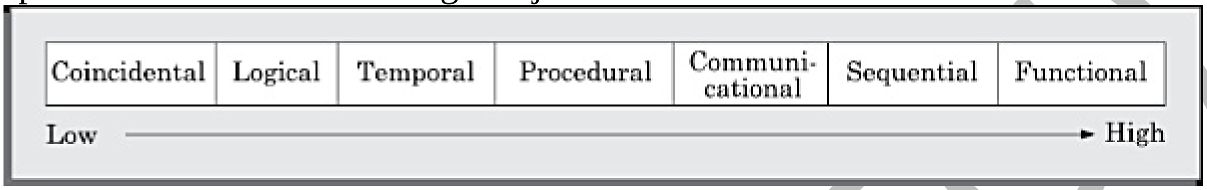


Figure -- Classification of cohesion.

Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely An example of a module with coincidental cohesion has been shown in Figure (a).Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc.

Temporal cohesion: When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion. As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data. Consider the activities associated with order processing in a trading house. The functions login(), place-order(), check-order(), print bill(), place-order-on-vendor(), update-inventory(), and logout() all do different thing and operate on different data.

Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure. As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admit Student, enter Marks, print Grade Sheet, etc. access and manipulate data stored in an array named student Records defined within the module.

Sequential cohesion: A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.

Functional cohesion: A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task. For example all functions in a withdraw module will communicated together to finish the with draw task.

Classification of Coupling--

The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled. The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module. Let us now classify the different types of coupling that can exist between two modules. Between any two interacting modules, any of the following five different types of coupling can exist.

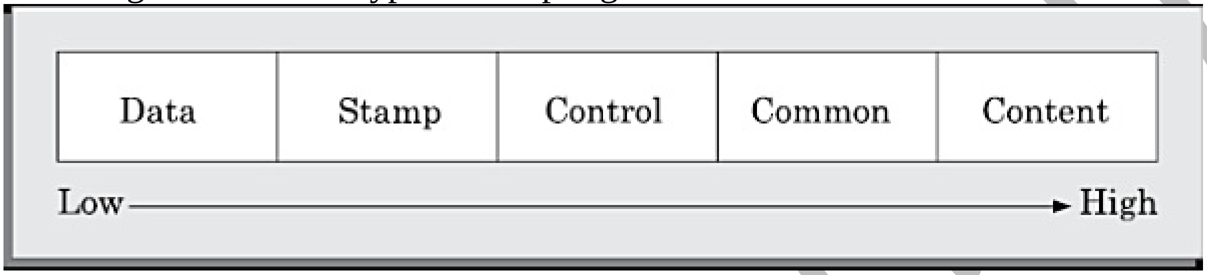


Figure: Classification of coupling.

Data coupling: Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share some global data items.

Content coupling: Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

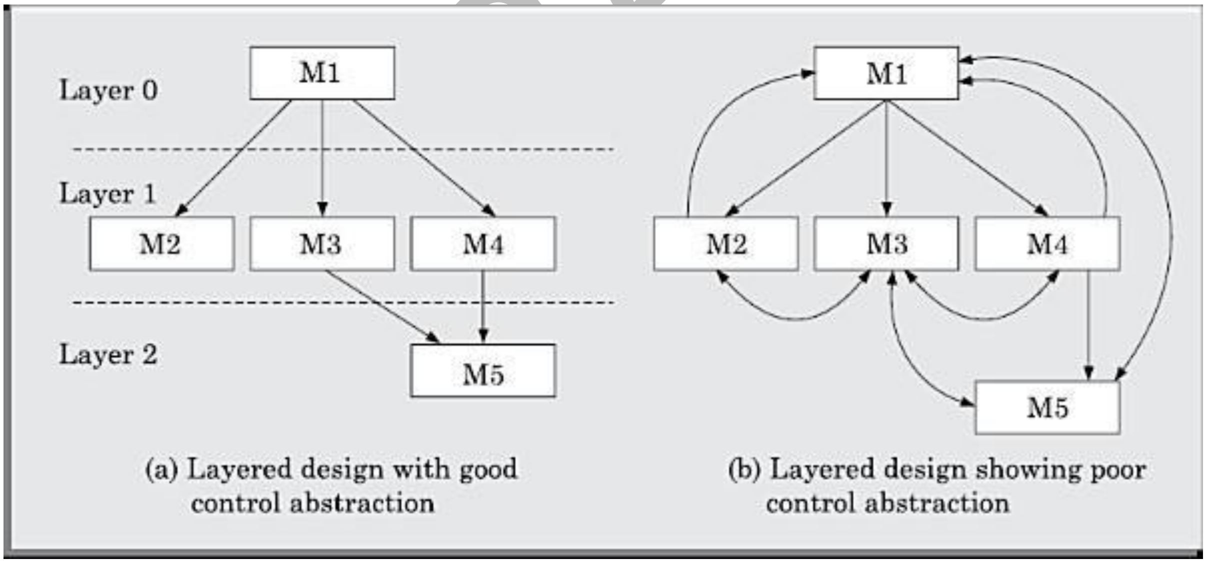
LAYERED ARRANGEMENT OF MODULES --

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 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. 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.

- The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent. Understanding a layered design is easier since to understand one module, one would have to at best consider the modules at the lower layers

Super ordinate and subordinate modules: In a control hierarchy, a module that controls another module is said to be super ordinate 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 (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 (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.
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 (a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.


Layer 0
M1
Layer 1
M2
M3
M4
Layer 2
M5
(a) Layered design with good control abstraction
M1
M2
M3
M4
M5
(b) Layered design showing poor control abstraction
Figure: Examples of good and poor control abstraction.

APPROACHES TO SOFTWARE DESIGN--
There are two fundamentally different approaches to software design that are in use today—function-oriented design, and object-oriented design. Though these two design approaches are radically different, they are complementary rather than competing techniques. The object oriented approach is a relatively newer technology and is still evolving. For development of large programs, the object- oriented approach is becoming increasingly popular due to

certain advantages that it offers. On the other hand, function-oriented designing is a mature technology and has a large following.

Function-oriented Design—

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

Top-down decomposition: A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system. In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions. For 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.

This high-level function may be refined into the following sub functions:

• assign-membership-number
• create-member-record
• print-bill

Each of these sub functions may be split into more detailed sub functions and so on.

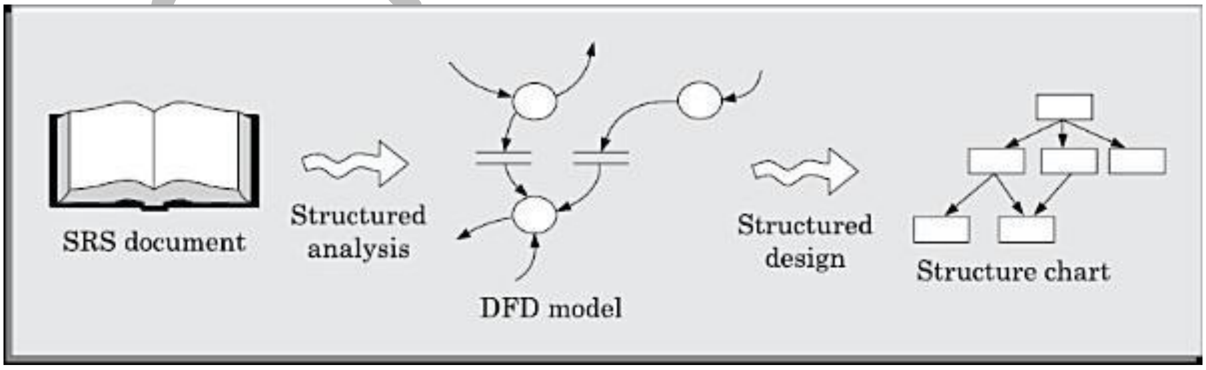
FUNCTION-ORIENTED SOFTWARE DESIGN ---

The term top-down decomposition is often used to denote the successive decomposition of a set of high-level functions into more detailed functions. After top-down decomposition has been carried out, the different identified functions are mapped to modules and a module structure is created. We shall call the design technique discussed in this text a s structured analysis/structured de sign (SA/SD) methodology. The SA/SD technique can b e used to perform the high-level design of software. The details of SA/SD technique are discussed further.

OVERVIEW OF SA/SD METHODOLOGY As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

- Structured analysis (SA)
- Structured design (SD)

During structured analysis, the SRS document is transformed into a data flow diagram (DFD) model. During structured design, the DFD model is transformed into a structure chart.



- ➢ 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. That is, each function that the system needs to perform is analyzed and hierarchically decomposed into more detailed functions.

- On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the high level design or the software architecture for the given problem.
- This is represented using a structure chart. The high-level design stage is normally followed by a detailed design stage. During the detailed design stage, the algorithms and data structures for the individual modules are designed. The detailed design can directly be implemented as a working system using a conventional programming language.

STRUCTURED ANALYSIS—

We have already mentioned that during structured analysis, the major processing tasks (high level functions) of the system are analyzed, and the data flow among these processing tasks is represented graphically.

- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high level function is independently decomposed into detailed functions.
- Graphical representation of the analysis results u s i n g data flow diagrams (DFDs).
- 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.

Note that a 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. 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. DFD is an elegant modeling technique that can be used not only to represent the results of structured analysis of a software problem, but also useful for several other applications such as showing the flow of documents or items in an organization.

Data Flow Diagrams (DFDs)---

The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system. T he 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. A DFD model uses a very limited number of primitive symbols to represent the functions performed by a system and the data flow among these functions. 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.

Primitive symbols used for constructing DFDs There are essentially five different types of symbols used for constructing DFDs.

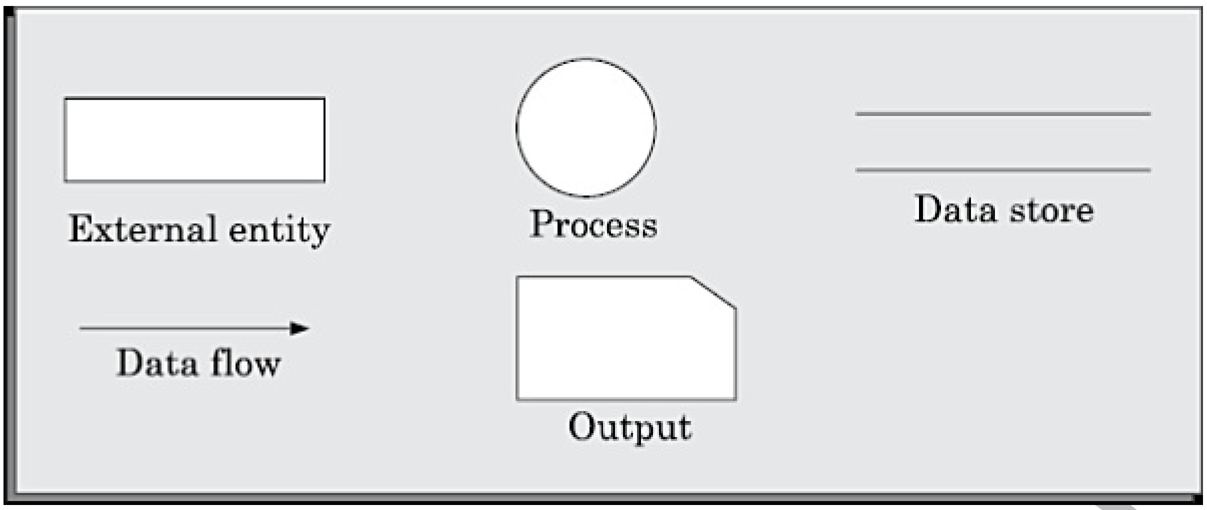


Figure: Symbols used for designing DFDs.

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. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. As an example of a data store, number is a data store

Output symbol: The output symbol is The output symbol is used when a hard copy is produced.

Important concepts associated with constructing DFD models Before we discuss how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs:

Synchronous and asynchronous operations If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed. 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. However, if two bubbles are connected through a data store then the speed of operation of the bubbles is independent.

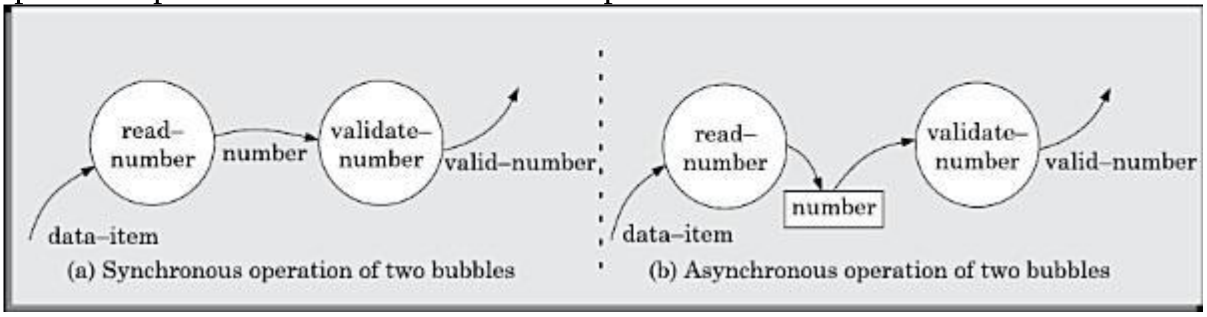


Figure : Synchronous and asynchronous data flow.

Data dictionary ---

Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model. Please remember that the DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1 DFD, level 2 DFDs, etc. A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design. The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa. Such impact analysis is especially useful when one wants to check the impact of changing an input value type, or a bug in some functionality, etc.

Data definition—

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

+: 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.

Example 1: Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3×3 square. A move consists of marking previously unmarked square. The player who first places three consecutive marks along a straight line on the square (i.e. along a row, column, or diagonal) wins the game. As soon as either 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, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

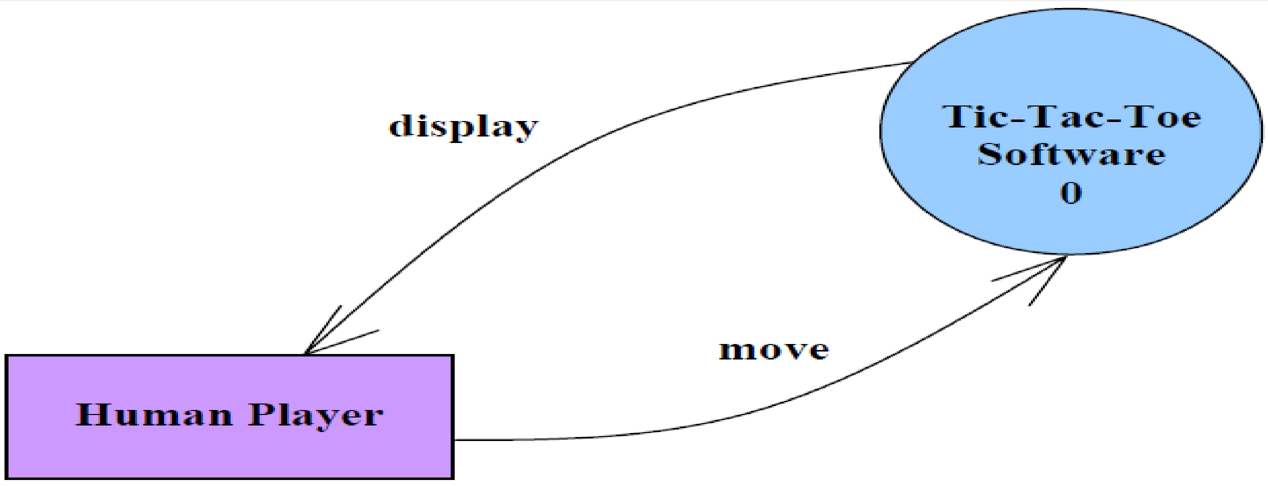


Figure (a) Level 0 for Tic-Tac-Toe game

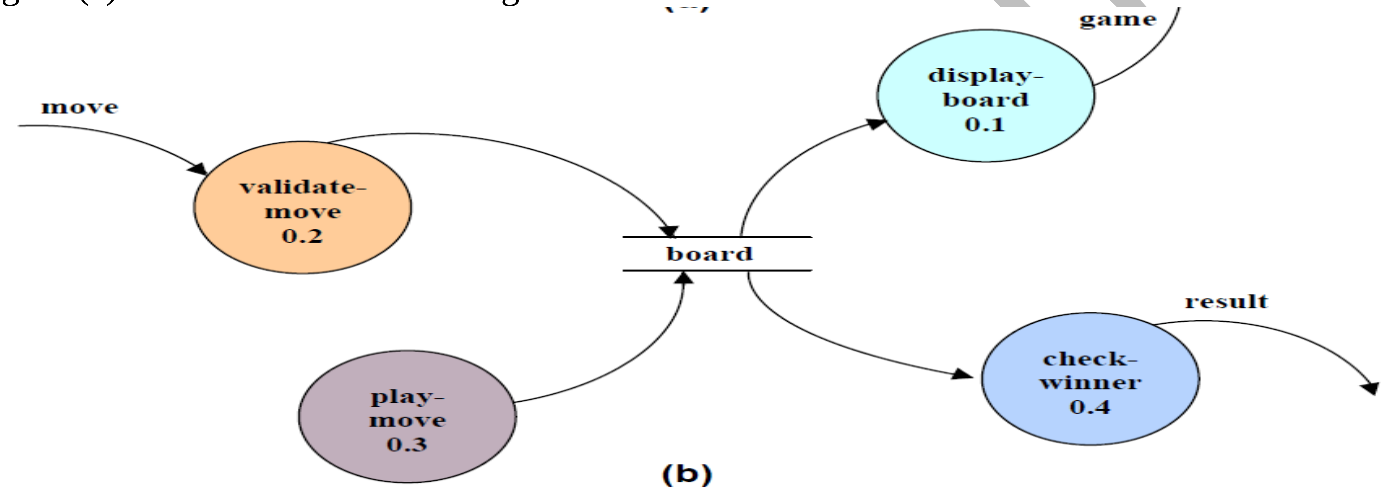


Figure (b) Level 1 DFD for Tic-Tac-Toe game

It may be recalled that the DFD model of a system typically consists of several DFDs: level 0, level 1, etc. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the model. Above figure represents the level 0 and level 1 DFDs for the tic-tac-toe game. The data dictionary for the model is given below.

Data Dictionary for the DFD model in Example 1

move: integer /*number between 1 and 9 */
display: game+result
game: board
board: {integer}9
result: [“computer won”, “human won” “draw”]

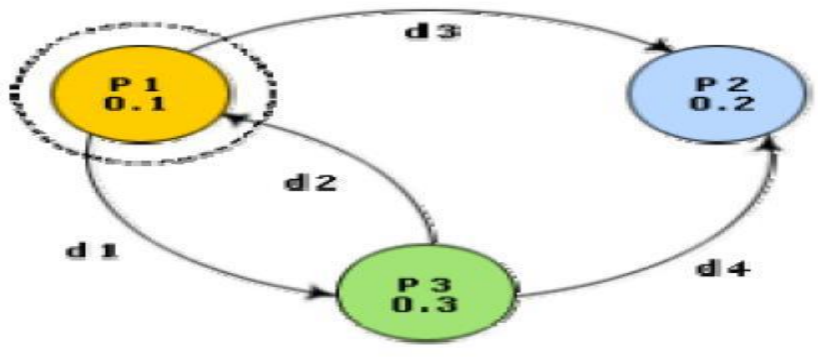
Importance of Data Dictionary

A data dictionary plays a very important role in any software development process because of the following reasons:

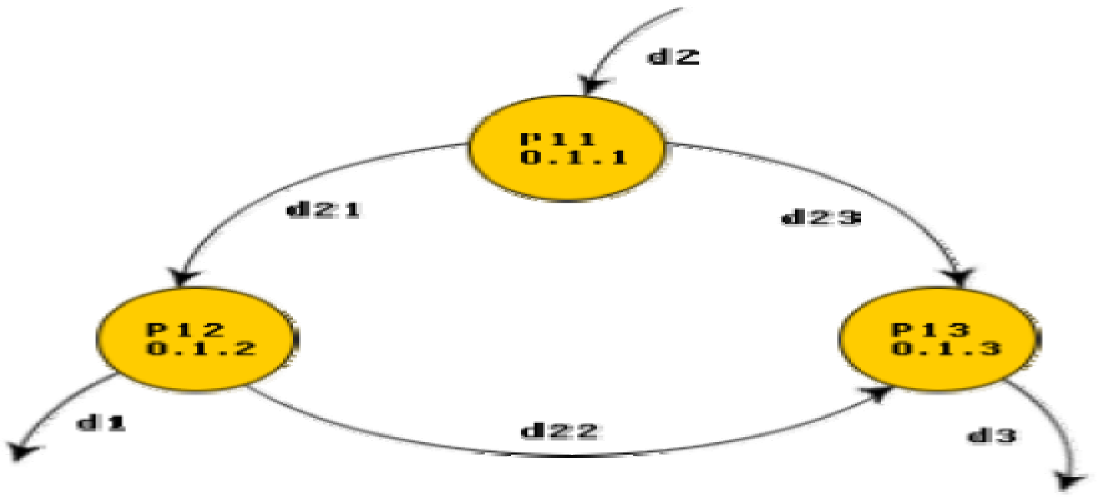
• A data dictionary provides a standard terminology for all relevant data for use by the engineers working in a project. A consistent vocabulary for data items is very important, since in large projects different engineers of the project have a tendency to use different terms to refer to the same data, 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.

Balancing a DFD --
The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD. The concept of balancing a DFD has been illustrated in figure. In the level 1 of the DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1. In the next level, bubble 0.1 is decomposed. The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.



(a) Level 1 DFD



(b) Level 2 DFD

Figure: An example showing balanced decomposition

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.
Example 1: 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 then determine the root mean square (rms) of the three input numbers and display it. In this example, the context diagram (figure) is simple to draw. The system accepts three integers from the user and returns the result to him.

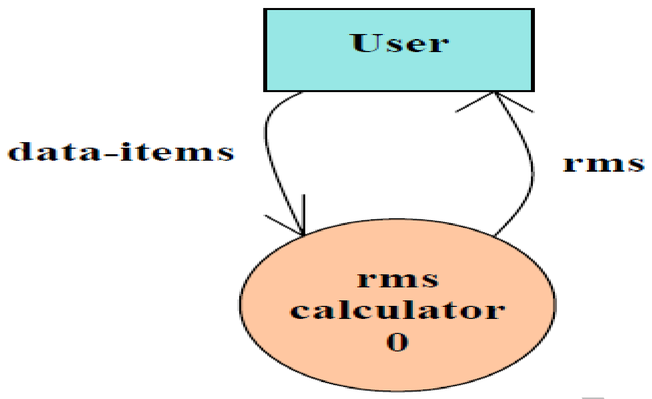


Figure: Context Diagram
To develop the data flow model of a system, first the most abstract representation of the problem is to be worked out. The most abstract representation of the problem is also called the context diagram. After, developing the context diagram, the higher-level DFDs have to be developed.
Context Diagram: - This has been described earlier.
Level 1 DFD: - To develop the level 1 DFD, examine the high-level functional requirements. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the level 1 DFD. We can then examine the input data to these functions and the data output by these functions and represent them appropriately in the diagram.
If a system has more than 7 high-level functional requirements, then some of the related requirements have to be combined and represented in the form of a bubble in the level 1 DFD. Such a bubble can be split in the lower DFD levels. If a system has less than three high-level functional requirements, then some of them need to be split into their sub-functions so that we have roughly about 5 to 7 bubbles on the diagram.
Decomposition:- Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into sub-functions at the successive levels of the DFD. Decomposition of a bubble is also known as factoring or exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything between 3 to 7 bubbles. Too few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes redundant. Also, too many

bubbles, i.e. more than 7 bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

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 by 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, etc. When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Example:-

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 check out 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 context diagram for this problem is shown in figures given below.

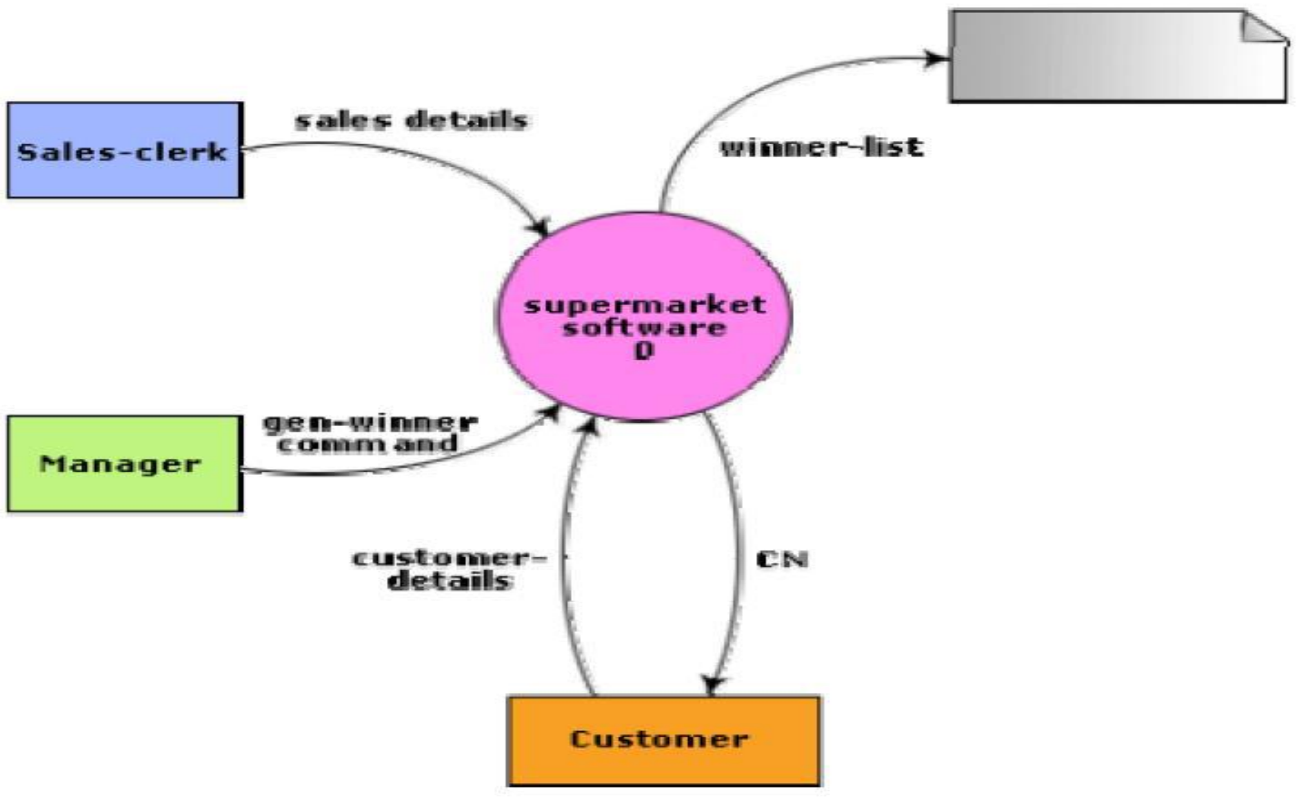


Figure: Context diagram for supermarket problem

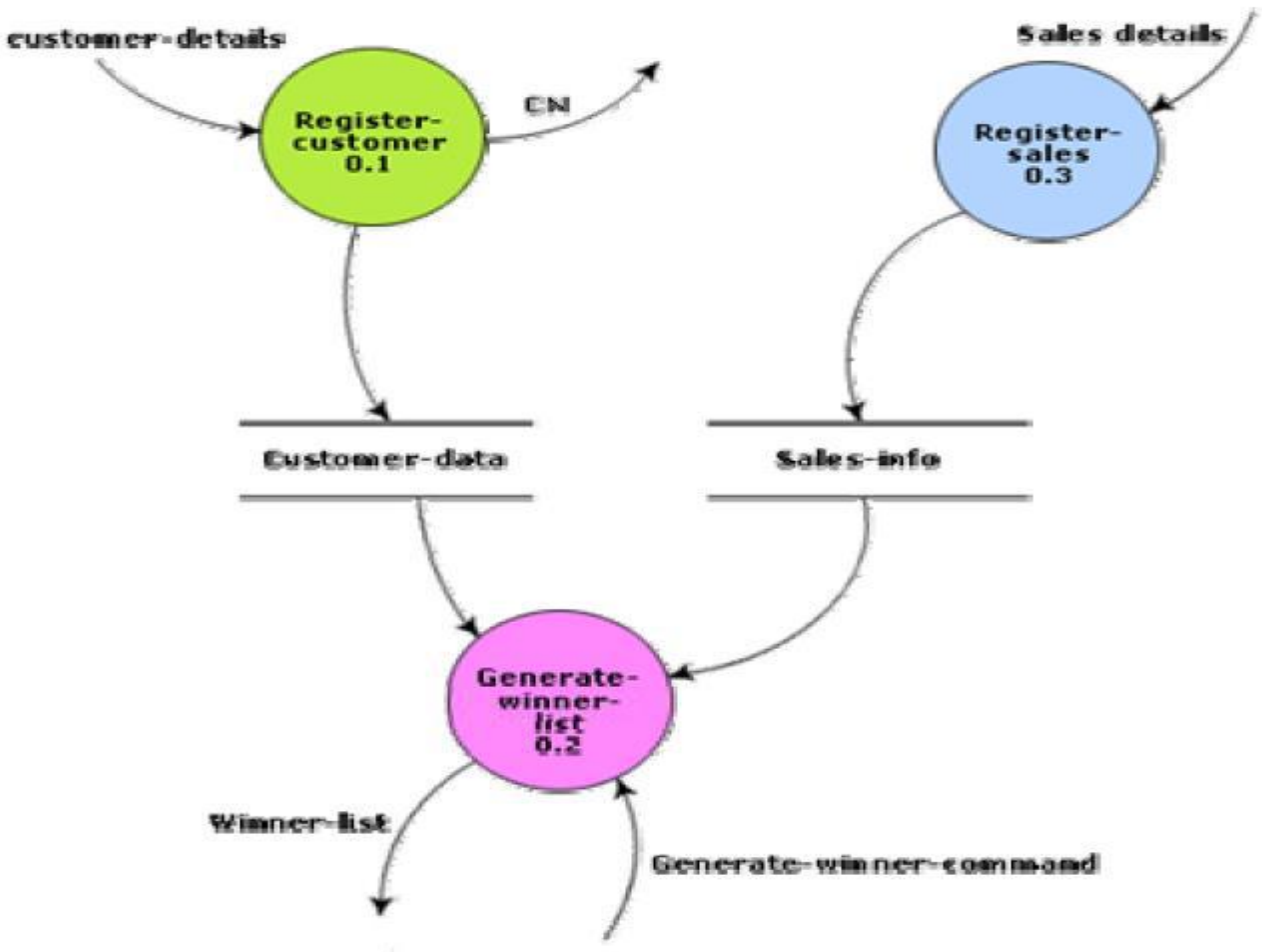


Figure: Level 1 diagram for supermarket problem

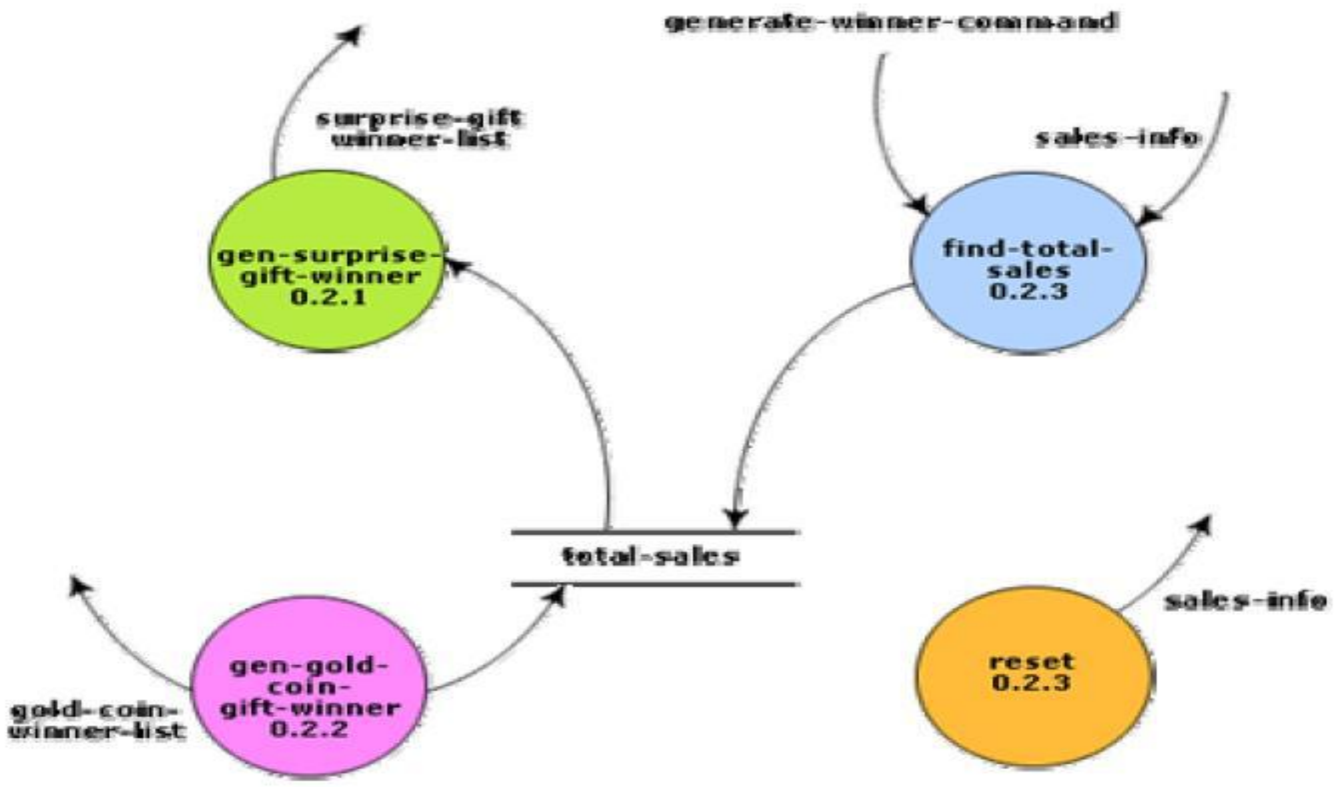


Figure: Level 2 diagram for supermarket problem

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.
- The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its sub-functions and we have to use subjective judgment to carry out decomposition.

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

The input portion of the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical forms (e.g. internal tables, lists, etc.). Each input portion is called an afferent branch.

The output portion of a DFD transforms output data from logical to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called the central transform.

In the next step of transform analysis, the structure chart is derived by drawing one functional component for the ***central transform***, and the ***afferent*** and ***efferent*** branches.

These are drawn below a root module, which would invoke these modules. Identifying the highest level input and output transforms requires experience and skill. One possible approach is to trace the inputs until a bubble is found whose output cannot be deduced from its inputs alone. Processes which validate input or add information to them are not central transforms. Processes which sort input or filter data from it are. The first level structure chart is produced by representing each input and output unit as boxes and each central transform as a single box. In the third step of transform analysis, the structure chart is refined by adding sub-functions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialization and termination process, identifying customer modules, etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Example: Structure chart for the RMS software

For this example, the context diagram was drawn earlier.

To draw the level 1 DFD (figure), from a cursory analysis of the problem description, we can see that there are four basic functions that the system needs to perform – accept the input numbers from the user, validate the numbers, calculate the root mean square of the input numbers and, then display the result.

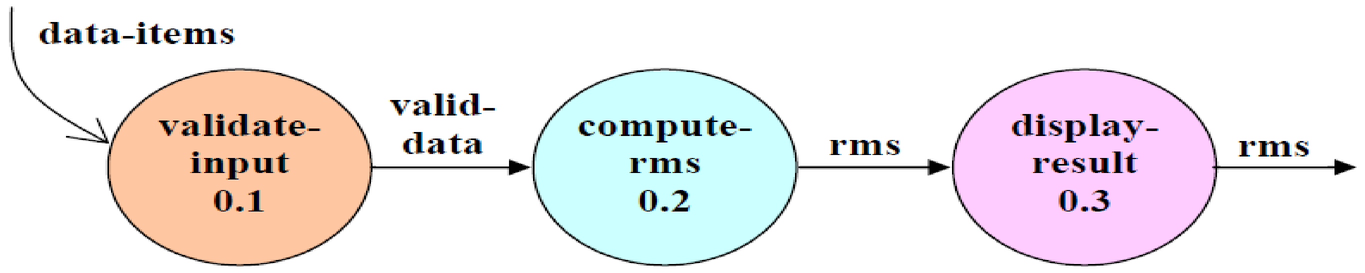


Figure: Level 1 DFD

By observing the level 1 DFD, we identify the validate-input as the afferent branch and write-output as the efferent branch. The remaining portion (i.e. compute-rms) forms the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in figure below.

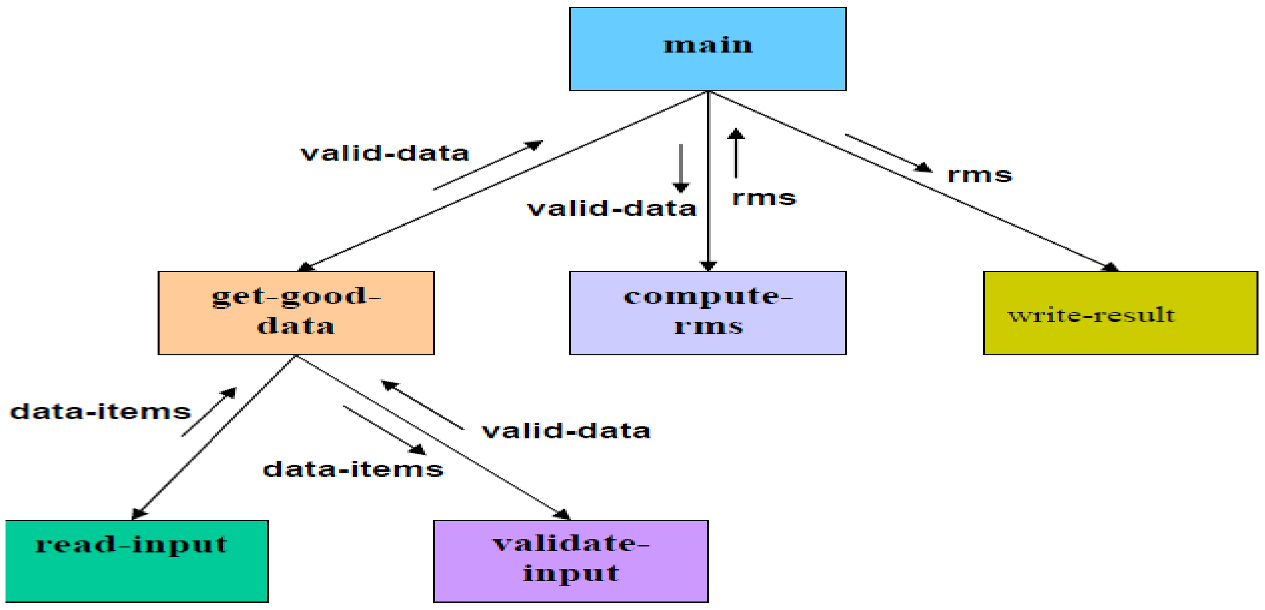


Figure: Structure Chart

Transaction Analysis

A transaction allows the user to perform some meaningful piece of work. Transaction analysis is useful while designing transaction processing programs. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. This is in contrast to a transform centered system which is characterized by similar processing steps for each data item. Each different way in which input data is handled is a transaction. A simple way to identify a transaction is to check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transaction may not require any input data. These transactions can be identified from the experience of solving a large number of examples.

For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction a module. Every transaction carries a tag, which identifies its type.Transaction analysis uses this tag to divide the system into transaction modules and a transaction-center module.

The structure chart for the supermarket prize scheme software is shown in figure.

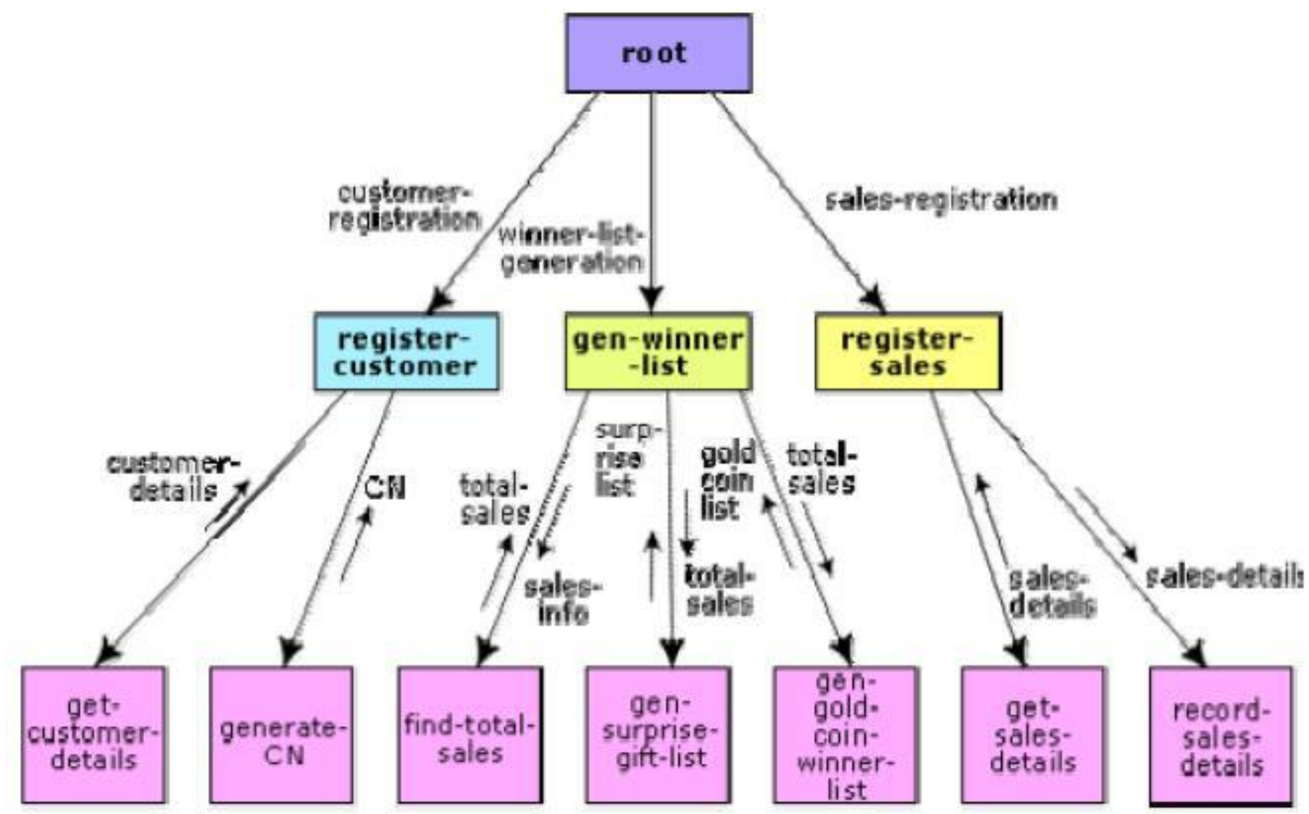


Figure: Structure Chart for the supermarket prize scheme

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.
- The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower level modules.
- The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out.
- To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

DESIGN REVIEW—

After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing, and maintenance perspectives, who may or may not be the members of the development team.

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.

Object-oriented Design

In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e. entities). Each object is associated with a set of functions that are called its methods. Each object contains its own data and is responsible for managing it. The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object. The system state is de centralized since there is no globally shared data in the system and data is stored in each object. For example, in library automation software, each library member may be a separate object with its own data and functions to operate on the stored data. The methods defined for one object cannot directly refer to or change the data of other objects.

Objects decompose a system into functionally independent modules. ADT is an important concept that forms an important pillar of object orientation. Let us now discuss the important concepts behind an ADT. There are, in fact, three important concepts associated with an ADT—data abstraction, data structure, data type.

Data abstraction: The principle of data abstraction implies that how data is exactly stored is abstracted away. This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organized, and manipulated inside the object.

Data structure: A data structure is constructed from a collection of primitive data items. Just as a civil engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organized collection of primitive data items such as integer, floating point numbers, characters, etc.

Data type: A type is a programming language terminology that refers to anything that can be instantiated. For example, int, float, char etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types. An ADT-based design displays high cohesion and low coupling. Therefore, object- oriented designs are highly modular. Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

Object and Class Concepts

Objects--

The purpose of class modeling is to describe objects. For example , ***Joe Smith. Simplex company, process number 7648*** and ***the top window*** are objects.

An ***object*** is a concept, abstraction, or thing with identity that has meaning for an application.

Classes--

An object is an ***instance*** of a class. A ***class*** describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics. ***Person, company, process,*** and ***window*** are all classes.

Class Diagrams--

There are two kinds of models of structure—classes diagrams and object diagrams.

Class diagrams provide a graphic notation for modeling classes and their relationships, thereby describing possible objects. Class diagrams are useful both for abstract modeling and for designing actual programs. An ***object diagram*** shows individual objects and their relationships. Figure shows a class (left) and instances (right) described by it.

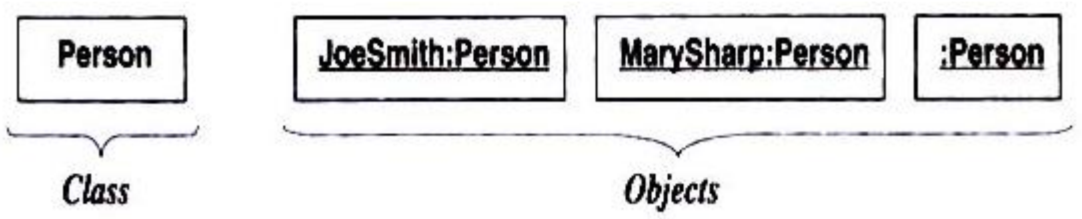


Figure **A class and objects.** Objects and classes are the focus of class modeling.

Values and Attributes--

A ***value*** is a piece of data. An ***attribute*** is a named property of a class that describes a value held by each object of the class. ***Name, birthdate,*** and ***weight*** arc attributes of ***Person*** objects. ***Color, model Year,*** and ***weight*** are attributes of ***Car*** objects. Each attribute has a value for each object.

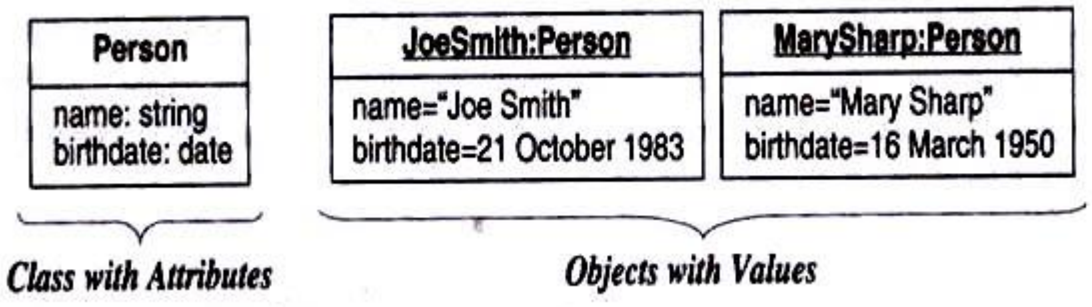


Figure **Attributes and values.** Attributes elaborate classes.

Operations and Methods--

An ***operation*** is a function or procedure that may be applied to or by objects in a class. ***Hire, fire,*** and ***payDividend*** are operations on class ***Company. Open, close, hide,*** and ***redisplay*** are operations on class ***Window.*** A ***method*** is the implementation of an operation for a class. For example, the class ***File*** may have an operation ***print.***

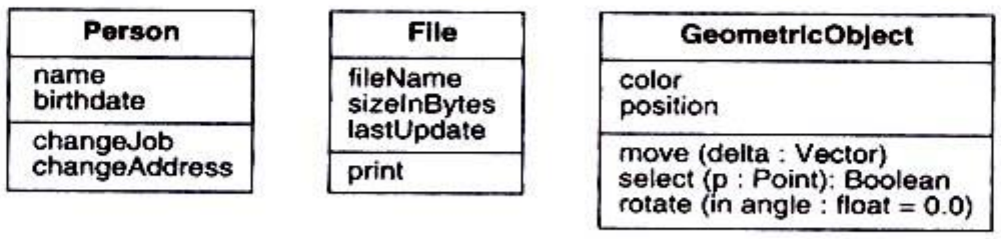
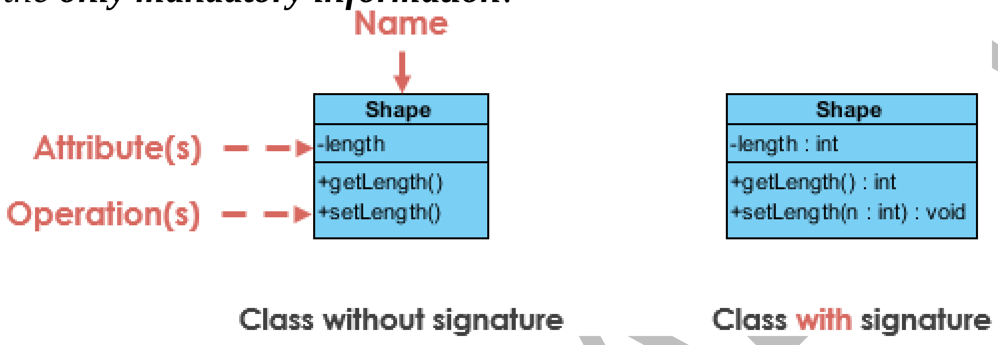


Figure **Operations.** An operation is a function or procedure that may be applied to or by objects in a class.

UML Class Notation--

A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the* ***only mandatory information***.



Class Name:

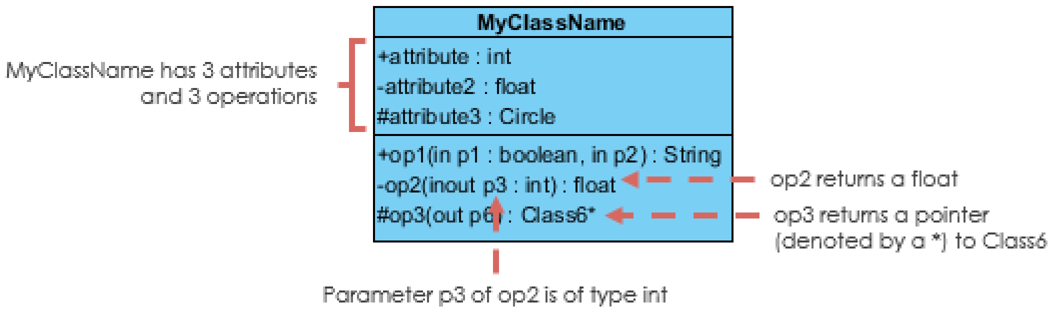
- The name of the class appears in the first partition.

Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

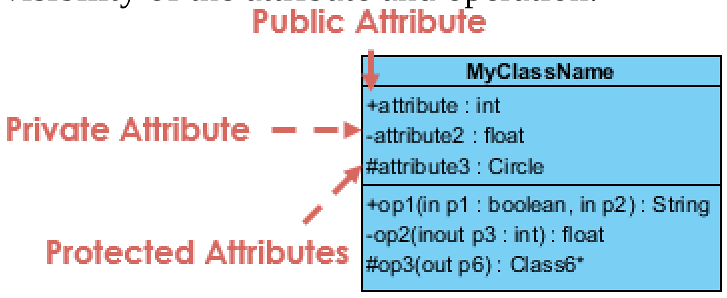
Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code


MyClassName has 3 attributes and 3 operations
MyClassName
+attribute : int
-attribute2 : float
#attribute3 : Circle
+op1(in p1 : boolean, in p2) : String
-op2(inout p3 : int) : float
#op3(out p6) : Class6*
op2 returns a float
op3 returns a pointer (denoted by a *) to Class6
Parameter p3 of op2 is of type int

Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations

<u>**Use Case Diagrams--**</u>

A system involves a set of use cases and a set of actors. Each use case represents a slice of the functionality the system provides. The set of use cases shows the complete functionality of the system at some level of detail. Similarly, each actor represents one kind of object for which the system can perform behavior. The set of actors represents the complete set of objects that the system can serve. Objects accumulate behavior from all the systems with which they interact as actors.

The UML has a graphical notation for summarizing use cases and Figure shows an example. A rectangle contains the use cases for a system with the actors listed on the outside. The name of the system may be written near a side of the rectangle. A name within an ellipse denotes a use case. A "stick man" icon denotes an actor, with the name being placed below or adjacent to the icon. Solid lines connect use cases to participating actors.

In the figure, the actor *Repair technician* participates in two use cases, the others in one each. Multiple actors can participate in a use case, even though the example has only one actor per use case.

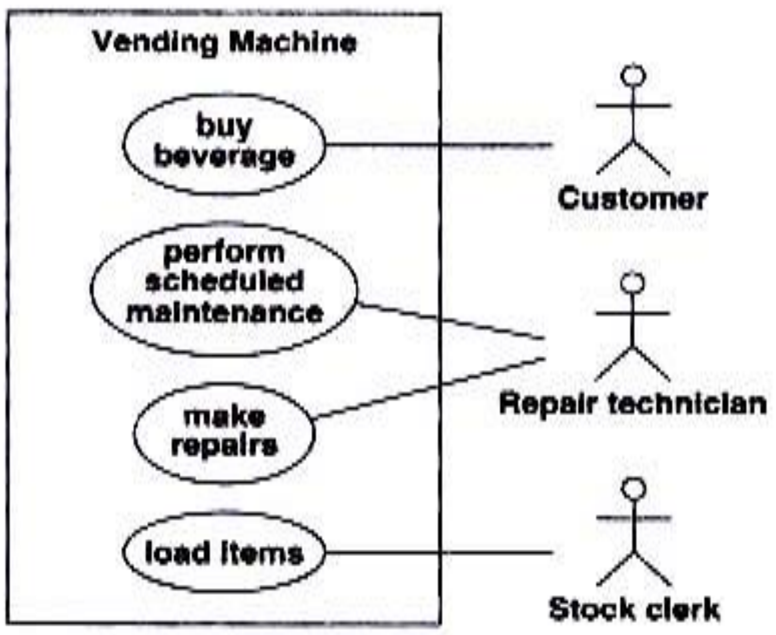


Figure **Use case diagram for a vending machine.** A system involves a set of use cases and a set of actors.

<u>Guidelines for Use Case Models--</u>

■ First determine the system boundary. Ensure that actors are focused. Each actor should have a single, coherent purpose.

■ Each use case must provide value to users. A use case should represent a complete transaction that provides value to users and should not be defined too narrowly.

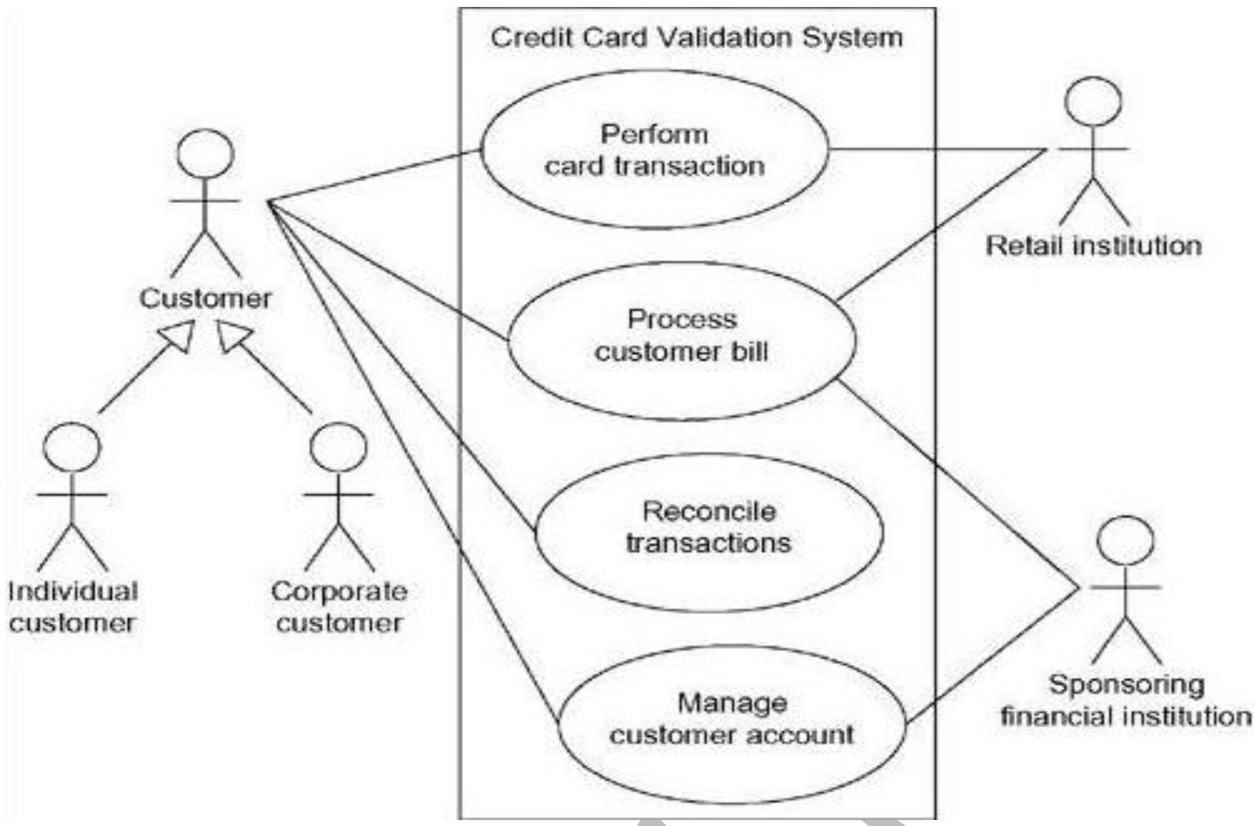
■ Relate use cases and actors. Every use case should have at least one actor, and every actor should participate in at least one use case. A use case may involve several actors, and an actor may participate in several use cases.

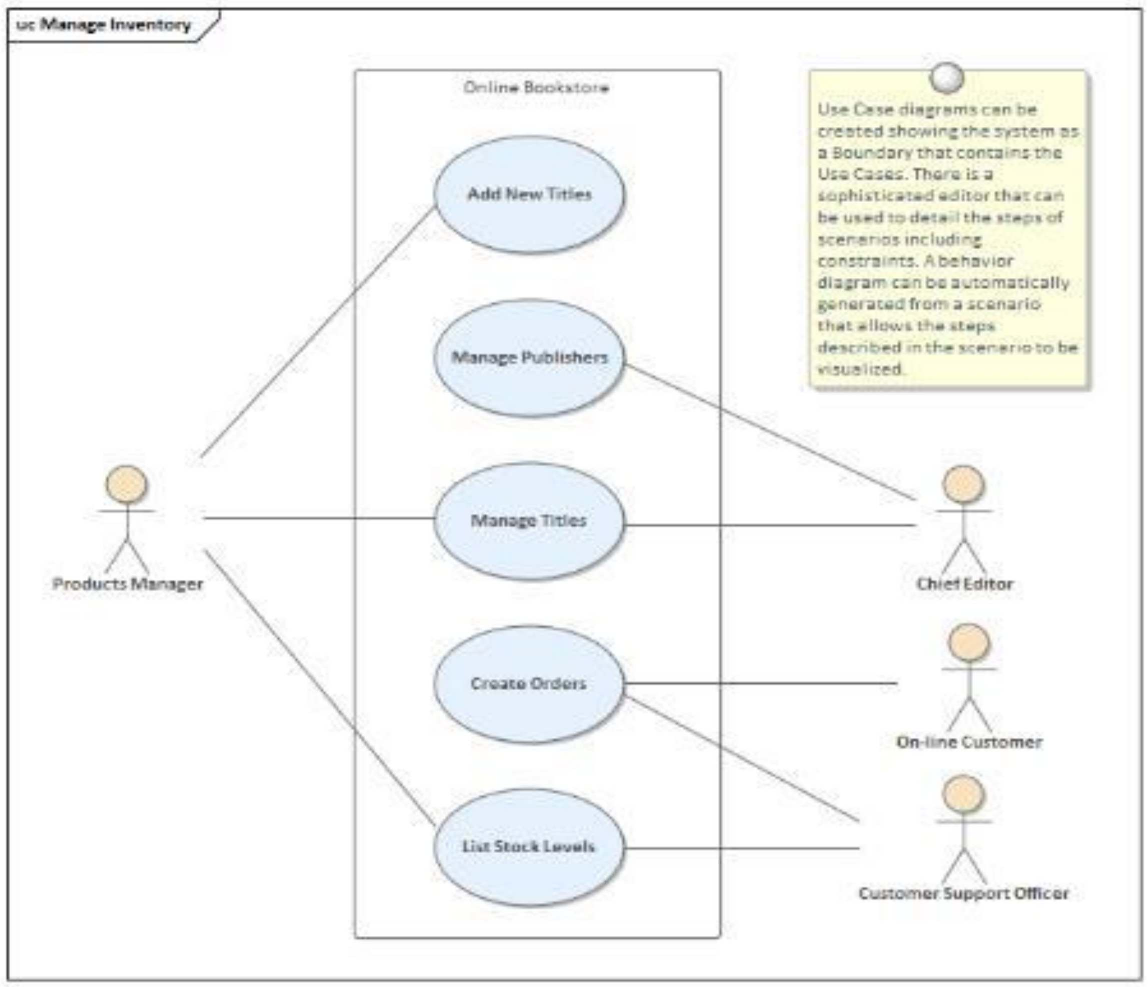
■ Remember that use cases are informal. It is important not to be obsessed by formalism in specifying use cases. They are not intended as a formal mechanism but as a way

to identify and organize system functionality from a user-centered point of view. It is acceptable if use cases are a bit loose at first. Detail can come later as use cases are expanded and mapped into implementations.

■ Use cases can be structured. For many applications, the individual use cases are completely distinct. For large systems, use cases can be built out of smaller fragments using relationships

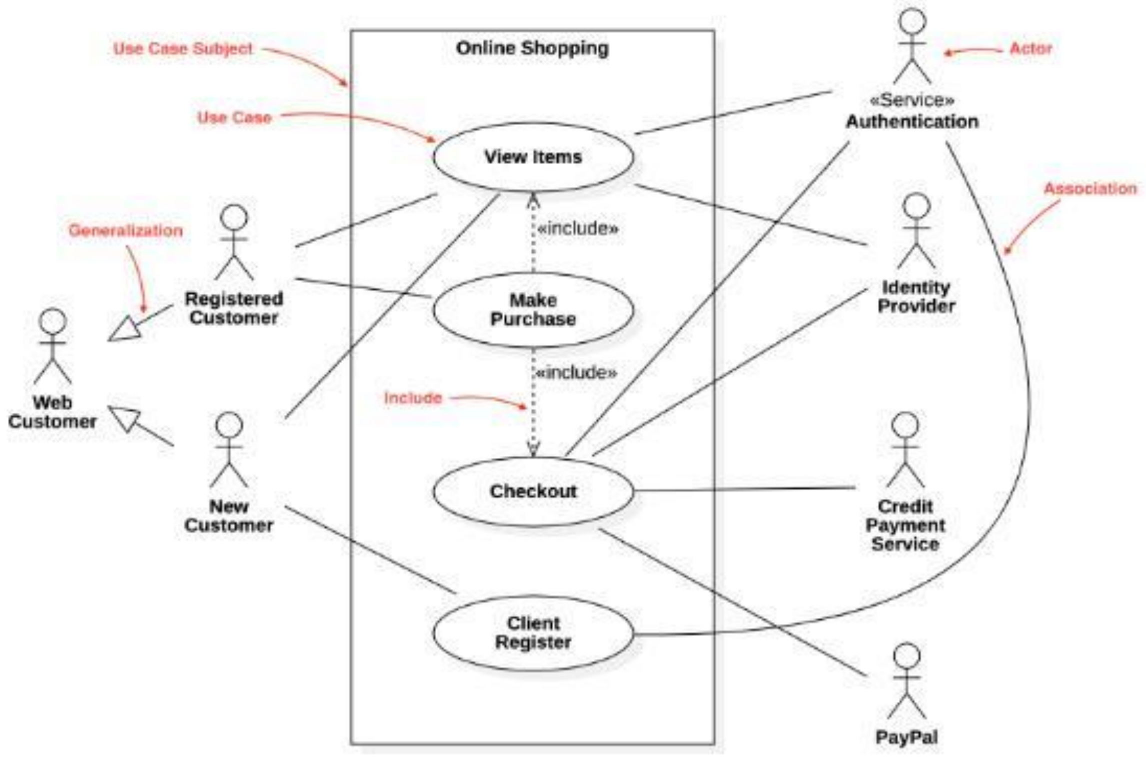
Examples---

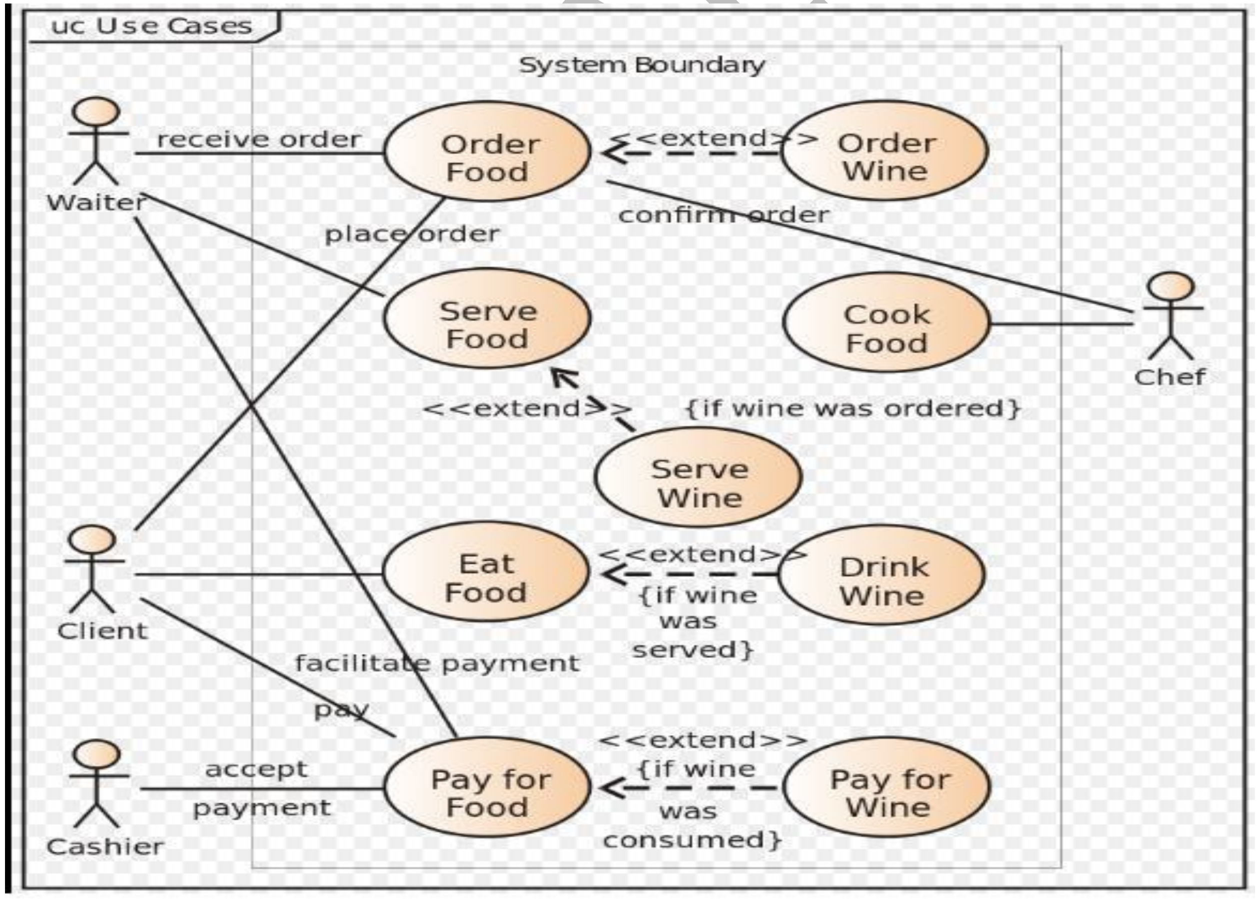




Use Case diagrams can be created showing the system as a Boundary that contains the Use Cases. There is a sophisticated editor that can be used to detail the steps of scenarios including constraints. A behavior diagram can be automatically generated from a scenario that allows the steps described in the scenario to be visualized.

Examples---


Use Case Subject
Online Shopping
Use Case
View Items
«include»
Generalization
Registered Customer
Make Purchase
«include»
Include
Web Customer
New Customer
Checkout
Client Register
«Service» Authentication
Actor
Association
Identity Provider
Credit Payment Service
PayPal


uc Use Cases
System Boundary
Waiter
receive order
Order Food
<<extend>>
Order Wine
place order
confirm order
Serve Food
Cook Food
Chef
<<extend>>
{if wine was ordered}
Serve Wine
Client
Eat Food
<<extend>>
{if wine was served}
Drink Wine
facilitate payment
pay
Cashier
accept payment
Pay for Food
<<extend>>
{if wine was consumed}
Pay for Wine

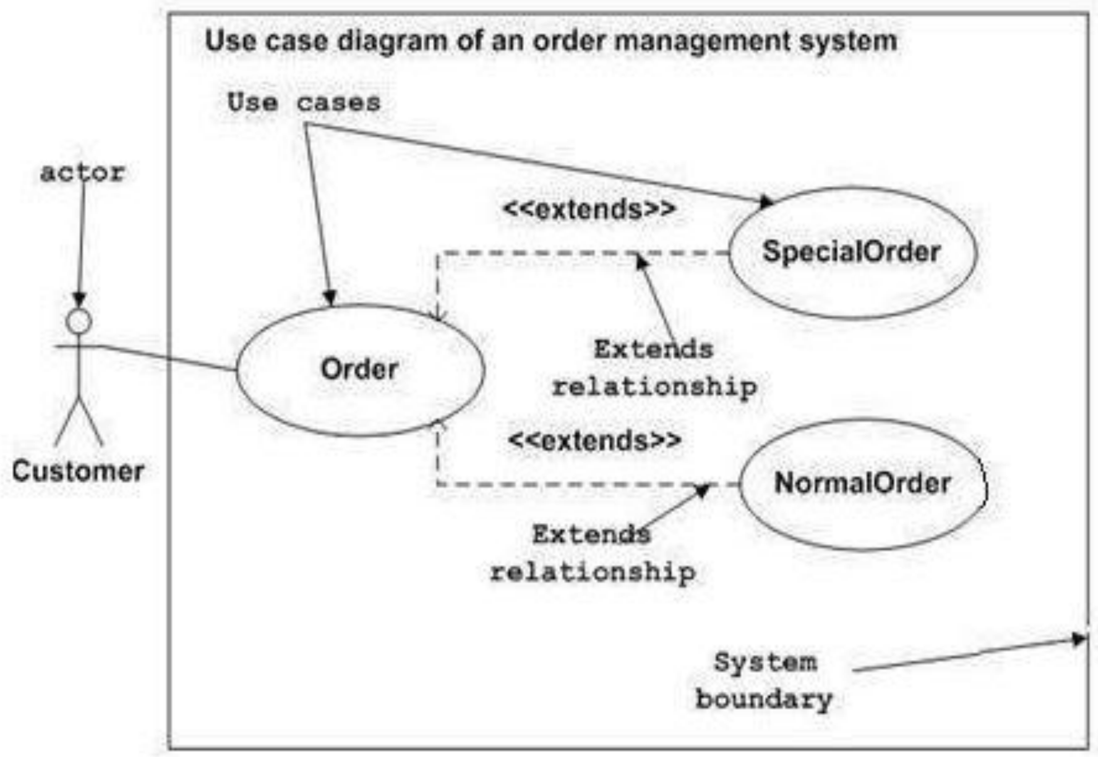
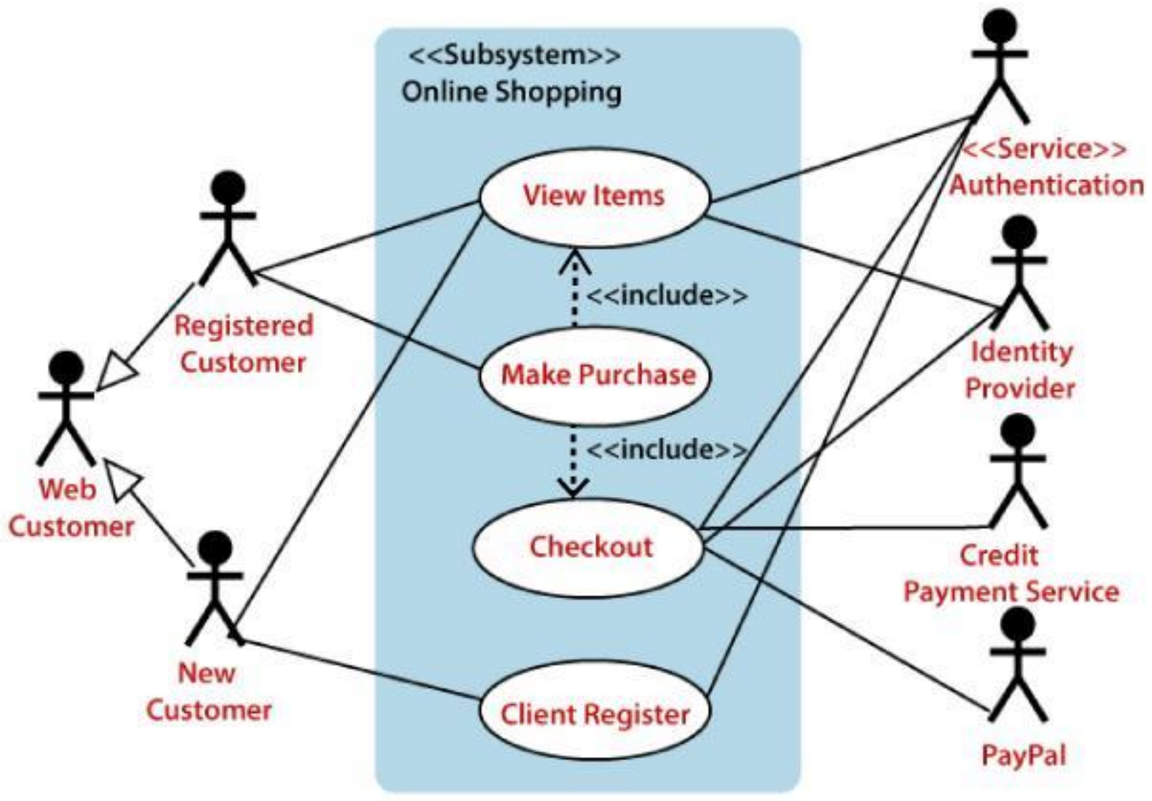
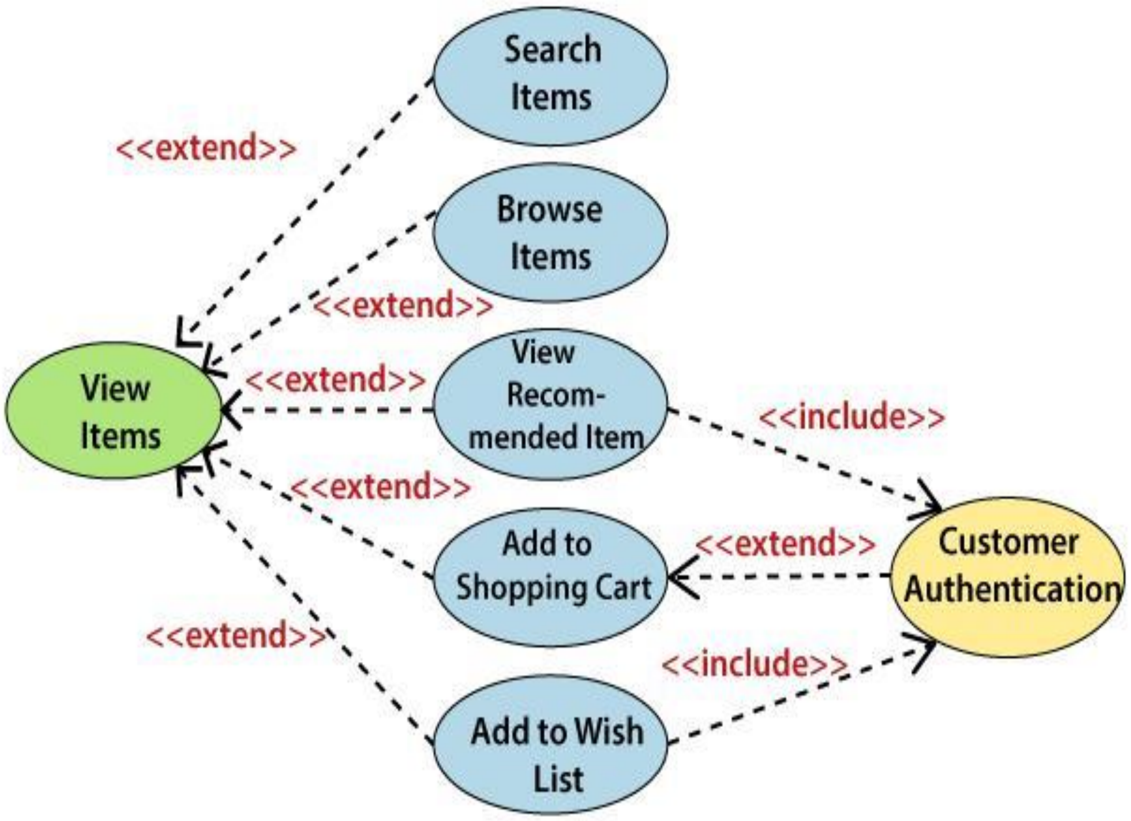
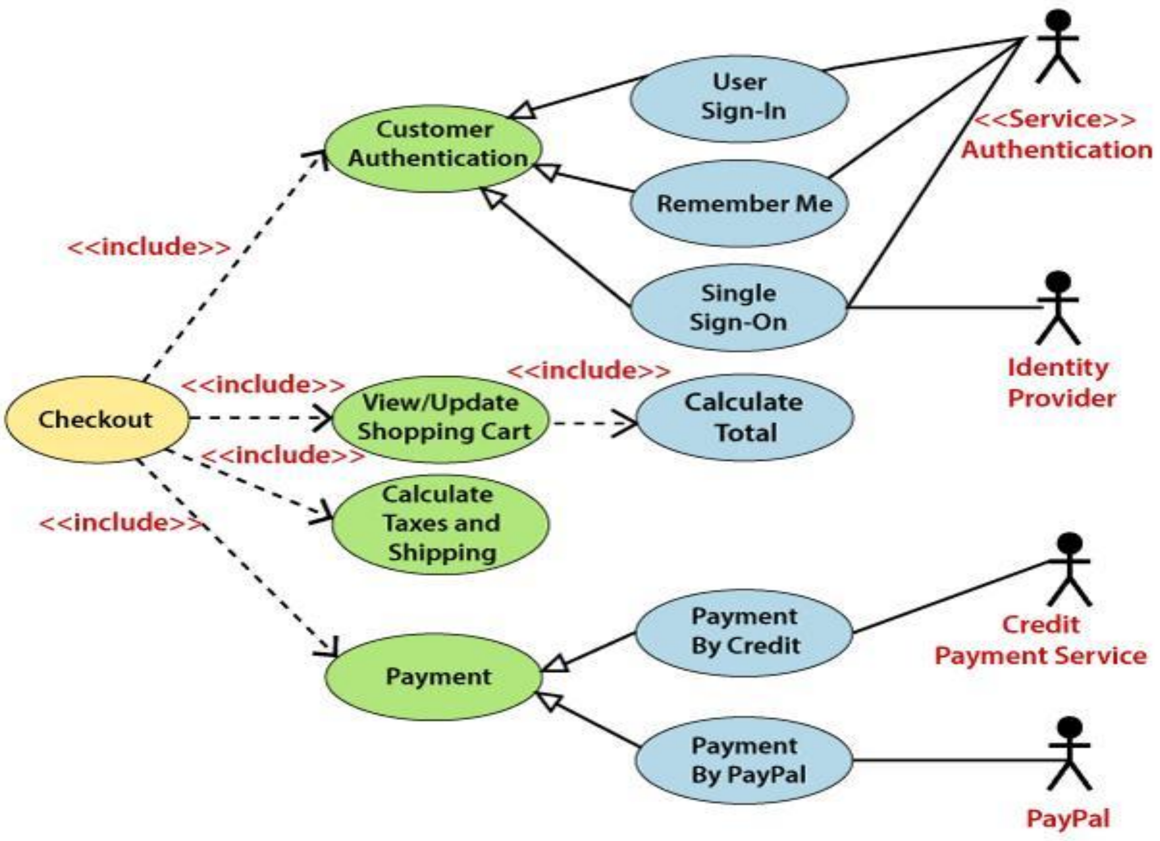


Figure: Sample Use Case diagram





Search Items
Browse Items
View Items
View Recom-mended Item
Add to Shopping Cart
Add to Wish List
Customer Authentication
<<extend>>
<<extend>>
<<extend>>
<<extend>>
<<extend>>
<<extend>>
<<include>>
<<include>>



Customer Authentication
User Sign-In
Remember Me
Single Sign-On
<<Service>> Authentication
Identity Provider
Checkout
<<include>>
<<include>>
View/Update Shopping Cart
<<include>>
Calculate Total
<<include>>
Calculate Taxes and Shipping
<<include>>
Payment
Payment By Credit
Payment By PayPal
Credit Payment Service
PayPal

Interaction Diagrams

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

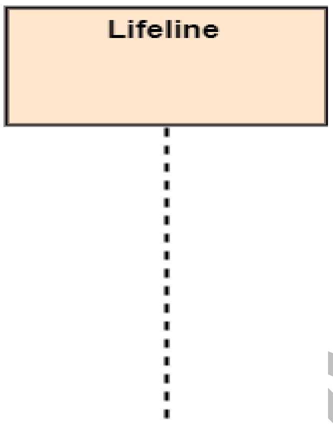
The purpose of interaction diagram is −

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

Notations of a Sequence Diagram

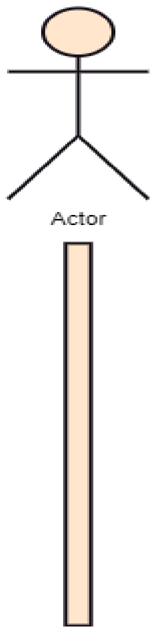
Lifeline

An individual participant in the sequence diagram is represented by a lifeline. It is positioned at the top of the diagram.



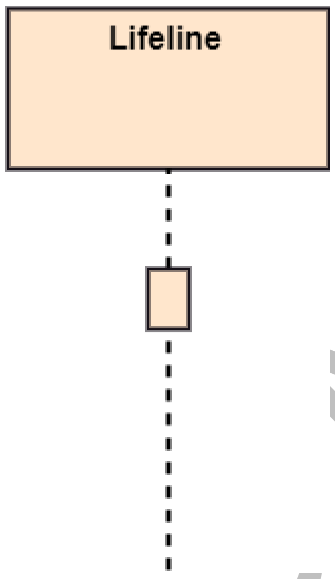
Actor

A role played by an entity that interacts with the subject is called as an actor. It is out of the scope of the system. It represents the role, which involves human users and external hardware or subjects. An actor may or may not represent a physical entity, but it purely depicts the role of an entity. Several distinct roles can be played by an actor or vice versa.



Activation

It is represented by a thin rectangle on the lifeline. It describes that time period in which an operation is performed by an element, such that the top and the bottom of the rectangle is associated with the initiation and the completion time, each respectively.

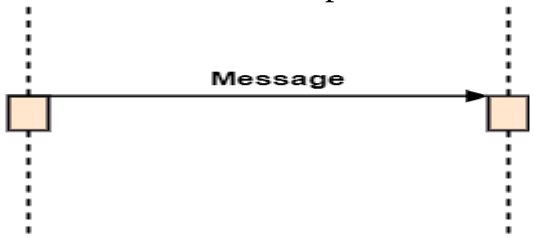


Messages--

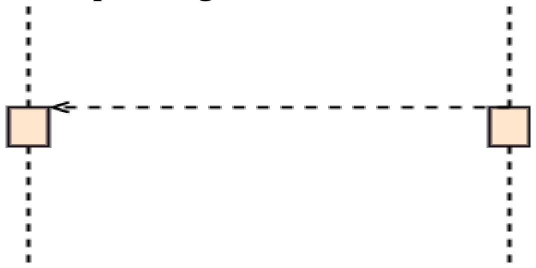
The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

Following are types of messages enlisted below:

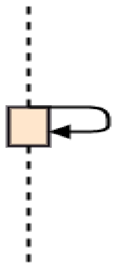
- **Call Message:** It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.



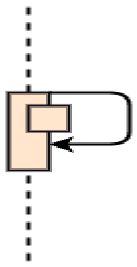
- **Return Message:** It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.



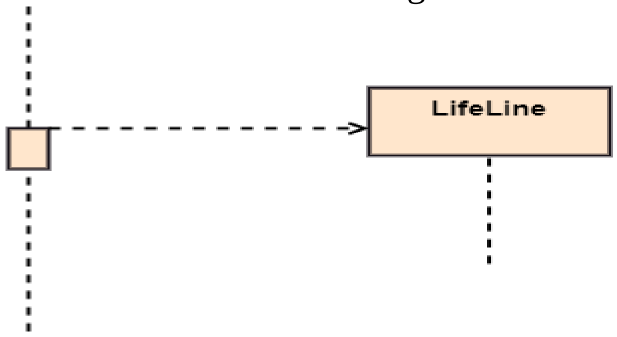
- **Self Message:** It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.



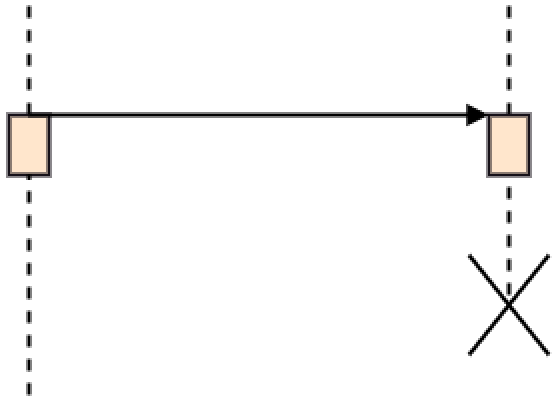
- **Recursive Message:** A self message sent for recursive purpose is called a recursive message. In other words, it can be said that the recursive message is a special case of the self message as it represents the recursive calls.



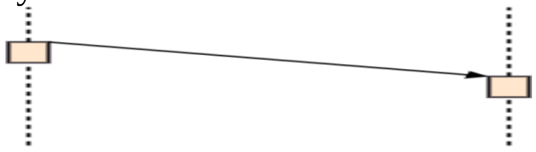
- **Create Message:** It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



- **Destroy Message:** It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.



- **Duration Message:** It describes a communication particularly between the lifelines of an interaction, which portrays the time passage of the message while modeling a system.

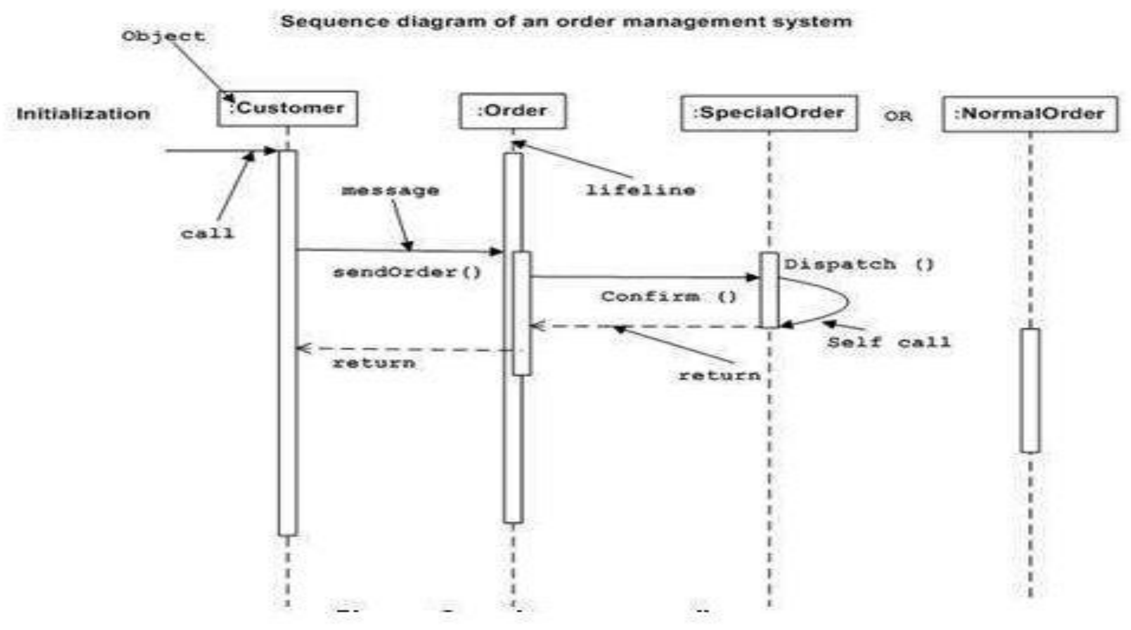


<u>The Sequence Diagram---</u>

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.



Benefits of a Sequence Diagram

1. It explores the real-time application.
2. It depicts the message flow between the different objects.
3. It has easy maintenance.
4. It is easy to generate.
5. Implement both forward and reverse engineering.
6. It can easily update as per the new change in the system.

The drawback of a Sequence Diagram

1. In the case of too many lifelines, the sequence diagram can get more complex.
2. The incorrect result may be produced, if the order of the flow of messages changes.
3. Since each sequence needs distinct notations for its representation, it may make the diagram more complex.
4. The type of sequence is decided by the type of message.

Example---

Figure shows a sequence diagram corresponding to the stock broker scenario. Each actor as well as the system is represented by a vertical line called a *lifeline* and each message by a horizontal arrow from the sender to the receiver. Time proceeds from top to bottom, but the spacing is irrelevant; the diagram shows only the sequence of messages, not their exact timing.

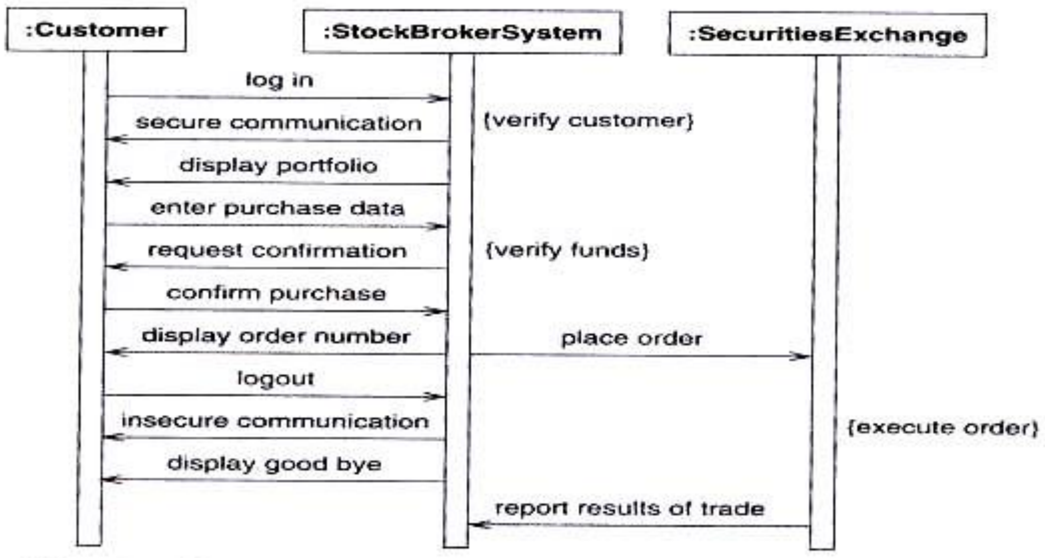


Figure **Sequence diagram for a session with an online stock broker.** A sequence diagram shows the participants in an interaction and the sequence of messages among them.

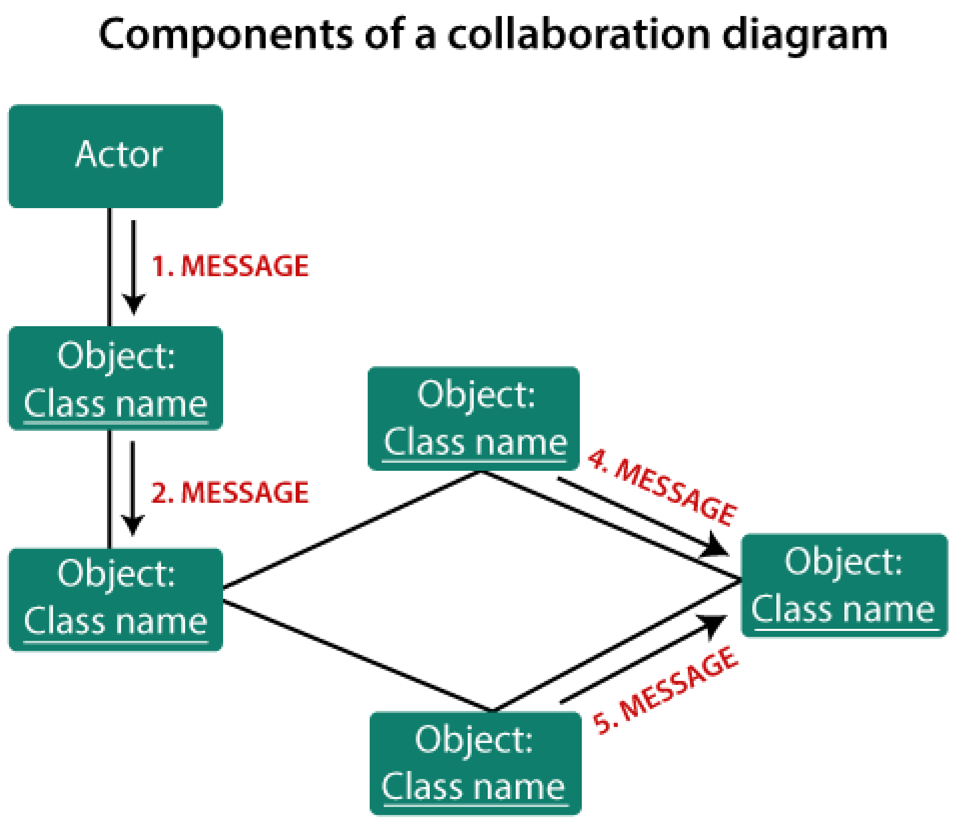
UML Collaboration / Communication Diagram

The collaboration diagram is used to show the relationship between the objects in a system. Both the sequence and the collaboration diagrams represent the same information but differently. Instead of showing the flow of messages, it depicts the architecture of the object residing in the system as it is based on object-oriented programming. An object consists of several features. Multiple objects present in the system are connected to each other. The collaboration diagram, which is also known as a communication diagram, is used to portray the object's architecture in the system.

Notations of a Collaboration/Communication Diagram

Following are the components of a component diagram that are enlisted below:

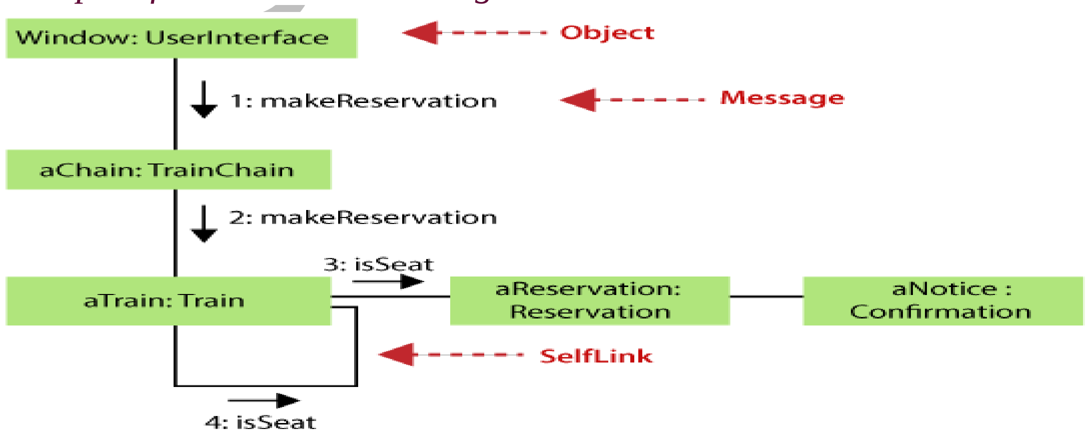
1. **Objects:** The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.
   In the collaboration diagram, objects are utilized in the following ways:
   - The object is represented by specifying their name and class.
   - It is not mandatory for every class to appear.
   - A class may constitute more than one object.
   - In the collaboration diagram, firstly, the object is created, and then its class is specified.
   - To differentiate one object from another object, it is necessary to name them.
2. **Actors:** In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
3. **Links:** The link is an instance of association, which associates the objects and actors. It portrays a relationship between the objects through which the messages are sent. It is represented by a solid line. The link helps an object to connect with or navigate to another object, such that the message flows are attached to links.
4. **Messages:** It is a communication between objects which carries information and includes a sequence number, so that the activity may take place. It is represented by a labeled arrow, which is placed near a link. The messages are sent from the sender to the receiver, and the direction must be navigable in that particular direction. The receiver must understand the message.



Steps for creating a Collaboration Diagram

1. Determine the behavior for which the realization and implementation are specified.
2. Discover the structural elements that are class roles, objects, and subsystems for performing the functionality of collaboration.
   - Choose the context of an interaction: system, subsystem, use case, and operation.
3. Think through alternative situations that may be involved.
   - Implementation of a collaboration diagram at an instance level, if needed.
   - A specification level diagram may be made in the instance level sequence diagram for summarizing alternative situations.

Example of a Collaboration Diagram



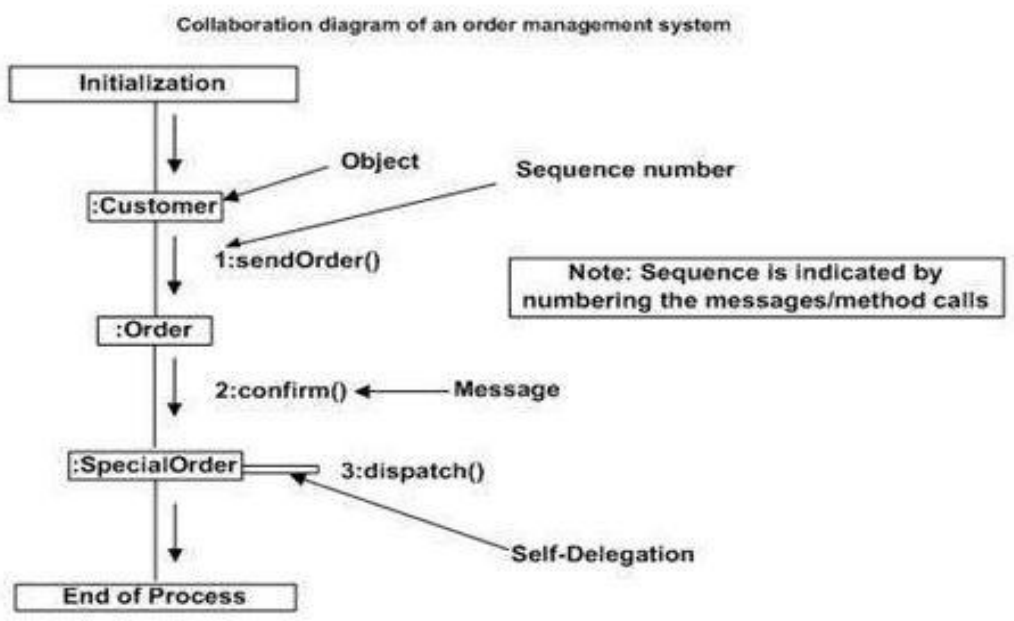
Benefits of a Collaboration Diagram

1. The collaboration diagram is also known as Communication Diagram.
2. It mainly puts emphasis on the structural aspect of an interaction diagram, i.e., how lifelines are connected.
3. The syntax of a collaboration diagram is similar to the sequence diagram; just the difference is that the lifeline does not consist of tails.
4. The messages transmitted over sequencing is represented by numbering each individual message.
5. The collaboration diagram is semantically weak in comparison to the sequence diagram.
6. The special case of a collaboration diagram is the object diagram.
7. It focuses on the elements and not the message flow, like sequence diagrams.
8. Since the collaboration diagrams are not that expensive, the sequence diagram can be directly converted to the collaboration diagram.
9. There may be a chance of losing some amount of information while implementing a collaboration diagram with respect to the sequence diagram.

The drawback of a Collaboration Diagram

1. Multiple objects residing in the system can make a complex collaboration diagram, as it becomes quite hard to explore the objects.
2. It is a time-consuming diagram.
3. After the program terminates, the object is destroyed.
4. As the object state changes momentarily, it becomes difficult to keep an eye on every single that has occurred inside the object of a system.

Example---



Activity Diagram--

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.

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

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

The purpose of an activity diagram can be described as −

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

Notation of an Activity diagram

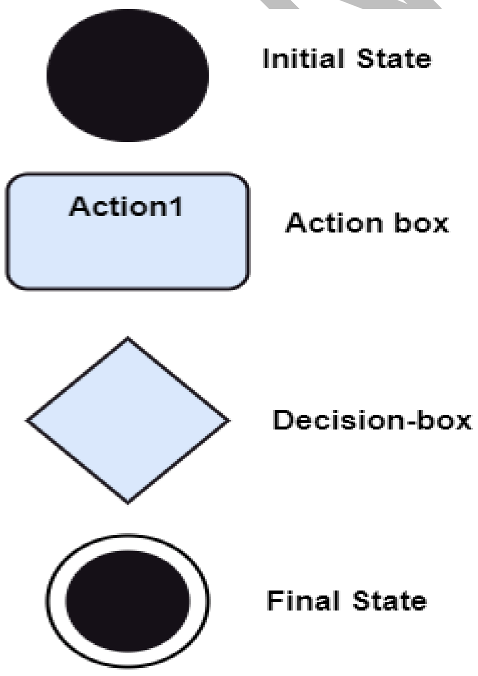
Activity diagram constitutes following notations:

Initial State: It depicts the initial stage or beginning of the set of actions.

Final State: It is the stage where all the control flows and object flows end.

Decision Box: It makes sure that the control flow or object flow will follow only one path.

Action Box: It represents the set of actions that are to be performed.



Following is an example of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

- Send order by the customer
- Receipt of the order
- Confirm the order
- Dispatch the order

After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process.

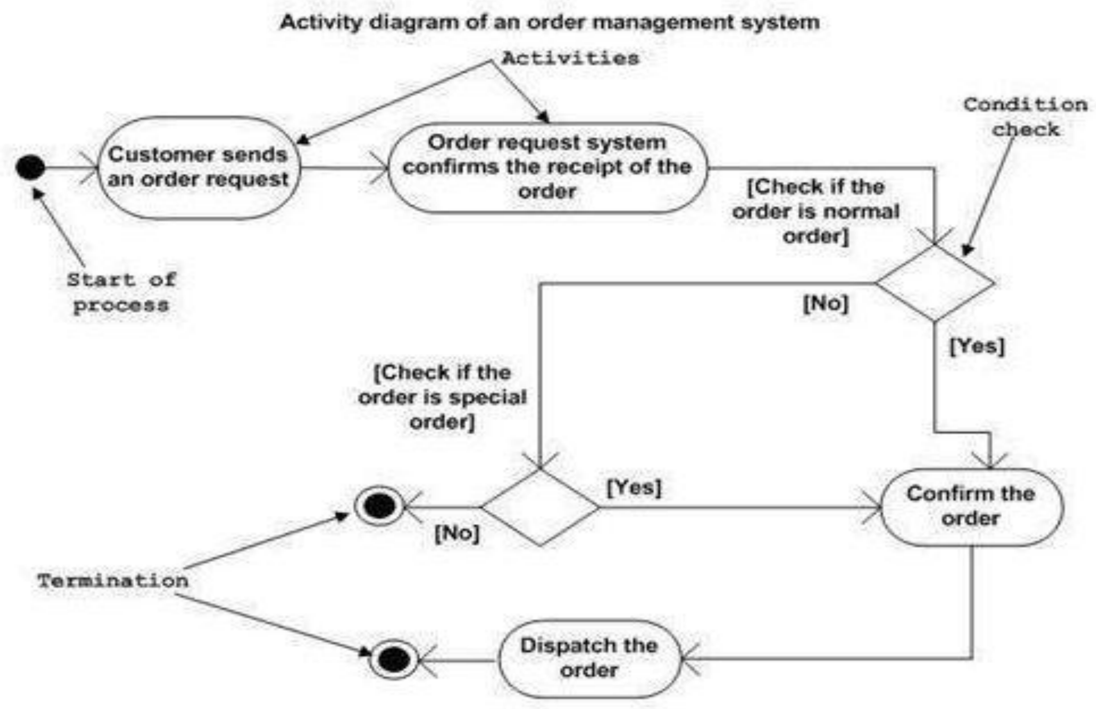

Activity diagram of an order management system
Activities
Customer sends an order request
Order request system confirms the receipt of the order
Condition check
[Check if the order is normal order]
Start of process
[No]
[Yes]
[Check if the order is special order]
[Yes]
[No]
Confirm the order
Termination
Dispatch the order

Figure below shows an activity diagram for the processing of a stock trade order that has been received by an online stock broker. The elongated ovals show activities and the arrows show their sequencing. The diamond shows a decision point and the heavy bar shows splitting or merging of concurrent threads.

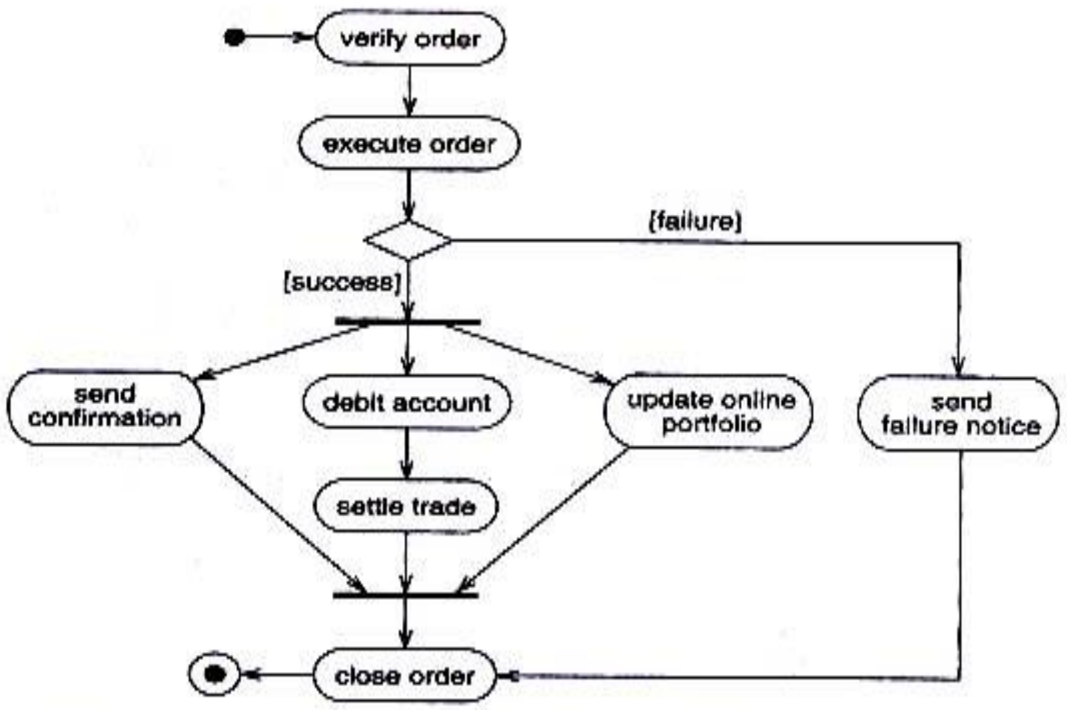
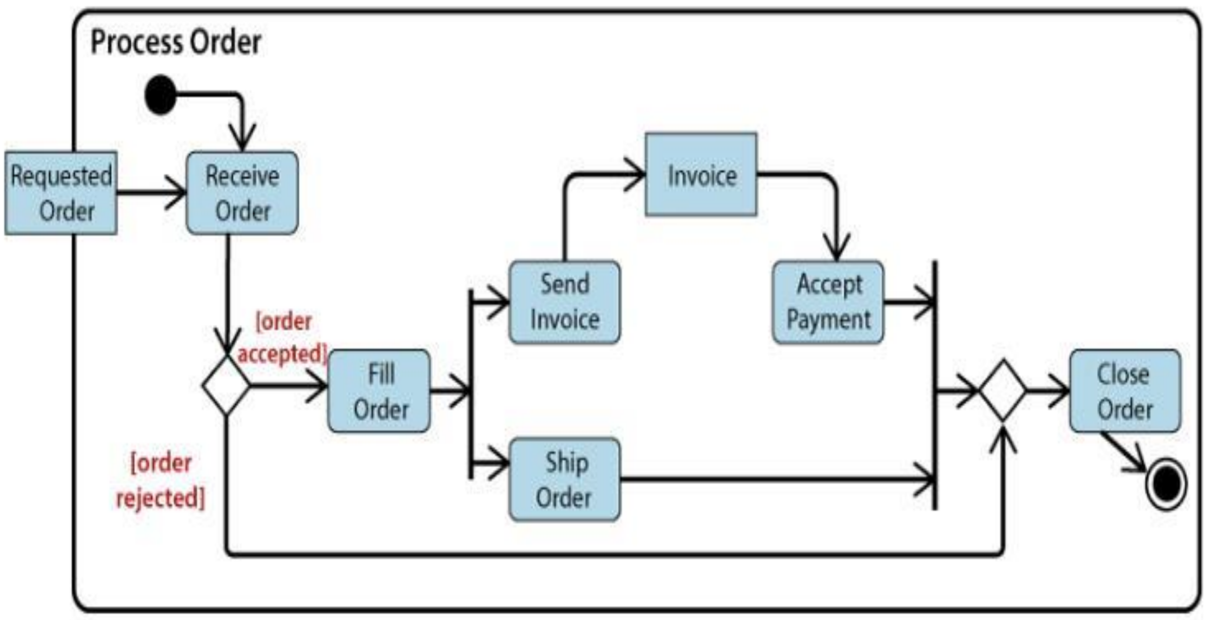


Figure **Activity diagram for stock trade processing.** An activity diagram shows the sequence of steps that make up a complex process.

<u>Guidelines for Activity Models--</u>



- Don't misuse activity diagrams. Activity diagrams are intended to elaborate use case and sequence models so that a developer can study algorithms and workflow.
- Level diagrams. Activities on a diagram should be at a consistent level of detail. Place additional detail for an activity in a separate diagram.

- Be careful with branches and conditions. If there are conditions, at least one must be satisfied when an activity completes—consider using an *else* condition.
- Be careful with concurrent activities. Concurrency means that the activities can complete in any order and still yield an acceptable result. Before a merge can happen, all inputs must first complete.
- Consider executable activity diagrams. Executable activity diagrams can help developers understand their systems better.

Statechart diagram--

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

Activity diagram explained in the next chapter, is a special kind of a Statechart diagram. As Statechart diagram defines the states, it is used to model the lifetime of an object.

Purpose of Statechart Diagrams

Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

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

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using Statechart diagrams −

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

How to Draw a Statechart Diagram?

Statechart diagram is used to describe the states of different objects in its life cycle. Emphasis is placed on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

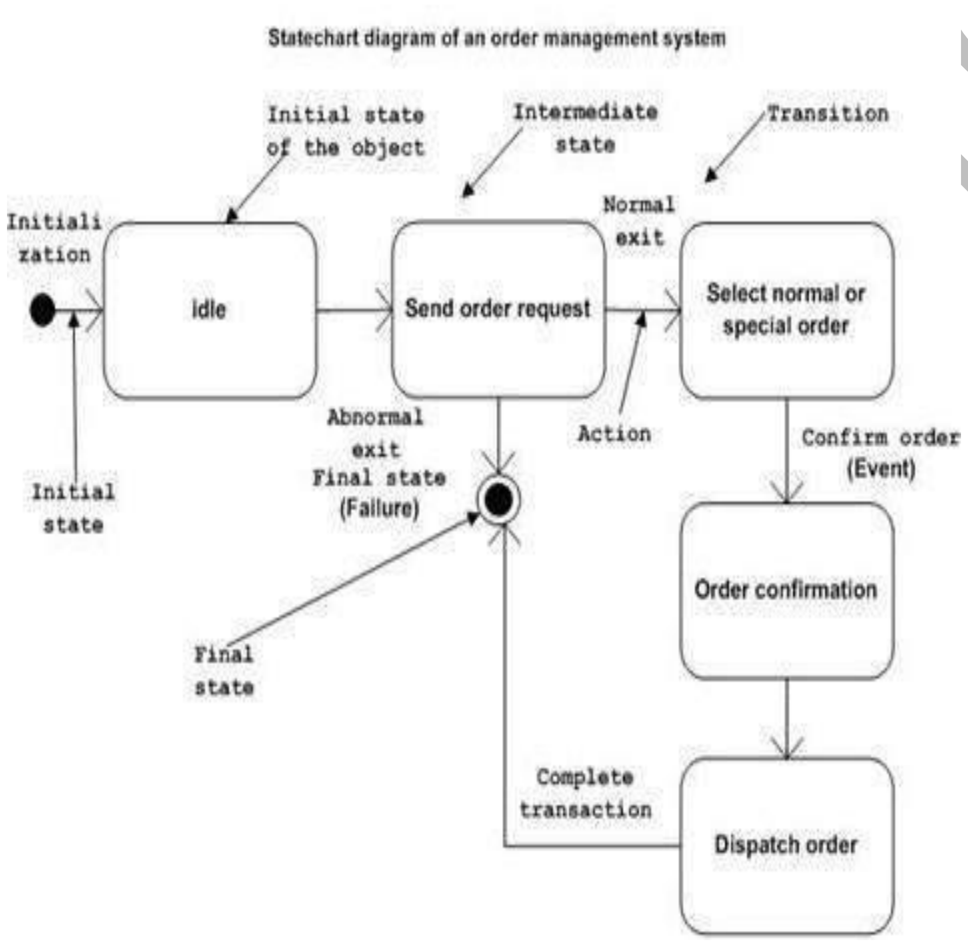
Before drawing a Statechart diagram we should clarify the following points −

- Identify the important objects to be analyzed.
- Identify the states.
- Identify the events.

Following is an example of a Statechart diagram where the state of Order object is analyzed

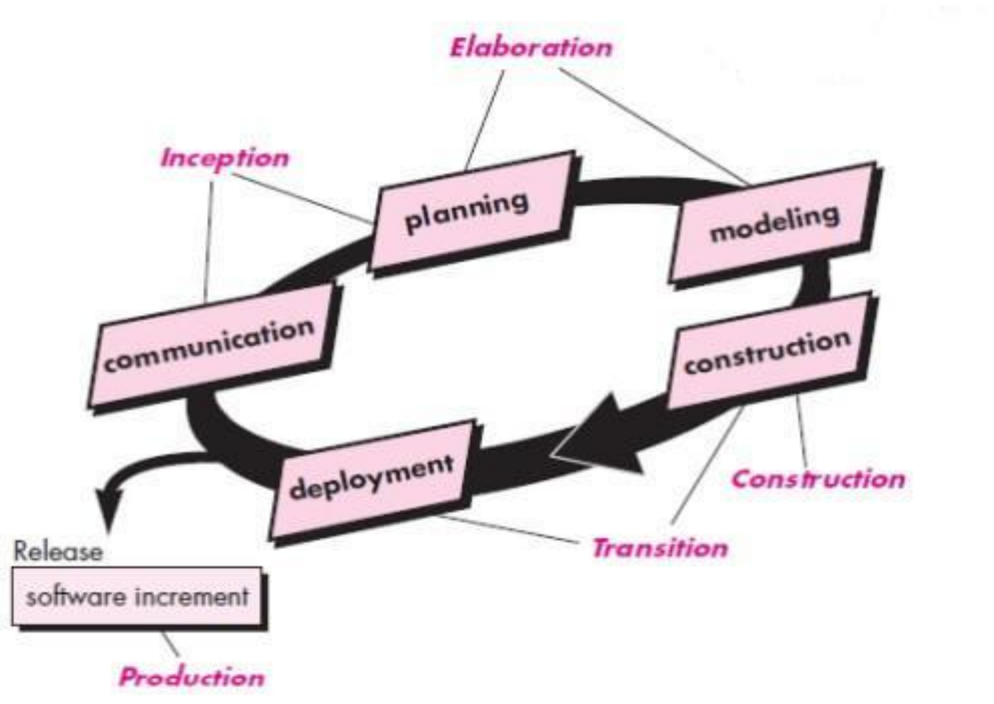
The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for the state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exits. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as a complete transaction as shown in the following figure. The initial and final state of an object is also shown in the following figure.



The Unified Process in Software Engineering---

Unified process (UP) is an architecture centric, use case driven, iterative and incremental development process. UP is also referred to as the unified software development process.



The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development. The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse" . It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Phases of the Unified Process

This process divides the development process into five phases:

- Inception
- Elaboration
- Conception
- Transition
- Production

The inception phase of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed.

The elaboration phase encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software the use case model, the requirements model, the design model, the implementation model, and the deployment model. Elaboration creates an "executable architectural baseline" that represents a "first cut" executable system.

The construction phase of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (the release) are then implemented in source code.

The transition phase of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing and user feedback reports both defects and necessary changes. At the conclusion of the transition phase, the software increment becomes a usable software release.

The production phase of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated. It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

OOD goodness criteria---

Coupling guidelines: The number of messages between 2 objects or among objects ought to be minimum. Excessive coupling between objects is decided to standard style and prevents reuse.

Cohesion guideline: In OOD, cohesion is regarding 3 levels:

1. **Cohesiveness of the individual methods:** The cohesiveness of every of the individual technique is fascinating since it assumes that every technique will solely a well-defined perform.
2. **Cohesiveness of the data and methods within a class:** This is fascinating since it assures that the ways of associate object do actions that the thing is, of course, accountable, i.e. it assures that no action has been improperly mapped to associate object.
3. **Cohesiveness of an entire class hierarchy:** The cohesiveness of ways among a category is fascinating since it promotes encapsulation of the objects.

Hierarchy and factoring guidelines: A base category mustn't have too several subclasses. If too several subclasses area unit derived from one base category, then it becomes troublesome to grasp the planning. In fact, there ought to about be no quite 7±2 categories derived from a base category at any level.

Keeping message protocols simple: Complex message protocols area unit is a sign of excessive coupling among objects. If a message needs quite three parameters, then it's a sign of dangerous style.

Number of Methods: Objects with an outsized range of ways area unit possible to be additional application-specific and conjointly troublesome to understand – limiting the likelihood of their employ.

Depth of the inheritance tree: The deeper a category is within the class inheritance hierarchy, the bigger is that the range of ways it's possible to inherit, creating it additionally advanced. Therefore, the peak of the inheritance tree mustn't be terribly giant.
Number of messages per use case: If ways of an outsized range of objects area unit invoked in a very chain action in response to one message, testing and debugging of the objects becomes difficult. Therefore, one message mustn't end in excessive message generation and transmission in a very system.

---END---