**Software Analysis Concepts and Principles**

The overall role of software in large system is identified during system engineering. However, it’s necessary to take a harder look at software’s role to understand the specific requirements that must be achieved to build high-quality software. That’s the job of software requirements analysis. To perform the job properly you should follow a set of underlying concepts and principles.

**1. Requirements Analysis**

Requirement analysis is a software engineering task that bridges the gap between system level requirements engineering and software design. Requirements engineering activities result in the specification of software’s operational characteristics, indicate software’s interface with other system elements, and establish constraints that software must meet. Requirement analysis allows the software engineer to refine domains that will be treated by software.

Software requirements analysis may be divided into five areas of effort:

(1)     problem recognition,

(2)     evaluation and synthesis,

(3)     modeling,

(4)     specification, and

(5)     review.

The analyst studies the *system specification* and the *software Project Plan*. It is important to understand software in a system context and to review the software scope that was used to generate planning estimates. Problem evaluation and solution synthesis is the next major area of effort for analysis. The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behavior in the context of events that affect the system, establish system interface characteristics, and uncover additional design constraints.

Throughout evaluation and solution synthesis, the analyst’s primary focus is on “what” not “how”. What data does the system produce and consume, what functions must the system perform, what behavior does the system exhibit, what interfaces are defined and what constraints apply?

**2. Requirements Elicitation for Software**

Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process.

**2.1 Initiating the Process**

The first meeting between a software engineer and the customer can be liked to the awkwardness of a first date between two adolescents. Communication must be initiated by asking *context-free questions*. That is a set of questions that will lead to basic understanding of the problem, the people who want a solution that will lead to basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself.

* Who is behind the request for this work?
* Who will use the solution?
* What will be the economic benefit of a successful solution?
* Is there another source for the solution that you need?

The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution:

* How would you characterize “good” output that would be generated by a successful solution?
* What problem(s) will this solution address?
* Can you show me the environment in which the solution will be used?
* Will special performance issues or constrains affect the way the solution is approached?

**2.2 Facilitated Application Specification Techniques**

Customers and software engineers have an unconscious “us and them” mind-set. With these problems in the mind that a number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification. Called *facilitated application technique (****FAST****)*. Basic guidelines for this technique are:

* A meeting is conducted at a neutral site and attended by both software engineers and customers.
* Rules for preparation and participation are established.
* An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
* A “facilitator” controls the meeting.
* A “definition mechanism” is used
* The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conductive to the accomplishment of the goal.

Initial meeting between the developer and customer occur and basic questions and answers help to establish the scope of the problem and the overall perception of a solution. The product request distributed to all attendees before the meeting date. The FAST team is composed of representatives from marketing, software and hardware engineering, and manufacturing. As the FAST meeting begins, the first topic of discussion is the need and justification for the new product – everyone should agree that the product justified. Once agreement has been established, each participant his or her list for discussion.

After individual lists are presented in one topic area, a combined list is created by the group. The combined list eliminates redundant entries, adds any new ideas that come up during the discussion, but does not delete anything. The combined list is shortened, lengthened, or reworded to properly reflect the product or system to be developed. The objective is to develop a *consensus list* in each topic area. Each sub team presents its mini-specs to all FAST attendees for discussion. After the mini-specs are completed, each FAST attendee makes a list of *validation criteria*for the product or system and presents his or her to the team.

**2.3 Quality Function Deployment**

*Quality function deployment (*QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD identifies three types of requirements:

**Normal requirements.** The objectives and goals that are stated for a product or system during meeting with customer. If these requirements are present, the customer is satisfied.

**Expected requirements.** These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.

**Exciting requirements.** These features go beyond the customer’s expectations and prove to be very satisfying when present.

*Functional deployment* is used to determine the value of each function that is required for the system. *Information deployment* identifies both the data objects and events that the system must consume and produce. These are tied to the functions. Finally, *task deployment*examines the behavior of the system or product within the context of its environment. *Value analysis* is conducted to determine the relative priority of requirements determined during each of the three deployments.

**2.4. Use-Cases**

As requirements are gathered as part of informal meetings, a software engineer can create a set of scenarios that identify a thread of usage for the system to be constructed. To create a use-case, the analyst must first identify the different types of people play as the system operates. Defined somewhat more formally an actor is anything that communicates with the system or product and that is external to the system itself.

It’s most important to note that an actor and a user are not the same thing. An actor represents a class of external entities that play just one role. Once actors have been identified, use-case can be developed. The use-case describes the manner in which an actor interacts with the system. The use-case should be answer below questions:

ü        What main tasks or functions are performed by an actor?

ü        What system information will the actor acquire, produce, or change?

ü        Will the actor have to inform the system about changes in the external environment?

ü        What information does the actor desire from the system?

ü        Does the actor wish to be informed about unexpected changes?

In general, use-case is simply a written narrative that describes the role of an actor as interaction with the system occurs.

**3. Analysis Principles**

Over the past two decades, a large number of analysis modeling methods have been developed. Investigators have identified analysis problems and their causes and have developed a variety of notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view.

* The information domain of a problem must be represented and understood.
* The functions that the software is to perform must be defined.
* The behavior of the software must be represented.
* The models that depict information, function, and
* The models that depict information function and behavior must be partitioned in a manner that uncovers details in a layered fashion.
* The analysis process should move from essential information toward implementation detail.

In addition to these operational analysis principles for requirements engineering:

* *Understand the problem before you begin to create the analysis model.*
* *Develop prototype that enable a user to understand how human/machine interaction will occur.*
* *Record the origin of and the reason for every requirement.*
* *Use multiple views of requirements.*
* *Rank requirements*.
* *Work to eliminate ambiguity*

**Software prototyping**

It is the process of implementing the presumed software requirements with an intention to learn more about the actual requirements or alternative design that satisfies the actual set of requirements.

**Need for software prototyping**

* To assess the set of requirements that makes a product successful in the market
* To test the feasibility without building the whole system.
* To make end-user involved in the design phase

**Benefits of Software Prototyping**

* It makes the developers clear about the missing requirements. Lets the developers know what actually the users want.
* Reduces the loss by bringing the manufacturer to a conclusion weather the system which we are about to build is feasible or not rather than building the whole system and finding it.
* One can have a working system in beforehand.
* It brings the user to get involved in the system design

**Types of Prototyping**

* Throw away prototyping
* Evolutionary prototyping
* Operational prototyping

**Throw away prototyping**

* Objective - Derive end system requirements
* Throw away prototyping is one type of approach where an initial prototype is built mainly focusing on the poorly understood requirements
* Once the requirements are understood requirements document is updated anda conventional development process is followed to build system
* Used to reduce requirements risk
* The prototype is developed from an initial specification, delivered for experiment then discarded
* The throw-away prototype should NOT be considered as a final system
* Some system characteristics may have been left out
* There is no specification for long-term maintenance
* The system will be poorly structured and difficult to maintain

**Evolutionary Prototyping**

* Objective – Deliver a working system + requirements
* Evolutionary prototyping is the one in which a system is build using the well understood requirements.
* Specification, design and implementation are inter-twined
* The system is developed as a series of increments that are delivered to the customer
* Techniques for rapid system development are used such as CASE tools and 4GLs
* User interfaces are usually developed using a GUI development toolkit

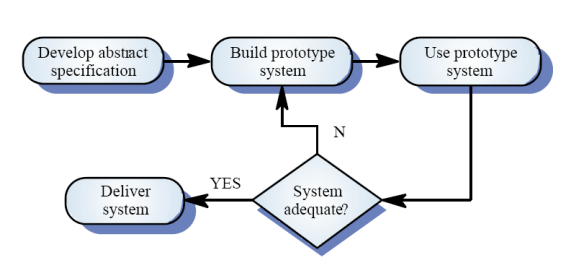


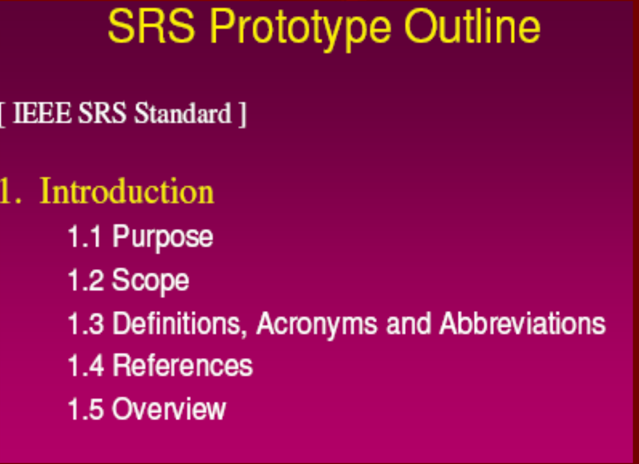
Fig: Evolutionary Prototype

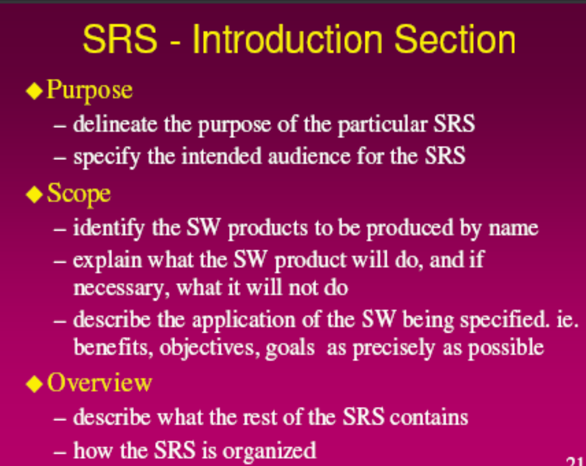
**Operational Prototyping**

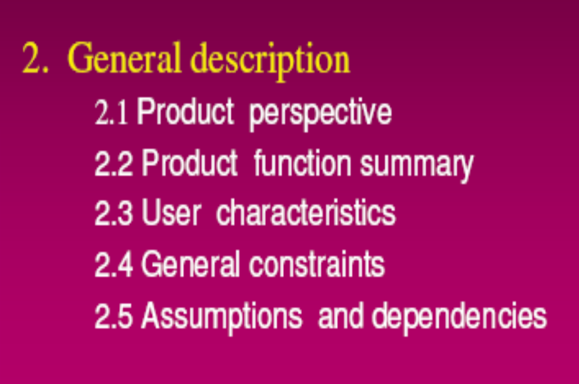
* Used when requirements are either critical and understood or not critical and poorly understood
* Throw away prototypes are selectively built on top of evolutionary prototype
* A trained prototyper keeps track of user.

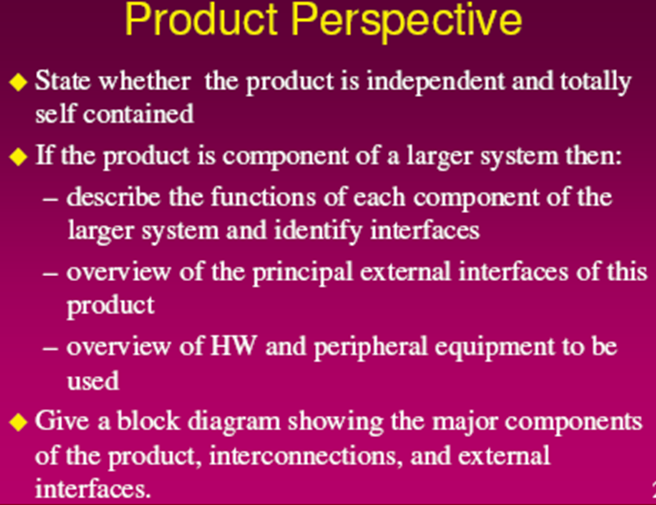
**Software requirement specification**

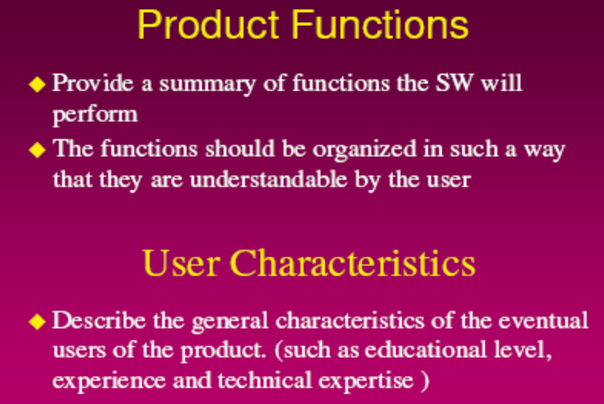
* The software requirement specification is produced at the culmination of the analysis task.
* SRS document is a contract between the development team and the customer.
* The SRS document is known as black-box specification since the system is considered as a black box whose internal details are not known and only its external is visible





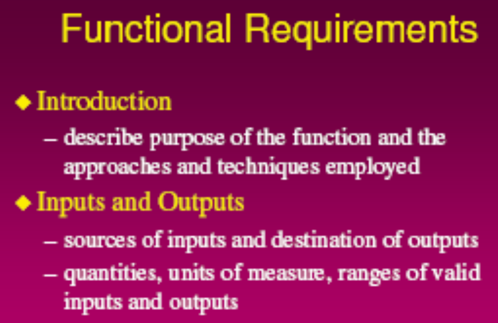




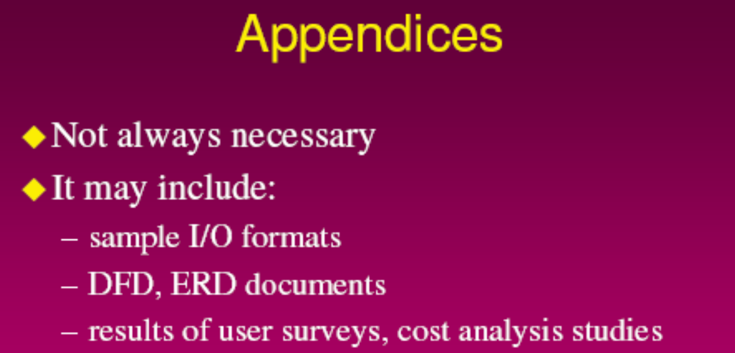












**Specification Review**

* SRS-foundation of the development stage.
* Review is conducted to ensure completeness, accuracy and consistency in SRS.
* Avoid vague terms in SRS (usually, often)
* Once review is complete SRS is signed off by both customer and developer.

**Analysis model**

* Provides the first technical representation of a system
* Is easy to understand and maintain
* Deals with the problem of size by partitioning the system
* Uses graphics whenever possible
* Differentiates between essential information versus implementation information
* Helps in the tracking and evaluation of interfaces
* Provides tools other than narrative text to describe software logic and policy

**Objectives of analysis model**

* To describe what the customer require
* To establish a basis for the creation of a software design
* To define a set of requirements that can be validated once the software is built.

Uses a combination of text and diagrams to show requirement

**Elements of Analysis model**

**Scenario-based elements:** The system is described from the user’s point of view using a scenario-based approach. For example, basic **use cases** and their corresponding use-case diagrams, and UML activity diagram.

**Class-based elements:** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a **UML class diagram**

**Behavioral elements**: The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior. The **state diagram** is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A state is any externally observable mode of behavior. In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event.

**Flow-oriented elements:** Information is transformed as it flows through a computerbased system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. For example, **data flow diagram** (DFD) and control flow diagram (CFD).

**Data modeling**

Defines all data that are entered, stored, transformed and produced within an application.

Identify the following items

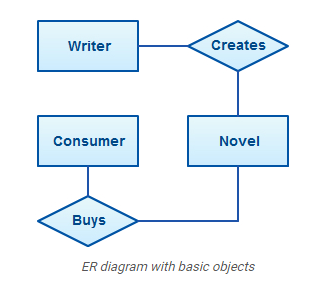
– Data objects (Entities)

– Data attributes

– Relationships

– Cardinality (number of occurrences)

An **Entity Relationship Diagram (ERD)(used for data modeling)** is a visual representation of different entities within a system and how they relate to each other. For example, the elements writer, novel, and a consumer may be described using ER diagrams the following way:

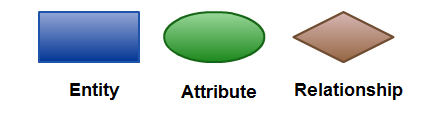
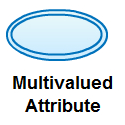


**How to Draw ER Diagrams**

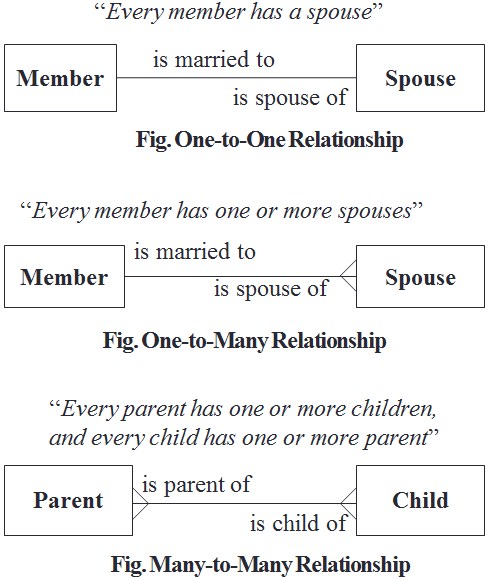
Below points show how to go about creating an ER diagram.

* Identify all the entities in the system. An entity should appear only once in a particular diagram. Create rectangles for all entities and name them properly.
* Identify relationships between entities. Connect them using a line and add a diamond in the middle describing the relationship.
* Add attributes for entities. Give meaningful attribute names so they can be understood easily.

**ER Diagram Symbols and Notations**

**Cardinality**

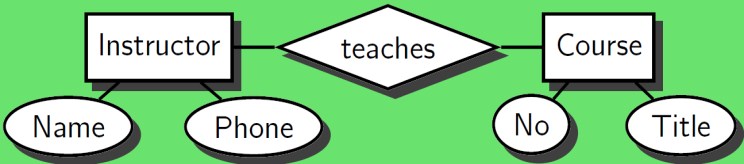
Specification of the number of occurrences of one object that can be related to the number of occurrences of another object.

Possible relationships are:

* One-to-One
* One-to-Many
* Many-to-Many

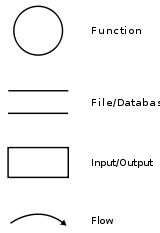
Example: For example: ER-diagram about instructors and courses.

* Instructors teach courses.
* Instructors have a name and a phone number.
* Courses are numbered and have titles.



**Functional Modeling**

Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies hardware, software, and human elements to transform it; and produces output in a variety of forms. Regardless of size and complexity, the flow model for any computer-based system can be created. Data flow diagram (DFD) and Control flow diagram (CFD) are two common flow models. DFD or CFD have various graphical notations. A rectangle is used to represent an external entity such as hardware, a person, another program or another system that produces or receives information. A circle (or bubble) represents a process or transform that is applied to data (or control) and changes it in some way. An arrow represents one or more data items (data objects). The double line represents a data store.



Data Flow Diagrams (DFD)

The DFD takes an input-process-output view of a system. A data flow diagram is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram, also known as a data flow graph or a bubble chart. The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail.

A level 0 DFD, also called a fundamental system model or a context model, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. The system of level-0 DFD is decomposed into lower-level DFD(Level-1) with the set of processes, data stores, and the data flows between these processes and data stores. Each process of level-1 is further decomposed into an even lower level diagram containing its sub-processes. This approach continues on subsequent sub-processes until the necessary and sufficient level of detail is reached which is called primitive process.

**Example − Let us consider a software system, Wholesaler Software, that automates the transactions of a wholesale shop. The shop sells in bulks and has a clientele comprising of merchants and retail shop owners. Each customer is asked to register with his/her particulars and is given a unique customer code, C\_Code. Once a sale is done, the shop registers its details and sends the goods for dispatch. Each year, the shop distributes Christmas gifts to its customers, which comprise of a silver coin or a gold coin depending upon the total sales and the decision of the proprietor.**

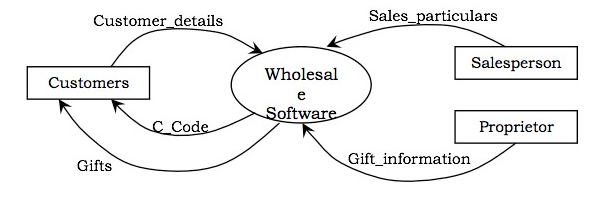
The functional model for the Wholesale Software is given below. The figure below shows the top-level DFD (context level). It shows the software as a single process and the actors that interact with it.

The actors in the system are −

Customers

Salesperson

Proprietor



In the next level DFD, as shown in the following figure, the major processes of the system are identified, the data stores are defined and the interaction of the processes with the actors, and the data stores are established.

**In the system, three processes can be identified, which are −**

Register Customers

Process Sales

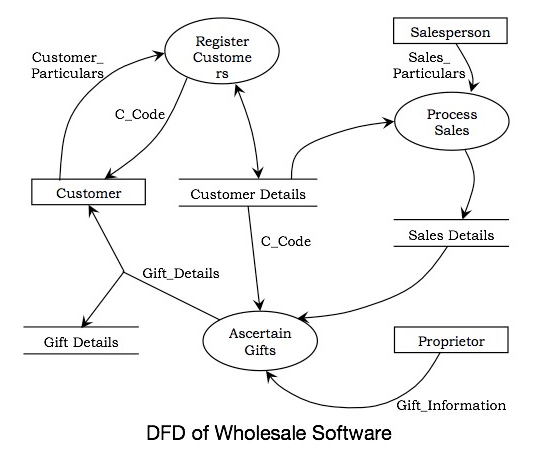
Ascertain Gifts

**The data stores that will be required are −**

Customer Details

Sales Details

Gift Details



**Draw level 0 and level1 DFD for the following scenario.!!!!!!!!!!!**

Customer can place an Order. The Order Food process receives the Order, forwards it to the Kitchen, store it in the Order data store, and store the updated Inventory details in the Inventory data store. The process also delivers a Bill to the Customer.

The Manager can receive Reports through the Generate Reports process, which takes Inventory details and Orders as input from the Inventory and Order data store respectively.

The Manager can also initiate the Order Inventory process by providing Inventory order. The process forwards the Inventory order to the Supplier and stores the updated Inventory details in the Inventory data store.

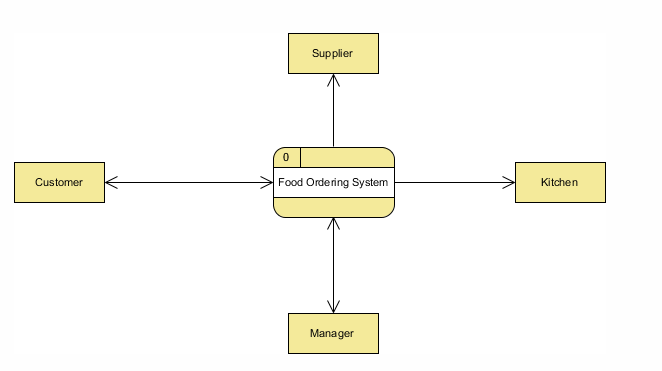


FIG: Level 0 DFD (Context Diagram)

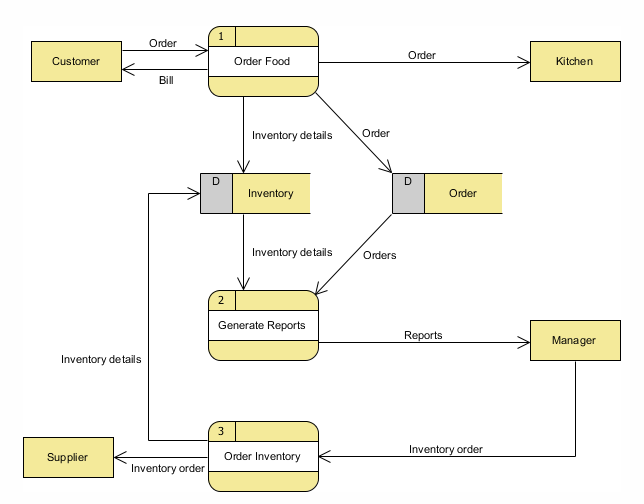


FIG: Level 1 DFD

**Behavioral modeling**

The behavioral model indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within the system.

2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.

3. Create a sequence for each use case.

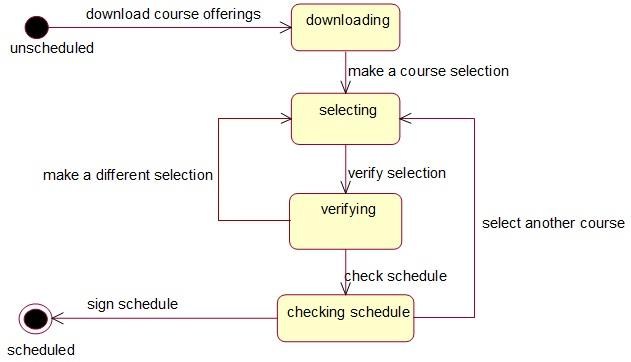
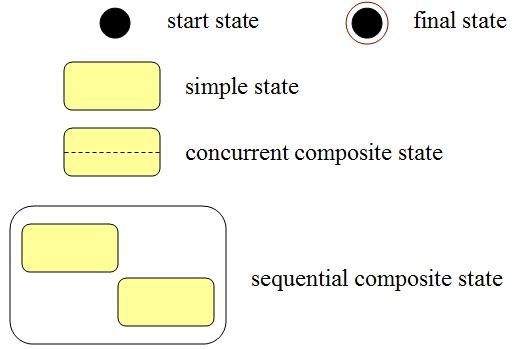
4. Build a state diagram for the system.

5. Review the behavioral model to verify accuracy and consistency.

The UML's behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system where the dynamic aspects of a system represent its changing parts. The behavioral diagram used to show how the system evolves over time (responds to requests, events etc.). For a house, the airflow through room is dynamic aspect & similarly the flow of messages over time due to the physical movement of components across a network is dynamic aspect for the software system. The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

**State chart Diagram (For behavioral modeling)**

A state-chart diagram shows a state machine, consisting of states, transitions, events, and activities. They are especially important in modeling the behavior of an interface, class, or collaboration. State-chart diagrams emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.



**Structured Analysis**

Structured Analysis is a development method that allows the analyst to understand the system and its activities in a logical way.

It is a systematic approach, which uses graphical tools that analyze and refine the objectives of an existing system and develop a new system specification which can be easily understandable by user.

It has following attributes −

* It is graphic which specifies the presentation of application.
* It divides the processes so that it gives a clear picture of system flow.
* It is logical rather than physical i.e., the elements of system do not depend on vendor or hardware.
* It is an approach that works from high-level overviews to lower-level details.

## Data Dictionary

A data dictionary is a structured repository of data elements in the system. It stores the descriptions of all DFD data elements that is, details and definitions of data flows, data stores, data stored in data stores, and the processes.

A data dictionary improves the communication between the analyst and the user. It plays an important role in building a database. Most DBMSs have a data dictionary as a standard feature. For example, refer the following table –

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Data Name** | **Description** | **No. of Characters** |
| 1 | ISBN | ISBN Number | 10 |
| 2 | TITLE | title | 60 |
| 3 | SUB | Book Subjects | 80 |
| 4 | ANAME | Author Name | 15 |