SOFTWARE REQUIREMENT ANALYSIS PHASE

Requirements cover not only the desired functionality of a system or software product, but also address non-functional issues (e.g., performance, interface, and reliability requirements), constraints on the design (e.g., operating in conjunction with existing hardware or software), and constraints on the implementation (e.g., cost, must be programmed in Ada). The process of determining requirements for a system is referred to as requirements definition. Where software is a component of another system, software requirements can be distinguished from the system requirements of the larger artifact.

Requirements products have three, sometimes competing objectives:

1) To achieve agreement regarding the requirements between system developers, customers, and end-users.
2) To provide the basis for software design.
3) To support verification and validation.

The requirements definition process comprises these steps:

a) Requirements identification
b) Identification of software development constraints
c) Requirements analysis
d) Requirements representation
e) Requirements communication
f) Preparation for validation of software

Classes of project participants are:
Customers contract for the software project and are assumed to be responsible for acceptance of the software. Requirements analysts act as catalysts in identifying requirements from the information gathered from many sources, in structuring the information and in communicating draft requirements to different audiences. End-users, install operate, use, and maintain the system incorporating the software.

The principal objectives of requirements products are:

a) To achieve agreement on the requirements: Requirements definition, the process culminating in the production of requirements products, is a communications-intensive one that involves the iterative elicitation and refinement of information from a variety of sources, usually including end-users with divergent perceptions. The requirements definition process of what is needed. Frequent review of the evolving requirements by persons with a variety of backgrounds is essential to the convergence of this iterative process. Requirements documents must facilitate communication with end-users and customers, as well as with software designers and personnel who will test the software when developed.
b) To provide a basis for software design: Requirements documents must provide precise input to software developers who are not experts in the application domain. However, the precision required for this purpose is frequently at odds with the need for requirements documents to facilitate communication with other people.
c) To provide a reference point for software validation: Requirements documents are used to perform software validation, i.e., to determine if the developed software satisfies the requirements from which it was developed. Requirements must be stated in a measurable form, so tests can be developed to show unambiguously whether each requirement has been satisfied.

Software Requirements Process

The requirements process typically consists of three basic tasks: problem or requirement analysis, requirement specification, and requirements validation. Problem analysis often starts with a high-level

"problem statement." During analysis the problem domain and the environment are modeled in an effort to understand the system behavior, constraints on the system, its inputs and outputs, etc. The basic purpose of this activity is to obtain a thorough understanding of what the software needs to provide. The understanding obtained by problem analysis forms the basis of requirements specification, in which the focus is on clearly specifying the requirements in a document. Issues such as representation, specification languages, and tools, are addressed during this activity. As analysis produces large amounts of information and knowledge with possible redundancies; properly organizing and describing the requirements is an important goal of this activity. Requirements validation focuses on ensuring that what has been specified in the SRS are indeed all the requirements of the software and making sure that the SRS is of good quality. The requirements process terminates with the production of the validated SRS. Though it seems that the requirements process is a linear sequence of these three activities, in reality it is not so for anything other than trivial systems. In most real systems, there is considerable overlap and feedback between these activities. So, some parts of the system are analyzed and then specified while the analysis of the other parts is going on. Furthermore, if the validation activities reveal problems in the SRS, it is likely to lead to further analysis and specification. However, in general, for a part of the system, analysis precedes specification and specification precedes validation.

Once the specification is "complete" it goes through the validation activity. This activity may reveal problems in the specifications itself, which requires going back to the specification step, or may reveal shortcomings in the understanding of the problem, which requires going back to the analysis activity. During requirements analysis the focus is on understanding the system and its requirements. For a complex system, this is a hard task, and the time-tested method of "divide-and-conquer," i.e., decomposing the problem or system into smaller parts and then understanding the parts and their relationships, is inevitably applied to manage the complexity. Also, for managing the complexity and the large volume of information that becomes available during analysis, various structures are used during analysis to represent the information to help view the system as a series of abstractions. Examples of these structures are data flow diagrams and object diagrams. For a portion of the system, analysis typically precedes specification. Once the analysis is complete and the structures built, the system part has to be specified.

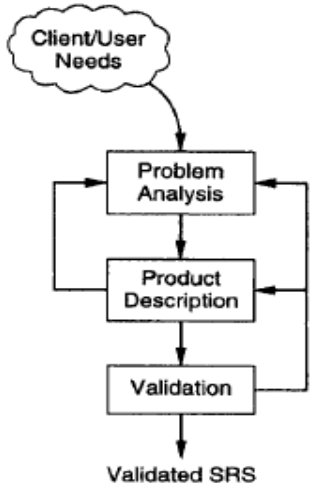


Fig The requirement process.

[The transition from analysis to specification is not simple due to the different objectives of the two activities. In specification, we have to specify only what the software is supposed to do, i.e., focus on the external behavior of the system. In order to identify all the external behaviors, the structure of the

problem and its various components need to be clearly understood besides understanding its inputs and outputs. However, the structure itself may not be of much use in specification, as its focus is exclusively on the external behavior or the eventual system, not the internal structure of the problem domain. Due to this, one should not expect that once the analysis is done, specification will be straightforward. Furthermore, many "outputs" of the analysis are not used directly in the SRS. This does not mean that these outputs are not useful—they are essential in modeling the problem that leads to the proper understanding of the requirements, which is a prerequisite to specification. Hence, the use of the analysis activity and structures that it built may be indirect, aiding understanding rather than directly aiding specification.]

Also, there is the issue of the level of detail that the requirement process should aim to uncover and specify. This is also an issue that cannot be easily resolved and that depends on the objective of the requirement specification phase. If the objective is to define the overall broad needs of the system, the requirements can be very abstractly stated. Generally, the purpose of such requirements is to perform some feasibility analysis or use the requirements for competitive bidding. At the lower level are the requirements where all the behavior and external interfaces of the software are clearly specified. Such requirements are clearly very detailed and are suitable for software development. An example can illustrate this point. Suppose a car manufacturer wants to have an inventory control system. At an abstract level, the requirements of the inventory control system could be stated in terms of the number of parts it has to track, level of concurrency it has to support, whether it will be on-line or batch processing, what types of information and reports it will provide (e.g., status of each item on demand, purchase orders for items that are low in inventory, consumption patterns), etc. Requirements specification at this level of abstraction can be used to estimate the costs and perform a cost-benefit analysis. It can also be used to invite tenders from various developers. However, such a requirements specification is of little use for a developer given the contract to develop the software. That developer needs to know the exact format of the reports, all the queries that can be performed and their structure, total number of terminals the system has to support, the structure of the major databases that will exist, etc. These are all specifying the external behavior of the software, but when viewed from the higher level of abstraction they can be considered as specifying how the abstract requirements should be implemented (e.g., the details of a report can be viewed as defining how the basic objective of providing information is satisfied).
Hence, the SRS is used to provide all the detailed information needed by a software developer for properly developing the system. The three major activities in the requirements phase, viz. analysis, specification, and verification can be described as follows:

Problem Analysis

The basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and the users, what exactly is desired from the software, and what the constraints on the solution are. Frequently the client and the users do not understand or know all their needs, because the potential of the new system is often not fully appreciated. The analysts have to collect and organize information about the client's organization and its processes, as well as help the clients and users identify their requirements.
The basic principle used in analysis is the same as in any complex task: divide and conquer. That is, partition the problem into subproblems and then try to understand each subproblem and its relationship to other subproblems in an effort to understand the total problem. There are a few methods for problem analysis, some of them are explained below.

Informal Approach

The informal approach to analysis is one where no defined methodology is used. With this approach, the analyst will have a series of meetings with the clients and end users. In the early meetings, the clients and end users will explain to the analyst about their work, their environment, and their needs as they perceive them. Any documents describing the work or the organization may be given, along with outputs of the existing methods of performing the tasks. In these early meetings, the analyst is basically the listener,

absorbing the information provided. Once the analyst understands the system to some extent, he uses the next few meetings to seek clarifications of the parts he does not understand. He may document the information in some manner (he may even build a model if he wishes), and he may do some brainstorming or thinking about what the system should do. In the final few meetings, the analyst essentially explains to the client what he understands the system should do and uses the meetings as a means of verifying if what he proposes the system should do is indeed consistent with the objectives of the clients. An initial draft of the SRS may be used in the final meetings.

Structured Analysis

This technique uses function-based decomposition while modeling the problem. It focuses on the functions performed in the problem domain and the data consumed and produced by these functions. The structured analysis method helps the analyst decide what type of information to obtain at different points in analysis, and it helps organize information so that the analyst is not overwhelmed by the complexity of the problem. It is a top-down refinement approach, which was originally called structured analysis and specification and was proposed for producing the specifications. The analysis model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. To accomplish these objectives, the analysis model derived during structured analysis takes the form illustrated below:

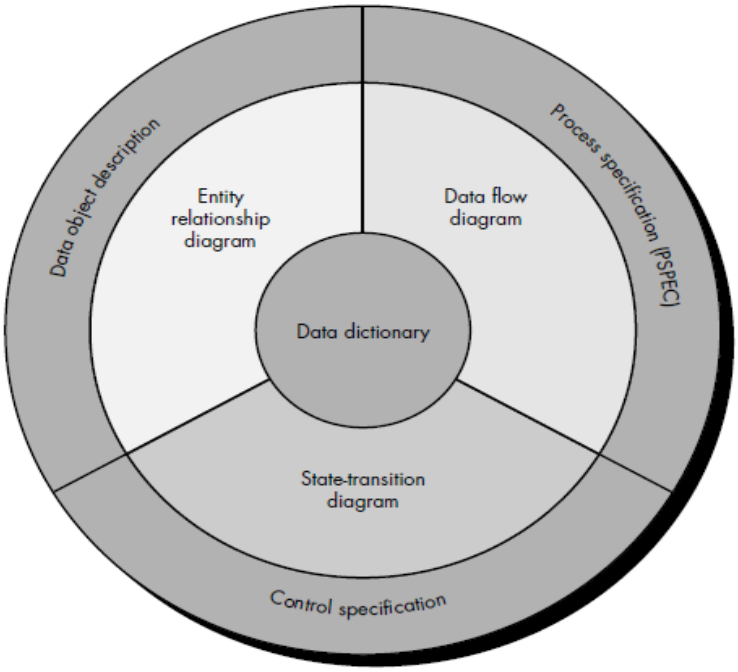


FIGURE
The structure of the analysis model

At the core of the model lies the data dictionary—a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the core. The entity relation diagram (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description. The data flow diagram (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and sub functions) that transform the data flow. The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a process specification (PSPEC). The state transition diagram (STD) indicates how the system behaves as a consequence of external events. To

accomplish this, the STD represents the various modes of behavior (called states) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about the control aspects of the software is contained in the control specification (CSPEC). The analysis model encompasses each of the diagrams, specifications, descriptions, and the dictionary

ER Diagrams

ERD, described in detail later in this section, enables a software engineer to identify data objects and their relationships using a graphical notation. In the context of structured analysis, the ERD defines all data that are entered, stored, transformed, and produced within an application. The entity relationship diagram focuses solely on data (and therefore satisfies the first operational analysis principles), representing a "data network" that exists for a given system. The ERD is especially useful for applications in which data and the relationships that govern data are complex.

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

Data Objects: A data object is a representation of almost any composite information that must be understood by software. By composite information, we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object. A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes. Data objects (represented in bold) are related to one another. For example, person can own car, where the relationship own connotes a specific "connection" between person and car. The relationships are always defined by the context of the problem that is being analyzed.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. For example, a Chevy Corvette is an instance of the data object car.

Attributes: Attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for car might serve well for an application that would be used by a Department of Motor Vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for car might also include ID number, body type and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

Relationships: Data objects are connected to one another in different ways. Consider two data objects, book and bookstore. A connection is established between book and bookstore because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of books and bookstores within the context of the software to be built. We can define a set of object/relationship pairs that define the relevant relationships. For example,

- A bookstore orders books.
- A bookstore displays books.
- A bookstore stocks books.

• A bookstore sells books.
• A bookstore returns books.

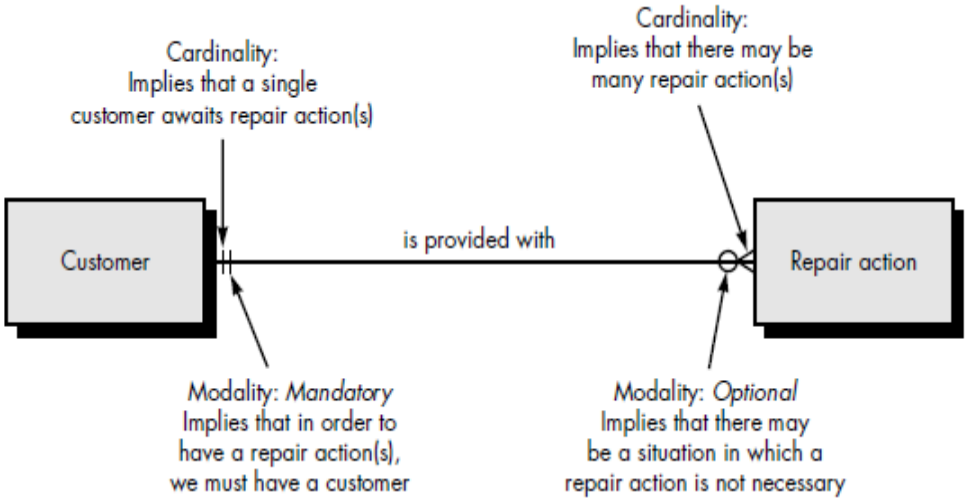
The relationships orders, displays, stocks, sells, and returns define the relevant connections between book and bookstore. It is important to note that object/relationship pairs are bidirectional. That is, they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.

Cardinality: The data model must be capable of representing the number of occurrences of objects in a given relationship. Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object]. Cardinality is usually expressed as simply 'one' or 'many.' For example, an employee can have one boss to report to, while a parent can have many children. Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as:

- *One-to-one (l:l)*—An occurrence of [object] 'A' can relate to one and only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'
- *One-to-many (l:N)*—One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.' For example, a mother can have many children, but a child can have only one mother.
- *Many-to-many (M:N)*—An occurrence of [object] 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.' For example, an uncle can have many nephews, while a nephew can have many uncles.

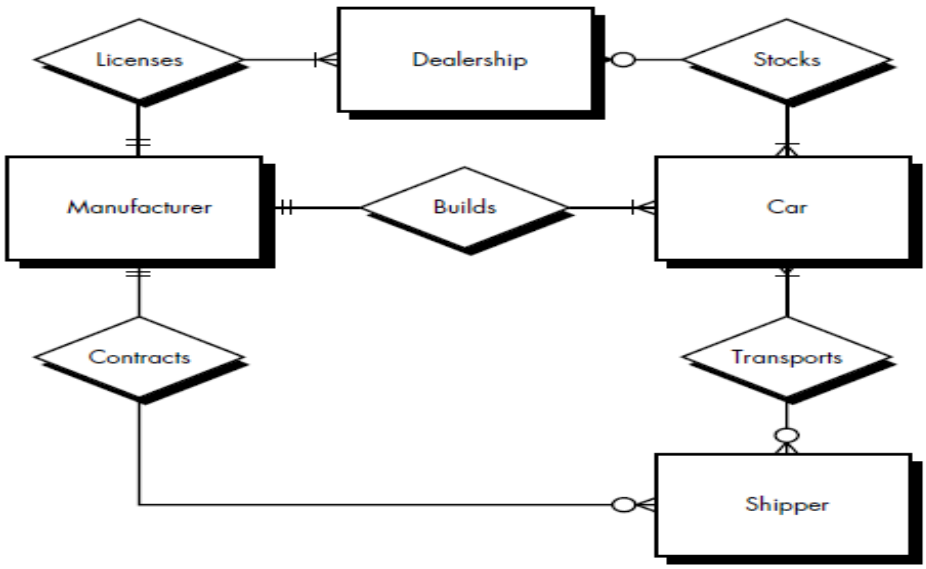
Cardinality is the maximum number of objects that can participate in a relationship. It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

Modality: The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. Figure below illustrates the relationship, cardinality, and modality between the data objects customer and repair action.

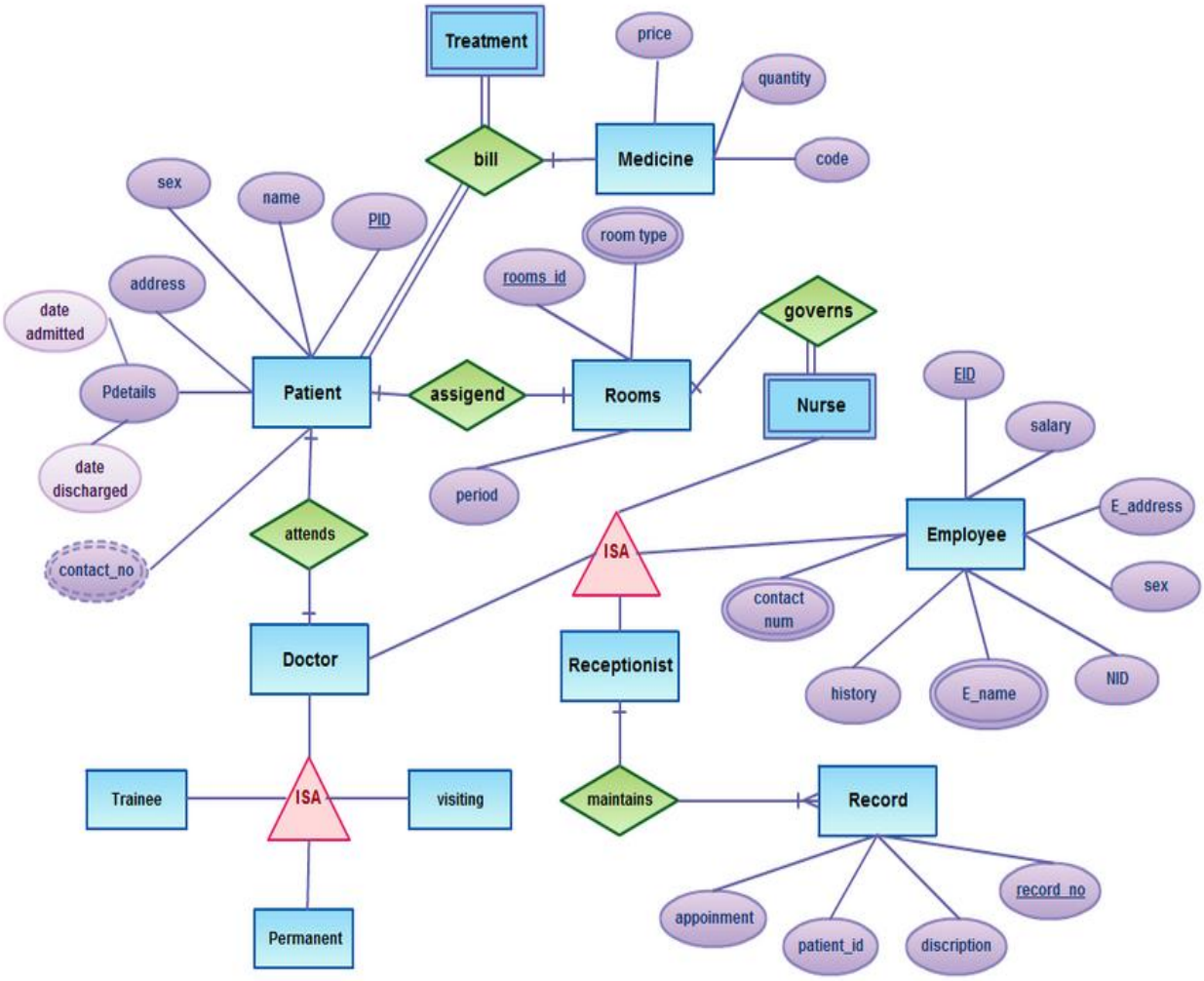


A set of primary components identified for the ERD are: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality.

Sample ER diagrams are shown below:



E-R Diagram for Hospital Management System



For more e.g. on crow's foot notation, modality and cardinality:
http://www.philblock.info/hitkb/i/interpreting_entity-relationship_diagrams.html
http://www2.cs.uregina.ca/~bernatja/crowsfoot.html

Data Flow Modeling

Data-flow based modeling, often referred to as the structured analysis technique, uses function-based decomposition while modeling the problem. It focuses on the functions performed in the problem domain and the data consumed and produced by these functions. It is a top-down refinement approach, which was originally called structured analysis and specification, and was proposed for producing the specifications. Below are described the data flow diagram and data dictionary on which the technique relies heavily.

Data Flow Diagrams and Data Dictionary

Data flow diagrams (also called data flow graphs) are commonly used during problem analysis. Data flow diagrams (DFDs) are quite general and are not limited to problem analysis for software requirements specification. They were in use long before the software engineering discipline began. DFDs are very useful in understanding a system and can be effectively used during analysis.

A DFD shows the flow of data through a system. It views a system as a function that transforms the inputs into desired outputs. Any complex system will not perform this transformation in a "single step," and a data will typically undergo a series of transformations before it becomes the output. The DFD aims to capture the transformations that take place within a system to the input data so that eventually the output data is produced. The agent that performs the transformation of data from one state to another is called a process (or a bubble). So, a DFD shows the movement of data through the different transformations or processes in the system. The processes are shown by named circles and data flows are represented by named arrows entering or leaving the bubbles. A rectangle represents a source or sink and is a net originator or consumer of data. A source or a sink is typically outside the main system of study. An example of a DFD for a system that pays workers is shown in the figure below. In this DFD there is one basic input data flow, the weekly timesheet, which originates from the source worker. The basic output is the paycheck, the sink for which is also the worker. In this system, first the employee's record is retrieved, using the employee ID, which is contained in the timesheet. From the employee record, the rate of payment and overtime are obtained. These rates and the regular and overtime hours (from the timesheet) are used to compute the pay. After the total pay is determined, taxes are deducted. To compute the tax deduction, information from the tax-rate file is used. The amount of tax deducted is recorded in the employee and company records. Finally, the paycheck is issued for the net pay. The amount paid is also recorded in company records. Some conventions used in drawing this DFD should be explained. All external files such as employee record, company record, and tax rates are shown as a labeled straight line. The need for multiple data flows by a process is represented by a '*' between the data flows. This symbol represents the AND relationship. For example, if there is a '*' between the two input data flows A and B for a process, it means that A AND B are needed for the process. In the DFD, for the process "weekly pay" the data flow "hours" and "pay rate" both are needed, as shown in the DFD. Similarly, the OR relationship is represented by a ' + ' between the data flows. This DFD is an abstract description of the system for handling payment. It does not matter if the system is automated or manual. This diagram could very well be for a manual system where the computations are all done with calculators, and the records are physical folders and ledgers. The details and minor data paths are not represented in this DFD. This is done to avoid getting bogged down with details while constructing a DFD for the overall system. If more details are desired, the DFD can be further refined.

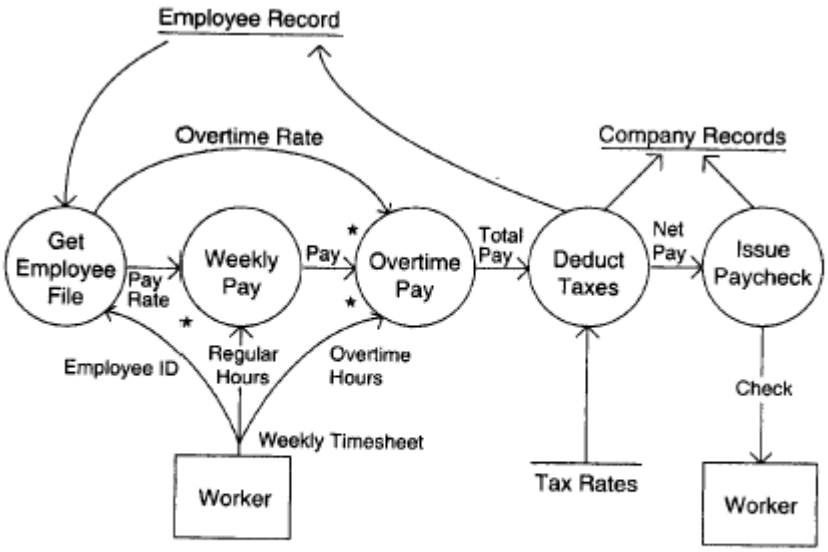


Figure . DFD of a system that pays workers.

Data flow diagrams (DFDs) are categorized as either logical or physical. A logical DFD focuses on the business and how the business operates. It describes the business events that take place and the data required and produced by each event. On the other hand, a physical DFD shows how the system will be implemented as we mentioned before. The chart below contrasts the features of logical and physical models. Notice that the logical model reflects the business, while the physical model depicts the system.

Design Feature	Logical	Physical
What the model depicts	How the business operates	How hue system will be implemented (or how the current system operates)
What the processes represent	Business activities	Programs, program modules and manual procedures
What the data stores represent	Collections of data, regardless of how the data is stored	Physical files and databases, manual files
Type of data stores	Show data stores representing permanent data collections	Master files, transaction files. Any processes that operate at two different times must be connected by a data store
System controls	Show business controls	Show controls for validating input data, for obtaining a record (record found status), for ensuring successful completion of a process and for system security (example: journal records)

Figure : Features common of logical and physical data flow diagrams.

Ideally, systems are developed by analyzing the current system (the current logical DFD), then adding features that the new system should include (the proposed logical DFD). Finally the best methods to implement the new system should be developed (the physical DFD). After the logical model for the new system has been developed, it may be used to create a physical data flow diagram for the new system. The progression of these models is illustrated below:

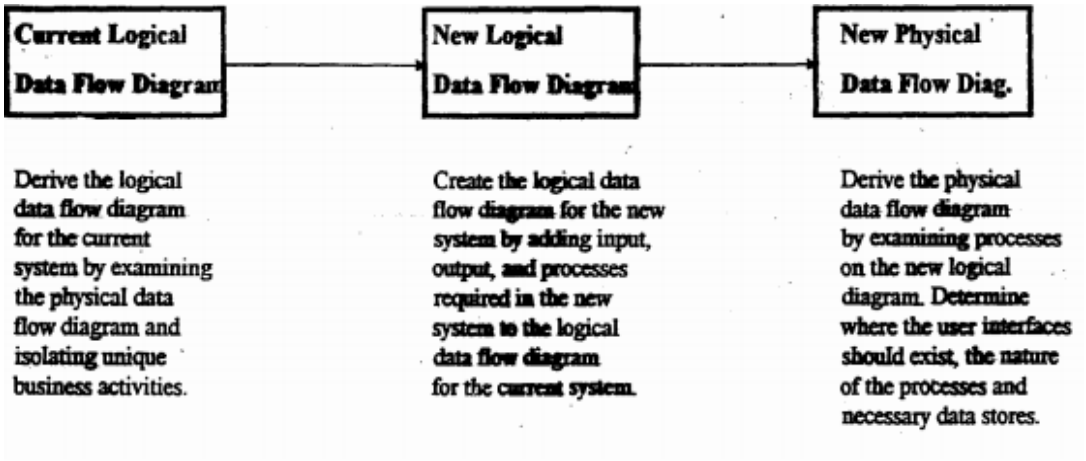
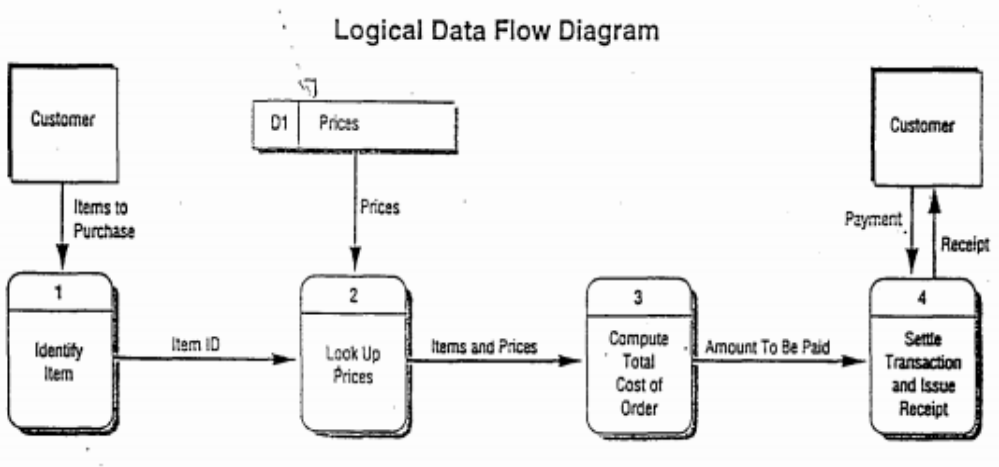
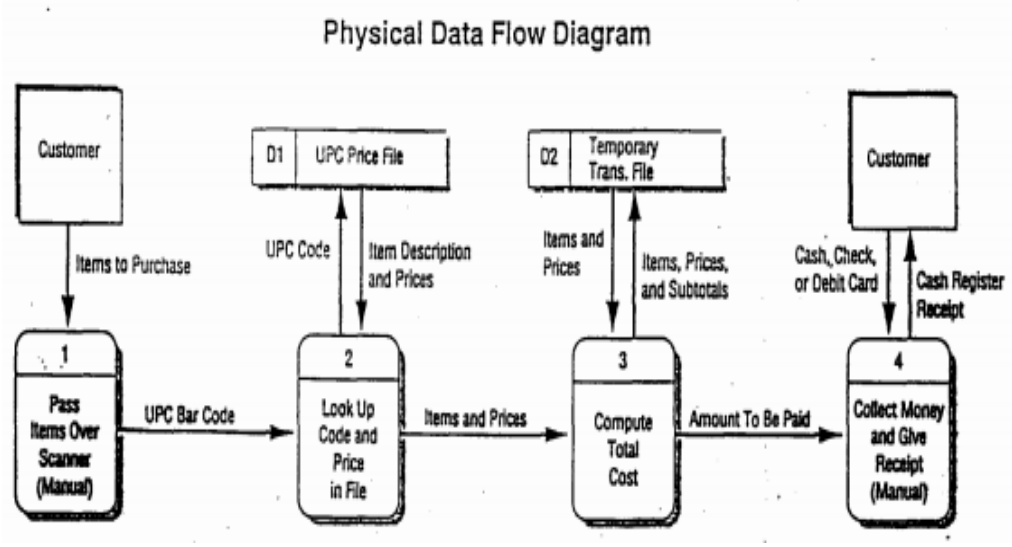


Figure : The progression of models from logical to physical.

Below is shown a logical DFD and a physical DFD for a grocery store cashier. The CUSTOMER brings the ITEMS to the register; PRICES for all ITEMS are LOOKED UP, and then totaled; next, PAYMENT is given to the cashier finally, the CUSTOMER is given a receipt. The logical DFD illustrates the processes involved without going into detail about the physical implementation of activities. The physical DFD shows that a bar code-the UPC PRICE code found on most grocery store items- is used. In addition, the physical DFD mentions manual processes such as scanning, explains that a temporary file is used to keep a subtotal of items, and indicates that the PAYMENT could be made by CASH, CHECK, or DEBIT CARD. Finally, it refers to the receipt by its name, CASH REGISTER RECEIPT.





It should be pointed out that a DFD is *not* a flowchart. A DFD represents the flow of data, while a flowchart shows the flow of control. A DFD does not represent procedural information. So, while drawing a DFD, one *must not* get involved in procedural details, and procedural thinking must be consciously avoided. For example, considerations of loops and decisions must be ignored. In drawing the DFD, the designer has to specify the major transforms in the path of the data flowing from the input to output. *How* those transforms are performed is *not* an issue while drawing the data flow graph. One way to construct a DFD is to start by identifying the major inputs and outputs. Minor inputs and outputs (like error messages) should be ignored at first. Then starting from the inputs, work toward the outputs, identifying the major transforms in the way. Start with a high-level data flow graph with few major transforms describing the entire transformation from the inputs to outputs and then refine each transform with more detailed transformations. Never try to show control logic. Label each arrow with proper data elements. Inputs and outputs of each transform should be carefully identified. Make use of * and + operations and show sufficient detail in the data flow graph. Try drawing alternate data flow graphs before settling on one.

Many systems are too large for a single DFD to describe the data processing clearly. It is necessary that some decomposition and abstraction mechanism be used for such systems. DFDs can be hierarchically organized, which helps in progressively partitioning and analyzing large systems. Such DFDs together are called a leveled DFD set. A leveled DFD set has a starting DFD, which is a very abstract representation of the system, identifying the major inputs and outputs and the major processes in the system. Then each process is refined and a DFD is drawn for the process. In other words, a bubble in a DFD is expanded into a DFD during refinement. For the hierarchy to be consistent, it is important that the net inputs and outputs of a DFD for a process are the same as the inputs and outputs of the process in the higher-level DFD. This refinement stops if each bubble is considered to be "atomic," in that each bubble can be easily specified or understood. It should be pointed out that during refinement, though the net input and output are preserved, a refinement of the data might also occur. That is, a unit of data may be broken into its components for processing when the detailed DFD for a process is being drawn. So, as the processes are decomposed, data decomposition also occurs. The precise structure of data flows is not specified in a DFD. The data dictionary is a repository of various data flows defined in a DFD. The associated data dictionary states precisely the structure of each data flow in the DFD. Components in the structure of a data flow may also be specified in the data dictionary, as well as the structure of files shown in the DFD. To define the data structure, different notations are used. Besides sequence or composition (represented by '+') selection and iteration are included. Selection (represented by vertical bar '|') means one OR the other, and repetition (represented by '*') means one or more occurrences. In the DFD shown earlier, data flows for weekly

timesheet are used. The data dictionary for this DFD is shown below. Most of the data flows in the DFD are specified here. The data dictionary entry for weekly timesheet specifies that this data flow is composed of three basic data entities—the employee name, employee ID, and many occurrences of the two-tuple consisting of regular hours and overtime hours. The last entity represents the daily working hours of the worker. The data dictionary also contains entries for specifying the different elements of a data flow.

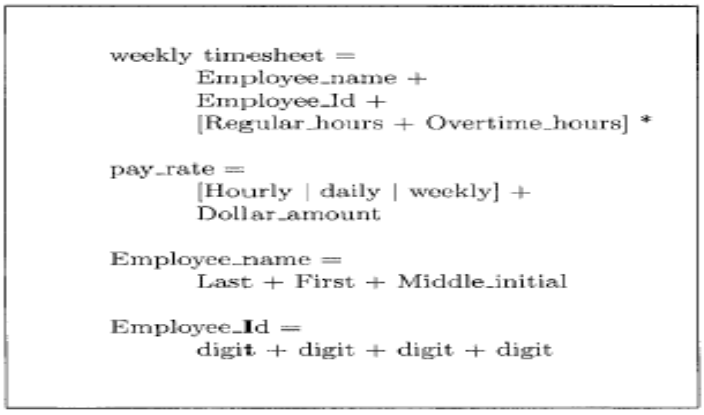


Figure : Data dictionary.

Once we have constructed a DFD and its associated data dictionary, we have to somehow verify that they are "correct." In addition to the walkthrough with the client, the analyst should look for common errors. Some common errors are:

- Unlabeled data flows
- Missing data flows; information required by a process is not available
- Extraneous data flows; some information is not being used in the process
- Consistency not maintained during refinement
- Missing processes
- Contains some control information

The Structured Analysis Method

The basic system view of this approach is that each system can be viewed as a transformation function operating within an environment that takes some inputs from the environment and produces some outputs for the environment. And as the overall transformation function of the entire system may be too complex to comprehend as a single function, the function should be partitioned into sub functions that together form the overall function. The sub-functions can be further partitioned and the process repeated until we reach a stage where each function can be comprehended easily. In any complex system the data transformation from the input to the output will not occur in a single step; rather the data will be transformed from the input to the output in a series of transformations starting from the input and culminating in the desired output. By tracking as the data flows through the system, the various functions being performed by a system can be identified. As this approach can be modeled easily by data flow diagrams, DFDs are used heavily in this method. The first step in this method is to study the "physical environment." During this, a DFD of the current non-automated (or partially automated) system is drawn, showing the input and output data flows of the system, how the data flows through the system, and what processes are operating on the data. While drawing the DFD for the physical environment, an analyst has to interact with the users to determine the overall process from the point of view of the data. This step is considered complete when the entire physical data flow diagram has been described and the user has accepted it as a true representation of the operation of the current system. The step may start with a *context diagram* in which the entire system is treated as a single process and all its inputs, outputs, sinks,

and sources are identified and shown. The basic purpose of analyzing the current system is to obtain a logical DFD for the system, where each data flow and each process is a logical entity or operation, rather than an actual name. Drawing a DFD for the physical system is only to provide a reasonable starting point for drawing the logical DFD. Hence, the next step in the analysis is to draw the logical equivalents of the DFD for the physical system. During this step, the DFD of the physical environment is taken and all specific physical data flows are represented by their logical equivalents.

For drawing a DFD, a top-down approach is suggested in the structured analysis method. In the structured analysis method, a DFD is constructed from scratch when the DFD for the physical system is being drawn when the DFD for the new system is being drawn. The second step largely performs transformations on the physical DFD. Drawing a DFD starts with a top-level DFD called the context diagram, which lists all the major inputs and outputs for the system. This diagram is then refined into a description of the different parts of the DFD showing more details. This results in a leveled set of DFDs. As pointed out earlier, during this refinement, the analyst has to make sure consistency is maintained and that net input and output are preserved during refinement. Clearly, the structured analysis provides methods for organizing and representing information about systems. It also provides guidelines for checking the accuracy of the information. Hence, for understanding and analyzing an existing system, this method provides useful tools.

More examples:

A restaurant owner feels that some amount of automation will help make her business more efficient. We must draw a DFD that models the new system to be built. After many meetings and discussions with the restaurant owner, the following goals for the new system were established:

- Automate much of the order processing and billing.
- Automate accounting.
- Make supply ordering more accurate so that leftovers at the end of the day are minimized and the orders that cannot be satisfied due to non availability are also minimized. This was being done without a careful analysis of sales.
- The owner also suspects that the staff might be stealing/eating some food/supplies. She wants the new system to help detect and reduce this.
- The owner would also like to have statistics about sales of different items.

The context diagram for the restaurant is shown below. The DFD for the new system is also given. Note that although taking orders might remain manual in the new system, the process might change, because the waiter might need to fill in codes for menu items. That is why it is also within the boundary of change. The DFD is largely self-explanatory. The major files in the system are:

Supplies file, Accounting file, Orders file, and the Menu. Some new processes that did not have equivalents earlier have been included in the system. These are "check for discrepancy," "accounting reports," and "statistics." Note that the processes are consistent in that the inputs given to them are sufficient to produce the outputs. For example, "checking for discrepancy" requires the following information to produce the report: total supplies received (obtained from the supplies file), supplies left at the end of the day, total orders placed by the customers (from the orders file), and the consumption rate for each menu item (from the menu). All these are shown as inputs to the process.

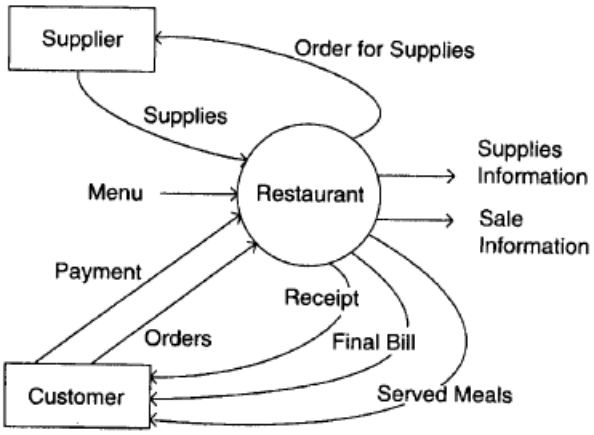


Figure : Context diagram for the restaurant.

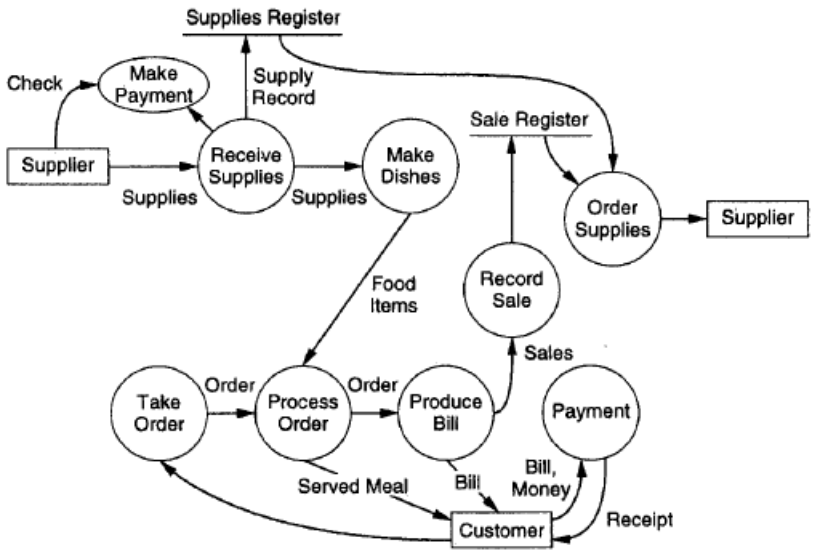


Figure : DFD for the existing restaurant system.

Supplies required for the next day are assessed from the total orders placed in the day and the orders that could not be satisfied due to lack of supplies (both kept in the order file). To see clearly if the information is sufficient for the different processes, the structure and exact contents of each of the data flows has to be specified. The data dictionary for this is given. The definitions of the different data flows and files are self-explanatory. Once this DFD and the data dictionary have been approved by the restaurant owner, the activity of understanding the problem is complete. After talking with the restaurant owner the man-machine boundary was also de fined (it is shown in the DFD). Now, such tasks as determining the detailed requirements of each of the bubbles shown in the DFD, and determining the nonfunctional requirements, deciding codes for the items in the menu and in the supply list remain. Further refinement for some of the bubbles might be needed. For example, it has to be determined what sort of accounting

reports or statistics are needed and what their formats should be. Once these are done, the analysis is complete and the requirements can then be compiled in a requirements specification document.

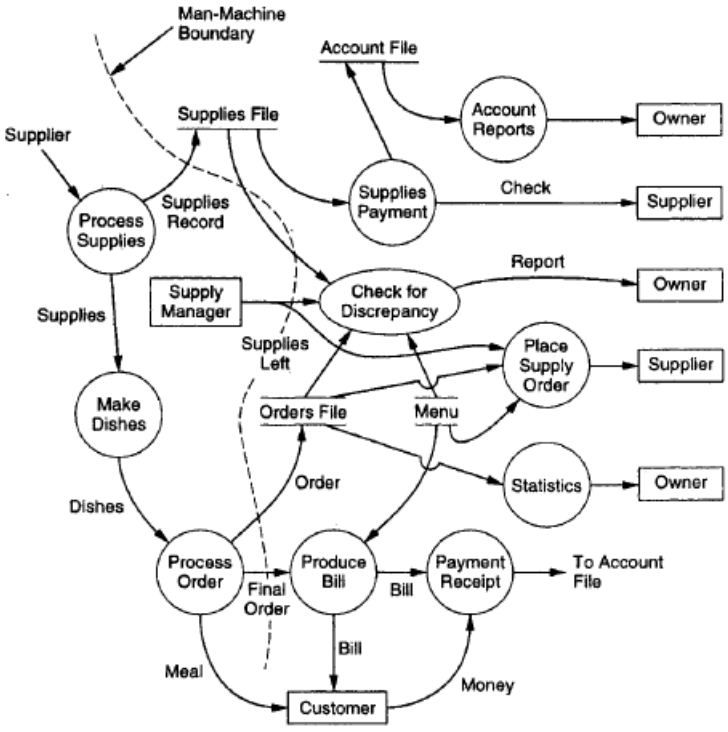


Figure : The DFD for the new restaurant system.

```
Supplies_file = [date + [item_no + quantity + cost]* ]*
Orders_file = [date + [menu_item_no + quantity + status]* ]*
status = satisfied | unsatisfied
order = [menu_item_no + quantity]*
menu = [menu_item_no + name + price + supplies_used]*
supplies_used = [supply_item_no + quantity]*
bill = [name + quantity + price]* +
       total_price + sales_tax + service_charge + grand_total
discrepancy_report = [supply_item_no +
       amt_ordered + amt_left + amt_consumed + descr]*
```

Figure : Data dictionary for the restaurant.

Object-Oriented Modeling

In object-oriented modeling, a system is viewed as a set of objects. The objects interact with each other through the services they provide. Some objects also interact with the users through their services such that the users get the desired services. Hence, the goal of modeling is to identify the objects (actually the object classes) that exist in the problem domain, define the classes by specifying what state information

they encapsulate and what services they provide, and identify relationships that exist between objects of different classes, such that the overall model is such that it supports the desired user services.
Object-oriented modeling and systems have been getting a lot of attention in the recent past. The basic reason for this is the belief that, object oriented systems are going to be easier to build and maintain. It is also believed that transitioning from object-oriented analysis to object-oriented design (and implementation) will be easy, and that object-oriented analysis is more immune to change because objects are more stable than functions. That is, in a problem domain, objects are likely to stay the same even if the exact nature of the problem changes, while this is not the case with function-oriented modeling.

Basic Concepts and Notation

In understanding or modeling a system using an object-oriented modeling technique, the system is viewed as consisting of objects. Each object has certain attributes, which together define the object. Separation of an object from its attributes is a natural method that we use for understanding systems (a man is separate from his attributes of height, weight, etc.). In object oriented systems, attributes hold the state (or define the state) of an object. An attribute is a pure data value (like integer, string, etc.), not an object. Objects of similar type are grouped together to form an object class (or just class). A class is essentially a type definition, which defines the state space of objects of its type and the operations (and their semantics) that can be applied to objects of that type. Formation of classes is also a general technique used by humans for understanding systems and differentiating between classes (e.g., an apple tree is an instance of the class of trees, and the class of trees is different from the class of birds). An object also provides some services or operations. These services are the only means by which the state of the object can be modified or viewed from outside. For operating a service, a message is sent to the object for that service. In general, these services are defined for a class and are provided for each object of that class. Encapsulating services and attributes together in an object is one of the main features that distinguish an object-oriented modeling approach from data modeling approaches, like the ER diagrams. Class diagrams represent a structure of the problem graphically using a precise notation. In a class diagram, a class is represented as a portrait-style rectangle divided into three parts. The top part contains the name of the class. The middle part lists the attributes that objects of this class possess. And the third part lists the services provided by objects of this class to model relationship between classes, a few structures are used. The generalization-specialization structure can be used by a class to inherit all or some attributes and services of a general class and add more attributes and services. This structure is modeled in object-oriented modeling through inheritance. By using a general class and inheriting some of its attributes and services and adding more, one can create a class that is a specialized version of the general class. And many specialized classes can be created from a general class, giving us class hierarchies.

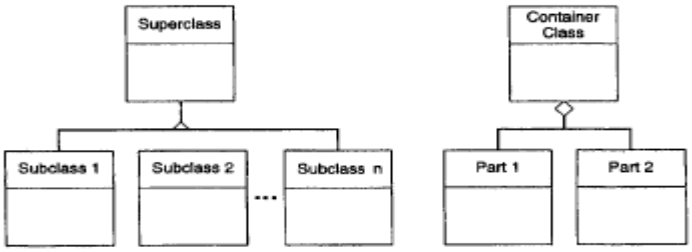


Figure : Class structures.

The *aggregation* structure models the whole-part relationship. An object may be composed of many objects; this is modeled through the aggregation structure. The representation of these in a class diagram is shown in Figure above. In addition to these, instances of a class may be related to objects of some other class. For example, an object of the class Employer may be related to many objects of the class

Employee. This relationship between objects also has to be captured if a system is to be modeled properly. This is captured through *associations.* An association is shown in the class diagram by having a line between the two classes. The multiplicity of an association specifies how many instances of one class may relate to instances of the other class through this association. An association between two classes can be one-to-one (i.e., one instance of one class is related to exactly one instance of the other class), one-to-many, or some other special cases. Multiplicity is specified by having a star (*) on the line adjacent to the class representing zero or more instances of the class may be related to an instance of the other class.
As example, suppose a system is being contemplated for a drugstore that will compute the total sales of the drugstore along with the total sales of different chemists that man the drugstore. The drugs are of two major types: off-the-shelf and prescription drugs. The system is to provide help in procuring drugs when out of stock, removing them when expired, replenishing the off-the-shelf drugs when needed, etc. A model of the system is shown in Figure below.

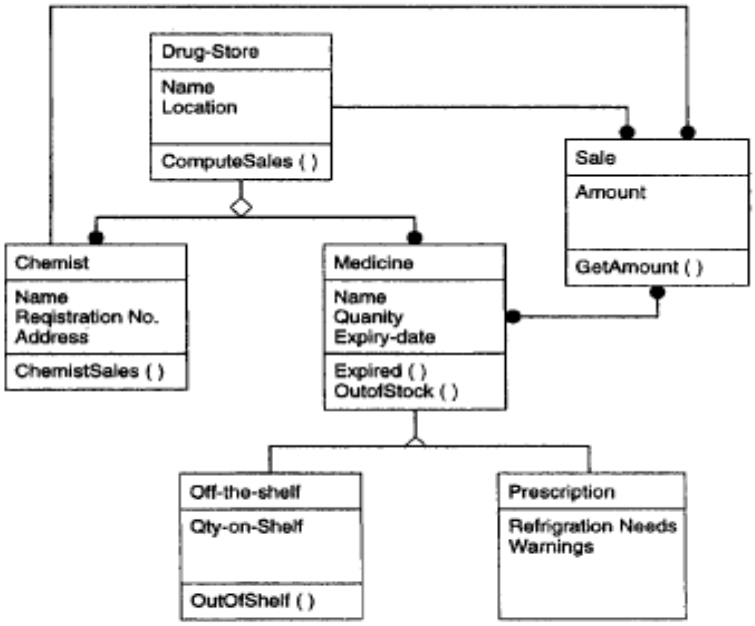


Figure : Model of a drugstore.

This class diagram has five classes of objects, each with a defined name, some attributes, and services. For example, an object of class Chemist has the attributes Name, Registration number, and Address. It has one service ChemistSales(), which computes the total sales by this chemist. The Drug-Store class is an aggregation of the class Medicine and the class Chemist (representing that a drugstore is composed of medicines and chemists). A Medicine may either be Off-the-shelf or Prescription. The class Medicine has some attributes like Name, Quantity in stock, and Expiry-date, and has services like Expired() (to list the expired medicines), OutOfStock() (to list medicines that are no longer in stock), etc. These attributes and services are inherited by the two specialized classes. In addition to these, the Off-the-shelf class has another attribute qty-on-shelf, representing how many have been put on the shelf and have services related to shelf stock. On the other hand, the Prescription class has Refrigeration-needs and Warnings as specialized attributes and services.

Performing Analysis

Now that we know what a model of a system consists of, the next question that arises is how to obtain the model for a system. In other words, how do we actually perform the analysis? The major steps in the analysis are:

- Identifying objects and classes
- Identifying structures
- Identifying attributes
- Identifying associations
- Defining services

Identifying Objects and Classes: An object during analysis is an encapsulation of attributes on which it provides some exclusive services. It represents something in the problem space. It has been argued that though things like interfaces between components, functions, etc. are generally volatile and change with changing needs, objects are quite stable in a problem domain.

To identify analysis objects, start by looking at the problem space and its description. Obtain a brief summary of the problem space. In the summary and other descriptions of the problem space, consider the nouns. Frequently, nouns represent entities in the problem space which will be modeled as objects. Structures, devices, events remembered, roles played, locations, organizational units, etc. are good candidates to consider. A candidate should be included as an object if the system needs to remember something about the object, the system needs some services from the object to perform its own services, and the object has multiple attributes (i.e., it is a high-level object encapsulating some attributes). If the system does not need to keep information about some real-world entity or does not need any services from the entity, it should not be considered as an object for modeling. Similarly, carefully consider objects that have only one attribute; such objects can frequently be included as attributes in other objects. Though the analysis focuses on identifying objects, in modeling, classes for these objects are represented.

Identifying Structure: Structures represent the hierarchies that exist between object classes. All complex systems have hierarchies. In object oriented modeling, the hierarchies are defined between classes that capture generalization-specialization and whole-part relationships. To identify the classification structure, consider the classes that have been identified as a generalization and see if there are other classes that can be considered as specializations of this. The specializations should be meaningful for the problem domain. For example, if the problem domain does not care about the material used to make some objects, there is no point in specializing the classes based on the material they are made of. Similarly, consider classes as specializations and see if there are other classes that have similar attributes. If so, see if a generalized class can be identified of which these are specializations. Once again, the structure obtained must naturally reflect the hierarchy in the problem domain; it should not be "extracted" simply because some classes have some attributes with the same names. To identify assembly structure, a similar approach is taken. Consider each object of a class as an assembly and identify its parts or components.

See if the system needs to keep track of the parts. If it does, then the parts must be reflected as objects; if not, then the parts should not be modeled as separate objects. Then, consider an object of a class as a part and see to which class's object it can be considered as belonging. Once again, this separation is maintained only if the system needs it. As before, the structures identified should naturally reflect the hierarchy in the problem domain and should not be "forced."

Identifying Attributes: Attributes add detail about the class and are the repositories of data for an object. For example, for an object of class Person, the attributes could be the name, sex, and address. The data stored in forms of values of attributes are hidden from outside the objects and are accessed and manipulated only by the service functions for that object. Which attributes should be used to define the class of an object depends on the problem and what needs to be done. For example, while modeling a hospital system, for the class Person attributes of height, weight, and date of birth may be needed, although these may not be needed for a database for a county that keeps track of populations in various neighborhoods. To identify attributes, consider each class and see which attributes are needed by the problem domain. This is frequently a simple task. Then position each attribute properly using the

structures; if the attribute is a common attribute, it should be placed in the superclass, while if it is specific to a specialized object it should be placed with the subclass. While identifying attributes, new classes may also get defined or old classes may disappear (e.g., if you find that a class really is an attribute of another).

Identifying Associations: Associations capture the relationship between instances of various classes. For example, an instance of the class Company may be related to an instance of the class Person by an "employs" relationship. This is similar to what is done in ER modeling. And like in ER modeling, an instance connection may be of 1:1 type representing that one instance of this type is related to exactly one instance of another class. Or it could be 1:M, indicating that one instance of this class may be related to many instances of the other class. There are M:M connections, and there are sometimes multi-way connections, but these are not very common. The associations between objects are derived from the problem domain directly once the objects have been identified. An association may have attributes of its own; these are typically attributes that do not naturally belong to either object. Although in many situations they can be "forced" to belong to one of the two objects without loosing any information, it should not be done unless the attribute naturally belongs to the object.

Defining Services: An object performs a set of predefined services. A service is performed when the object receives a message for it. Services really provide the active element in object-oriented modeling; they are the agent of state change or "processing." It is through the services that desired functional services can be provided by a system. To identify services, first identify the occur services, which are needed to create, destroy, and maintain the instances of the class. These services are generally not shown in the class diagrams. Other services depend on the type of services the system is providing. A method for identifying services is to define the system states and then in each state list the external events and required responses. For each of these, identify what services the different classes should possess. All the classes and their relationships are shown in a class diagram. The class diagram, clearly, gets large and complex for large systems. To handle the complexity, a *subject layer* in which the class model is partitioned into various subjects, with each subject containing some part of the diagram is suggested. Typically, a subject will contain many related classes.

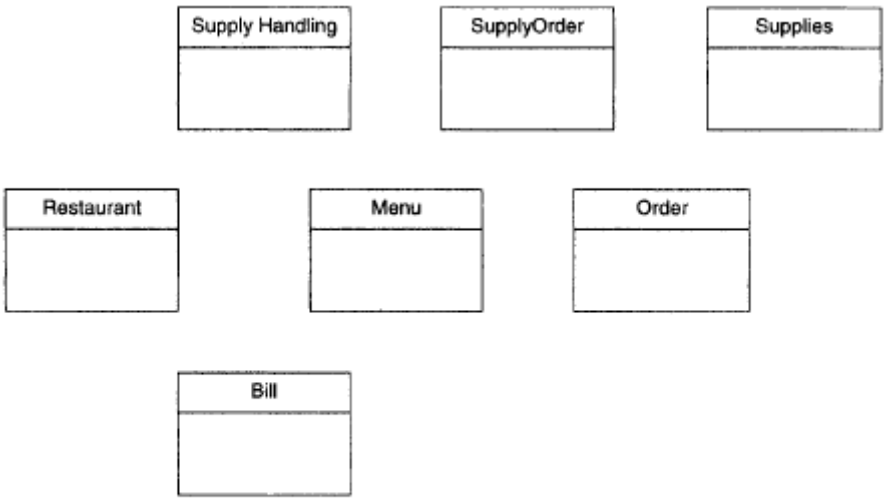


Figure : Initial classes in the restaurant example.

Example:

Consider the example of the restaurant, whose structured analysis was performed earlier. By stating the goals of the system (i.e., automate the bill generation for orders given by customers, obtain sale statistics,

determine discrepancy between supplies taken and supplies consumed, automate ordering of supplies) and studying the problem domain (i.e., the restaurant with customer, supplier, menu, etc.), we can clearly see that there are at least the following classes of objects: Restaurant, Restaurant owner, Bill, Menu, CustomerOrder, Supplier, SupplyOrder, Supply Handling, and Dishes. These are the initial classes.

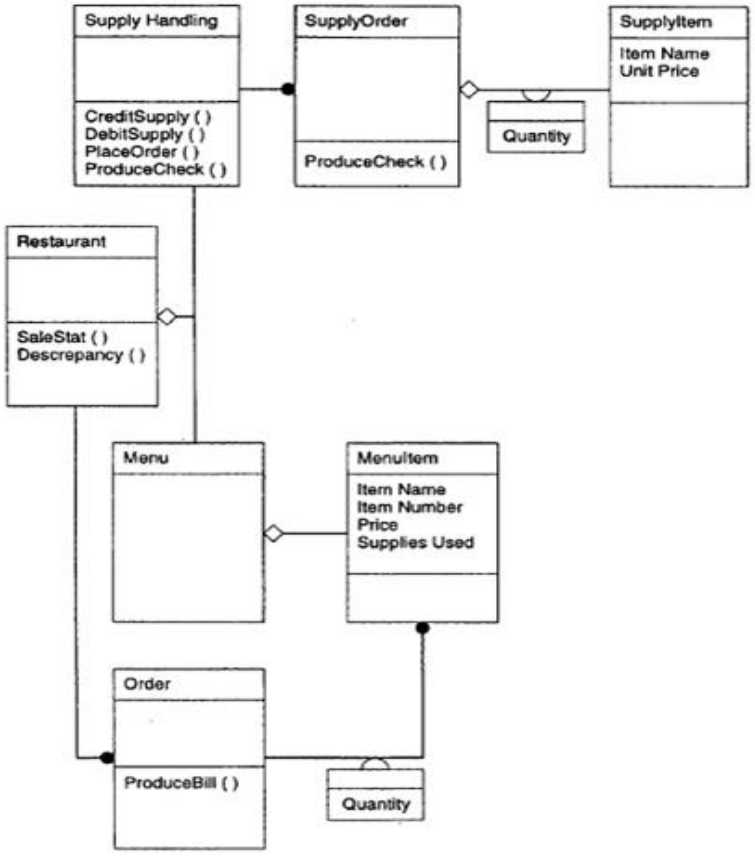


Figure : Class diagram for the restaurant.

By looking at the objectives and scope of the system, we find that no information about the Supplier or the Operator needs to be maintained in the system. So, they need not be modeled in the system as objects. For the same reason, entities like Dishes and Restaurant owner are not modeled as objects. The initial classes are shown above. This model will evolve and new classes may get added or some of these may not be needed. RestaurcLiit is an aggregation of Menu and Supply handling. A Menu is an aggregation of many Menultems. This requires us to add Menultem. Similarly, the SupplyOrder is an aggregation of (many) Supplyltems. This also requires us to add Supplyltem as a new class. The association between SupplyOrder and Supplyltem has an attribute quantity, reflecting the quantity ordered for a particular item by an order. There is no generalization-specialization hierarchy in this. Many attributes of various items can be directly identified. A Menultem has attributes of Number, Name, Price, Supplies used (i.e., which supplies it uses and quantity; this is needed to detect discrepancies in consumption and supplies used). Similarly, Supplyltem has Item name and Unit price as attributes. Next work is to identify relationships between objects. The Supply handling (unit) is related to (many) SupplyOrder. Similarly, an Order (by a customer) is related to many Menultem. Furthermore, this association has an attribute of its own—quantity. The quantity of the particular Menultem ordered in a particular Order is naturally a property of

the association between the specific Order and the specific MenuItem. Finally, we have to identify the services. Keeping our basic services of the system in mind (generate sales statistics, bill, discrepancy report, sale order), we define services of various classes. Supply handling object has the services CreditSupply() (used to record the receipt of supplies), DebitSupply() (used to record the supplies taken out), and PlaceOrder() (to place order of supplies). For SupplyOrder, one service is identified—ProduceCheck() to produce the check for the particular order. The object Order has one service—ProduceBill()—to produce the bill for the particular order. Handling the bill as essentially something generated for each order, we remove the object Bill from the object layer. The main object Restaurant has the services SaleStat() to generate the sale statistics for which it will require all order information (which it will obtain through its association with Order). It also has the service Discrepancy() to generate a discrepancy report. For this, it will need to find out what items have been consumed their quantity, and how much supply was debited. The former it can obtain from all the orders and the latter from Supply handling. The final class diagram is shown above.

Prototyping

In prototyping, a partial system is constructed, which is then used by the client, users, and developers to gain a better understanding of the problem and the needs. Hence, actual experience with a prototype that implements part of the eventual software system are used to analyze the problem and understand the requirements for the eventual software system. A software prototype can be defined as a partial implementation of a system whose purpose is to learn something about the problem being solved or the solution approach. As stated in this definition, prototyping can also be used to evaluate or check a design alternative (such a prototype is called a design prototype). Here we focus on prototyping used primarily for understanding the requirements. The rationale behind using prototyping for problem understanding and analysis is that the client and the users often find it difficult to visualize how the eventual software system will work in their environment just by reading a specification document. Visualizing the operation of the software that is yet to be built and whether it will satisfy the ultimate objectives, merely by reading and discussing the paper requirements, is indeed difficult. This is particularly true if the system is a totally new system and many users and clients do not have a good idea of their needs. The idea behind prototyping is that clients and the users can assess their needs much better if they can see the working of a system, even if the system is only a partial system. Prototyping emphasizes that actual practical experience is the best aid for understanding needs. By actually experimenting with a system, people can say, "I don't want this feature" or "I wish it had this feature" & so on. There are two approaches to prototyping; throwaway and evolutionary. In the throwaway approach the prototype is constructed with the idea that it will be discarded after the analysis is complete and the final system will be built from scratch. In the evolutionary approach, the prototype is built with the idea that it will eventually be converted into the final system. From the point of view of problem analysis and understanding, the throwaway prototypes are more suited. The following discussion is concentrated on throwaway prototypes. The first question that needs to be addressed is whether or not to prototype. It is important to clearly understand when prototyping should be done. The requirements of a system can be divided into three sets—^those that are well understood, those that are poorly understood, and those that are not known. In a throwaway prototype, the poorly understood requirements are the ones that should be incorporated. Based on the experience with the prototype, these requirements then become well understood. It might be possible to divide the set of poorly understood requirements further into two sets—those critical to design, and those not critical to design. The requirements that can be easily incorporated in the system later are considered noncritical to design. If which of the poorly understood requirements are critical and which are noncritical can be determined, then the throwaway prototype should focus mostly on the critical requirements. If the set of poorly understood requirements is substantial (in particular the subset of critical requirements), then a throwaway prototype should be built. The development activity starts with an SRS for the prototype. Developing the SRS for the prototype requires identifying the functions that should be included in the prototype. This decision is typically

application-dependent. In general, those requirements that tend to be unclear and vague, or where the clients and users are unsure or keep changing their mind, are the ones that should be implemented in the prototype. User interface, new features to be added (beyond automating what is currently being done), and features that may be infeasible, are common candidates for prototyping. Based on what aspects of the system are included in the prototype, the prototyping can be considered vertical or horizontal. In horizontal prototyping the system is viewed as being organized as a series of layers and some layer is the focus of prototyping. E.g., the user interface layer is frequently a good candidate for such prototyping, where most of the user interface is included in the prototype. In vertical prototyping, a chosen part of the system, which is not well understood, is built completely. This approach is used to validate some functionality or capability of the system.

Development of a throwaway prototype is fundamentally different from developing final production-quality software. The basic focus during prototyping is to keep costs low and minimize the prototype production time. Due to this, many of the bookkeeping, documenting, and quality control activities that are usually performed during software product development are kept to a minimum during prototyping. Efficiency concerns also take a back seat, and often very high-level interpretive languages are used for prototyping. For these reasons, temptation to convert the prototype into the final system should be resisted. Experience is gained by putting the system to use by the actual client and users. Constant interaction is needed with the client/users during this activity to understand their responses. Questionnaires and interviews might be used to gather user response. The final SRS is developed in much the same way as any SRS is developed. The difference here is that the client and users will be able to answer questions and explain their needs much better because of their experience with the prototype. Some initial analysis is also available. For prototyping for requirements analysis to be feasible, its cost must be kept low. Consequently, only those features that will have a valuable return from the user experience are included in the prototype. Exception handling, recovery, conformance to some standards and formats are typically not included in prototypes. Because the prototype is to be thrown away, only minimal development documents need to be produced during prototyping; for example, design documents, a test plan, and a test case specification are not needed during the development of the prototype. Another important cost-cutting measure is reduced testing. Testing consumes a major part of development expenditure during regular software development. By using cost-cutting methods, it is possible to keep the cost of the prototype to less than a few percent of the total development cost. The cost of developing and running a prototype can be around 10% of the total development cost. However, it should be pointed out that if the cost of prototyping is 10% of the total development cost, it does not mean that the cost of development has increased by this amount. The main reason is that the benefits obtained due to the use of prototype in terms of reduced requirement errors and reduced volume of requirement change requests are likely to be substantial, thereby reducing the cost of development itself.

Characteristics of an SRS

To properly satisfy the basic goals, an SRS should have certain properties and should contain different types of requirements. A good SRS is:

1) Correct
2) Complete
3) Unambiguous
4) Verifiable
5) Consistent
6) Ranked for importance and/or stability
7) Modifiable
8) Traceable

An SRS is *correct* if every requirement included in the SRS represents something required in the final system.

An SRS is *complete* if everything the software is supposed to do and the responses of the software to all classes of input data are specified in the SRS. Correctness and completeness go hand-in-hand; whereas correctness ensures that which is specified is done correctly, completeness ensures that everything is indeed specified. Correctness is an easier property to establish than completeness as it basically involves examining each requirement to make sure it represents the user requirement. Completeness, on the other hand, is the most difficult property to establish; to ensure completeness, one has to detect the absence of specifications, and absence is much harder to ascertain than determining that what is present has some property.

An SRS is *unambiguous* if and only if every requirement stated has one and only one interpretation. Requirements are often written in natural language, which are inherently ambiguous. If the requirements are specified in a natural language, the SRS writer has to be especially careful to ensure that there are no ambiguities. One way to avoid ambiguities is to use some formal requirements specification language. The major disadvantage of using formal languages is the large effort required to write an SRS, the high cost of doing so, and the increased difficulty reading and understanding formally stated requirements (particularly by the users and clients).

An SRS is *verifiable* if and only if every stated requirement is verifiable. A requirement is verifiable if there exists some cost-effective process that can check whether the final software meets that requirement. This implies that the requirements should have as little subjectivity as possible because subjective requirements are difficult to verify. Unambiguity is essential for verifiability.

As verification of requirements is often done through reviews, it also implies that an SRS is understandable, at least by the developer, the client, and the users. Understandability is clearly extremely important, as one of the goals of the requirements phase is to produce a document on which the client, the users, and the developers can agree.

An SRS is *consistent* if there is no requirement that conflicts with another. Terminology can cause inconsistencies; for example, different requirements may use different terms to refer to the same object. There may be logical or temporal conflict between requirements that causes inconsistencies. This occurs if the SRS contains two or more requirements whose logical or temporal characteristics cannot be satisfied together by any software system.

For example, suppose a requirement states that an event e is to occur before another event f. But then another set of requirements states (directly or indirectly by transitivity) that event f should occur before event e. Inconsistencies in an SRS can reflect of some major problems.

Generally, all the requirements for software are not of equal importance. Some are critical, others are important but not critical, and there are some which are desirable but not very important. Similarly, some requirements are "core" requirements which are not likely to change as time passes, while others are more dependent on time.

Writing an SRS is an iterative process. Even when the requirements of a system are specified, they are later modified as the needs of the client change. Hence an SRS should be easy to modify. An SRS is modifiable if its structure and style are such that any necessary change can be made easily while preserving completeness and consistency. Presence of redundancy is a major hindrance to modifiability, as it can easily lead to errors. For example, assume that a requirement is stated in two places and that the requirement later needs to be changed. If only one occurrence of the requirement is modified, the resulting SRS will be inconsistent.

An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development. Forward traceability means that each requirement should be traceable to some design and code elements. Backward traceability requires that it be possible to trace design and code elements to the requirements they support. Traceability aids verification and validation.

Of all these characteristics, completeness is perhaps the most important (and hardest to ensure). One of the most common problems in requirements specification is when some of the requirements of the client are not specified. This necessitates additions and modifications to the requirements later in the

development cycle, which are often expensive to incorporate. Incompleteness is also a major source of disagreement between the client and the supplier.

Components of an SRS

The basic issues/system properties that an SRS must address are:

a) Functionality or Functional Requirements: Functional requirements specify which outputs should be produced from the given inputs. They describe the relationship between the input and output of the system. For each functional requirement, a detailed description of all the data inputs and their source, the units of measure, and the range of valid inputs must be specified. All the operations to be performed on the input data to obtain the output should be specified. This includes specifying the validity checks on the input and output data, parameters affected by the operation, and equations or other logical operations that must be used to transform the inputs into corresponding outputs. For example, if there is a formula for computing the output, it should be specified. Care must be taken not to specify any algorithms that are not part of the system but that may be needed to implement the system. These decisions should be left for the designer. An important part of the specification is the system behavior in abnormal situations, like invalid input (which can occur in many ways) or error during computation. The functional requirement must clearly state what the system should do if such situations occur. Specifically, it should specify the behavior of the system for invalid inputs and invalid outputs. Furthermore, behavior for situations where the input is valid but the normal operation cannot be performed should also be specified. An example of this situation is an airline reservation system, where a reservation cannot be made even for valid passengers if the airplane is fully booked. In short, the system behavior for all foreseen inputs and all foreseen system states should be specified. These special conditions are often likely to be overlooked, resulting in a system that is not robust.

b) Performance or Performance Requirements: This part of an SRS specifies the performance constraints on the software system. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements: *static and dynamic*. *Static requirements* are those that do not impose constraint on the execution characteristics of the system. These include requirements like the number of terminals to be supported, the number of simultaneous users to be supported, and the number of files that the system has to process and their sizes. These are also called *capacity requirements* of the system. *Dynamic requirements* specify constraints on the execution behavior of the system. These typically include response time and throughput constraints on the system. Response time is the expected time for the completion of an operation under specified circumstances. Throughput is the expected number of operations that can be performed in a unit time. For example, the SRS may specify the number of transactions that must be processed per unit time, or what the response time for a particular command should be. Acceptable ranges of the different performance parameters should be specified, as well as acceptable performance for both normal and peak workload conditions. All of these requirements should be stated in measurable terms. Requirements such as "response time should be good" or the system must be able to "process all the transactions quickly" are not desirable because they are imprecise and not verifiable. Instead, statements like "the response time of command x should be less than one second 90% of the times" or "a transaction should be processed in less than one second 98% of the times" should be used to declare performance specifications.

c) Design constraints imposed on an implementation: There are a number of factors in the client's environment that may restrict the choices of a designer. Such factors include standards that must be followed, resource limits, operating environment, reliability and security requirements, and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

Standards Compliance: This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures. There may be audit tracing requirements, which require certain kinds of changes, or operations that must be recorded in an audit file.

Hardware Limitations: The software may have to operate on some existing or predetermined hardware, thus imposing restrictions on the design. Hardware limitations can include the type of machines to be used, operating system available on the system, languages supported, and limits on primary and secondary storage.

Reliability and Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex and expensive. Requirements about system behavior in the face of certain kinds of faults are specified. Recovery requirements are often an integral part here, detailing what the system should do if some failure occurs to ensure certain properties. Reliability requirements are very important for critical applications.

Security: Security requirements are particularly significant in defense systems and many database systems. Security requirements place restrictions on the use of certain commands, control access to data, provide different kinds of access requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system. Given the current security needs even of common systems, they may also require proper assessment of security threats, proper programming techniques, and use of tools to detect flaws like buffer overflow.

d) External interfaces or External Interface Requirements: All the interactions of the software with people, hardware, and other software should be clearly specified. For the user interface, the characteristics of each user interface of the software product should be specified. User interface is becoming increasingly important and must be given proper attention. A preliminary user manual should be created with all user commands, screen formats, an explanation of how the system will appear to the user, and feedback and error messages. Like other specifications these requirements should be precise and verifiable. So, a statement like "the system should be user friendly" should be avoided and statements like "commands should be no longer than six characters" or "command names should reflect the function they perform" used. For hardware interface requirements, the SRS should specify the logical characteristics of each interface between the software product and the hardware components. If the software is to execute on existing hardware or on predetermined hardware, all the characteristics of the hardware, including memory restrictions, should be specified. In addition, the current use and load characteristics of the hardware should be given. The interface requirement should specify the interface with other software the system will use or that will use the system. This includes the interface with the operating system and other applications. The message content and format of each interface should be specified.

Conceptually, any SRS should have these components. If the traditional approach to requirement analysis is being followed, then the SRS might even have portions corresponding to these. Functional requirements might be specified indirectly by specifying the services on the objects or by specifying the use cases.

Specification Language

Requirements specification necessitates the use of some specification language. The language should support the desired qualities of the SRS- modifiability, understandability, unambiguous, and so forth. In addition, the language should be easy to learn and use. Many of these characteristics conflict in the selection of a specification language. For example, to avoid ambiguity, it is best to use some formal language. But for ease of understanding a natural language might be preferable. Though formal notations exist for specifying specific properties of the system, natural languages are now most often used for

specifying requirements. If formal languages are to be used, they are often used to specify particular properties or for specific parts of the system, and these formal specifications are generally contained in the overall SRS, which is in a natural language. The overall SRS is generally in a natural language, and when feasible and desirable, some specifications in the SRS may use formal languages. The major advantage of using a natural language is that both client and supplier understand the language. However, by the very nature of a natural language, it is imprecise and ambiguous. To reduce the drawbacks of natural languages, most often natural language is used in a structured fashion. In structured English (for example), requirements are broken into sections and paragraphs. Each paragraph is then broken into subparagraphs. Many organizations also specify strict uses of some words like "shall," "perhaps," and "should" and try to restrict the use of common phrases in order to improve the precision and reduce the verbosity and ambiguity. A general rule when using a natural language is to be precise, factual, and brief, and organize the requirements hierarchically where possible, giving unique numbers to each separate requirement. In an SRS, some parts can be specified better using some formal notation. For example, to specify formats of inputs or outputs, regular expressions can be very useful. Similarly, when discussing systems like communication protocols, finite state automata can be used. Decision tables are useful to formally specify the behavior of a system on different combinations of inputs or settings. Similarly, some aspects of the system may be specified or explained using the models that may have been built during problem analysis. Sometimes models may be included as supporting documents that help clarify the requirements and the motivation better.

Structure of a Requirements Document / Software Requirements Specification (SRS)

The SRS is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.

The *Introduction* of the SRS states the goals and objectives of the software, describing it in the context of the computer-based system. It also contains the purpose, scope, overview, etc. of the requirements document. It also contains the references cited in the document and any definitions that are used. The general factors that affect the product and its requirements are mentioned next. Specific requirements are not provided, but a general overview is presented to make the understanding of the specific requirements easier. Product perspective is essentially the relationship of the product to other products; defining if the product is independent or is a part of a larger product, and what the principal interfaces of the product are. A general abstract description of the functions to be performed by the product is given. Schematic diagrams showing a general view of different functions and their relationships with each other can often be useful. Similarly, typical characteristics of the eventual end user and general constraints are also specified.

The specific requirements section of the SRS describes all the details that the software developer needs to know for designing and developing the system. This is typically the largest and most important part of the document.

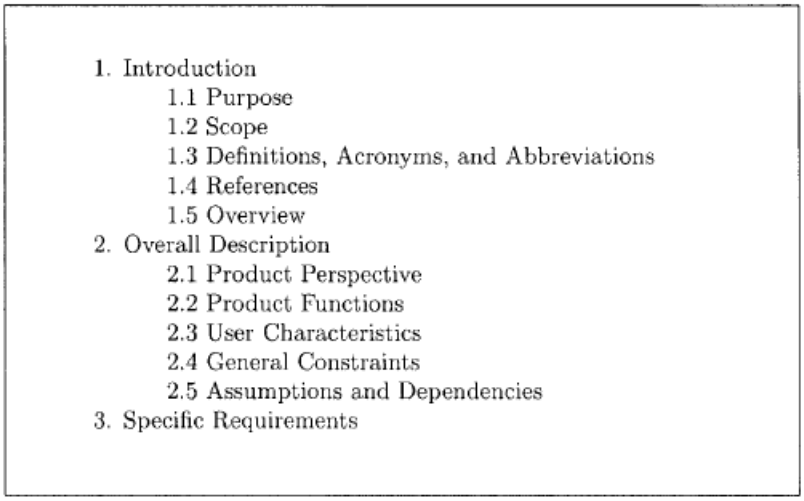


Figure .: General structure of an SRS.

In here the *external interface requirements* section specifies all the interfaces of the software: to people, other software, hardware, and other systems. User interfaces are clearly a very important component; they specify each human interface the system plans to have, including screen formats, contents of menus, and command structure. In hardware interfaces, the logical characteristics of each interface between the software and hardware on which the software can run are specified. Essentially, any assumptions the software is making about the hardware are listed here. In software interfaces, all other software that is needed for this software to run is specified, along with the interfaces. Communication interfaces need to be specified if the software communicates with other entities in other machines.

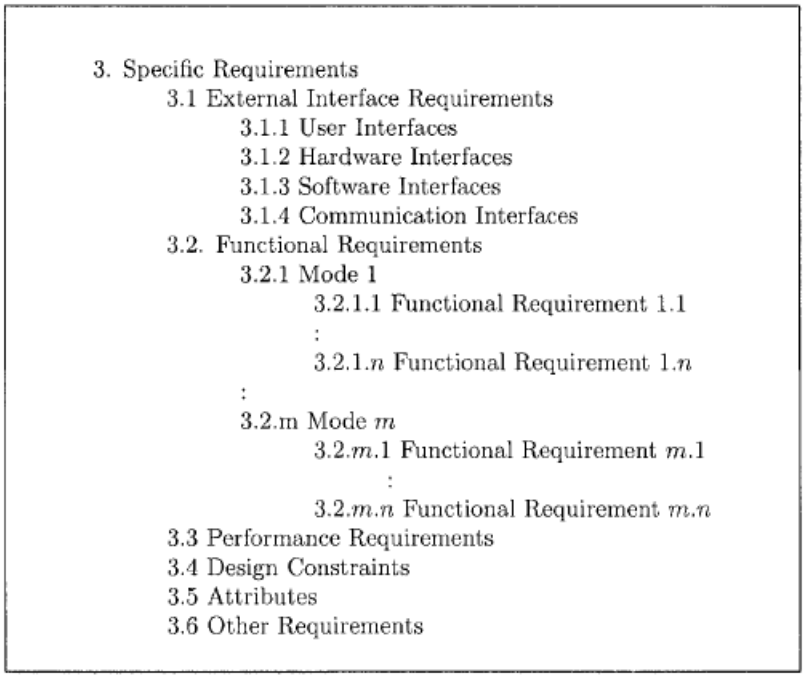


Figure 3.14: One organization for specific requirements.

In the *functional requirements section*, the functional capabilities of the system are described. In this organization, the functional capabilities for all the modes of operation of the software are given. For each

functional requirement, the required inputs, desired outputs, and processing requirements will have to be specified. For each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements. For the inputs, the source of the inputs, the units of measure, valid ranges, accuracies, etc. have to be specified. For specifying the processing, all operations that need to be performed on the input data and any intermediate data produced should be specified. This includes validity checks on inputs, sequence of operations, responses to abnormal situations, and methods that must be used in processing to transform the inputs into corresponding outputs. Note that no algorithms are generally specified, only the relationship between the inputs and the outputs (which may be in the form of an equation or a formula) so that an algorithm can be designed to produce the outputs from the inputs. For outputs, the destination of outputs, units of measure, range of valid outputs, error messages, etc. all have to be specified. The *performance section* should specify both static and dynamic performance requirements. All factors that constrain the system design are described in the performance constraints section. The *attributes section* specifies some of the overall attributes that the system should have. Any requirement not covered under these is listed under other requirements. *Design constraints* specify all the constraints imposed on design (e.g., security, fault tolerance, and standards compliance). Finally, the specification includes a Bibliography and Appendix. The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, graphs, and other material are presented as appendixes.

When use cases are employed, then the functional requirements section of the SRS is replaced/supplemented by use case descriptions. The product perspective part of the SRS may provide an overview or summary of the use cases. The key concern is that after the requirements have been identified, the requirements document should be organized in such a manner that it aids validation and system design.

Functional Specification with Use Cases

Functional requirements often form the core of a requirements document. The traditional approach for specifying functionality is to specify each function that the system should provide. Use cases specify the functionality of a system by specifying the behavior of the system, captured as interactions of the users with the system. Use cases can be used to describe the business processes of the larger business or organization that deploys the software, or it could just describe the behavior of the software system. Though use cases are primarily for specifying behavior, they can also be used effectively during analysis. So use cases can help in eliciting requirements also. They are generally viewed as part of an object oriented approach to software development though they are a general method for describing the interaction of a system (even non-IT systems.)

Use Case basics: Basics

Use cases are naturally textual descriptions, and represent the behavioral requirements of the system. This behavior specification can capture most of the functional requirements of the system. Therefore, use cases do not form the complete SRS, but can form a part of it. The complete SRS, as we have seen, will need to capture other requirements like performance and design constraints. Though the detailed use cases are textual, diagrams can be used to supplement the textual description. For example, the use case diagram of UML provides an overview of the use cases and actors in the system and their dependency. A UML use case diagram generally shows each use case in the system as an ellipse, shows the primary actor for the use case as a stick figure connected to the use case with a line, and shows dependency between use cases by arcs between use cases. Some other relationships between use cases can also be represented. However, as use cases are basically textual in nature, diagrams play a limited role in either developing or specifying use cases.

Term	Definition
Actors	A person or a system which uses the system being built for achieving some goal.
Primary actor	The main actor for whom a use case is initiated and whose goal satisfaction is the main objective of the use case.
Scenario	A set of actions that are performed to achieve a goal under some specified conditions.
Main success scenario	Describes the interaction if nothing fails and all steps in the scenario succeed.
Extension scenario	Describe the system behavior if some of the steps in the main scenario do not complete successfully.

Table : Use Case terms.

Let us consider that a small on-line auction system is to be built, in which different persons can sell and buy goods. We will assume that there is a separate financial subsystem through which the payments are made and that each buyer and seller has an account in it.

- ***Use Case 0***: **Auction an item**
  Primary Actor: Auction system
  Scope : Auction conducting organization
  Precondition: None
  Main Success Scenario:
  1. Seller performs *Put an item for auction*
  2. Various bidders perform *make a bid*
  3. On final date perform *Complete the auction of the item*
  4. Get feedback from seller; get feedback from buyer; update records

Figure : A summary level use case.

- ***Use Case 1*: Put an item up for auction**
  Primary Actor: Seller
  Precondition: Seller has logged in
  Main Success Scenario:

  1. Seller posts an item (its category, description, picture, etc.) for auction
  2. System shows past prices of similar items to seller
  3. Seller specifies the starting bid price and a date when auction will close
  4. System accepts the item and posts it

  Exception Scenarios:

  - 2 a) There are no past items of this category
    * System tells the seller this situation

- ***Use Case 2*: Make a bid**
  Primary Actor: Buyer
  Precondition: The buyer has logged in
  Main Success Scenario:

  1. Buyer searches or browses and selects some item
  2. System shows the rating of the seller, the starting bid, the current bids, and the highest bid; asks buyer to make a bid
  3. Buyer specifies a bid price, maximum bid price, and an increment
  4. System accepts the bid; Blocks funds in bidders account
  5. System updates the bid price of other bidders where needed, and updates the records for the item

  Exception Scenarios:

  - 3 a) The bid price is lower than the current highest
    * System informs the bidder and asks to rebid
  - 4 a) The bidder does not have enough funds in his account
    * System cancels the bid, asks the user to get more funds

Figure : Main use cases in an auction system.

- ***Use Case 3*: Complete auction of an item**
  Primary Actor: Auction System
  Precondition: The last date for bidding has been reached
  Main Success Scenario:

  1. Select highest bidder; send email to selected bidder and seller informing final bid price; send email to other bidders also.
  2. Debit bidder's account and credit seller's
  3. Transfer from seller's acct. commission amt. to organization's acct.
  4. Remove item from the site; update records

  Exception Scenarios: None

Figure : Main use cases in an auction system (contd.)

UCs where the scope is the enterprise can often run over a long period of time (e.g., process an application of a prospective candidate.) These use cases may require many different systems to perform different tasks before the UC can be completed. So, sometimes it may be useful to describe some of the key business processes as summary level use cases to provide the context for the system being designed and built. What should be the level of detail in a use case? The answer always depends on the project and the situation. So it is with use cases. Generally it is good to have sufficient details which are not overwhelming but are sufficient to build the system and meet its quality goals.

Validation of Requirements

The development of software starts with a requirements document, which is also used to determine eventually whether or not the delivered software system is acceptable. It is therefore important that the requirements specification contains no errors and specifies the client's requirements correctly. Also, the longer an error remains undetected, the greater the cost of correcting it. So, it is necessary to detect errors in the requirements before the design and development of the software begin. Due to the nature of the requirement specification phase, there is a lot of scope for misunderstanding and committing errors, and it is quite possible that the requirements specification does not accurately represent the client's needs. The basic objective of the requirements validation activity is to ensure that the SRS reflects the actual requirements accurately and clearly. A related objective is to check that the SRS document is itself of "good quality".

The type of errors that typically occur in an SRS can be classified in four types: *omission, inconsistency, incorrect fact, and ambiguity*. Omission is a common error in requirements. In this type of error, some user requirement is simply not included in the SRS; the omitted requirement may be related to the behavior of the system, its performance, constraints, or any other factor. Omission directly affects the external completeness of the SRS. Another common form of error in requirements is inconsistency. Inconsistency can be due to contradictions within the requirements themselves or to incompatibility of the stated requirements with the actual requirements of the client or with the environment in which the system will operate. The third common requirement error is incorrect fact. Errors of this type occur when some fact recorded in the SRS is not correct. The fourth common error type is ambiguity. Errors of this type occur when there are some requirements that have multiple meanings, that is, their interpretation is not unique. As requirements are generally textual documents that cannot be executed, inspections are eminently suitable for requirements validation. Consequently, inspections of the SRS, frequently called requirements review, are the most common method of validation. Because requirements specification formally specifies something that originally existed informally in people's minds, requirements validation must involve the clients and the users. Due to this, the requirements review team generally consists of client as well as user representatives. Requirements review is a review by a group of people to find errors

and point out other matters of concern in the requirements specifications of a system. The review group should include the author of the requirements document, someone who understands the needs of the client, a person of the design team, and the person(s) responsible for maintaining the requirements document. It is also good practice to include some people not directly involved with product development, like a software quality engineer. Although the primary goal of the review process is to reveal any errors in the requirements, such as those discussed earlier, the review process is also used to consider factors affecting quality, such as testability and readability. During the review, one of the jobs of the reviewers is to uncover the requirements that are too subjective and too difficult to define criteria for testing that requirement. Checklists are frequently used in reviews to focus the review effort and ensure that no major source of errors is overlooked by the reviewers. A checklist for requirements review should include items like:

- Are all hardware resources defined?
- Have the response times of functions been specified?
- Have all the hardware, external software, and data interfaces been defined?
- Have all the functions required by the client been specified?
- Is each requirement testable?
- Is the initial state of the system defined?
- Are the responses to exceptional conditions specified?
- Does the requirement contain restrictions that can be controlled by the designer?
- Are possible future modifications specified?

Requirements reviews are probably the most effective means for detecting requirement errors. Though requirements reviews remain the most commonly used and viable means for requirement validation, other ways can also be used, for example, if the requirements are written in a formal specification language or a language specifically designed for machine processing, then it is possible to have tools to verify some properties of requirements. These tools will focus on checks for internal consistency and completeness, which sometimes leads to checking of external completeness. However, these tools cannot directly check for external completeness (after all, how will a tool know that some requirement has been completely omitted?). For this reason, requirements reviews are needed even if the requirements are specified through a tool or are in a formal notation.
