3.3.1 Characteristics of an SRS

To properly satisfy the basic goals, an SRS should have certain properties and should contain different types of requirements. In this section, we discuss some of the desirable characteristics of an SRS and components of an SRS. A good SRS is [91, 92]:

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

The discussion of these properties here is based on [91, 92]. 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. An SRS is ranked for importance and/or stability if for each requirement the importance and the stability of the requirement are indicated. Stability of a requirement reflects the chances of it changing in future. It can be reflected in terms of the expected change volume.

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 [91]. 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 problem 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. The importance of having complete requirements cannot be overemphasized.

3.3.2 Components of an SRS

Completeness of specifications is difficult to achieve and even more difficult to verify. Having guidelines about what different things an SRS should specify will help in completely specifying the requirements. Here we describe some of the system properties that an SRS should specify. The basic issues an SRS must address are:

- Functionality
- Performance
- Design constraints imposed on an implementation
- External interfaces

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. However, functional requirements might be specified indirectly by specifying the services on the objects or by specifying the use cases.

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.

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.

Design Constraints

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 is 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.

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.

3.3.3 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. As one might expect, 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. In other words, 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, as discussed, 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.

3.3.4 Structure of a Requirements Document

All the requirements for the system have to be included in a document that is clear and concise. For this, it is necessary to organize the requirements document as sections and subsections. There can be many ways to structure a requirements document. Many methods and standards have been proposed for organizing an SRS. One of the main ideas of standardizing the structure of the document is that with an available standard, each SRS will fit a certain pattern, which will make it easier for others to understand (that is one of the roles of any standard). Another role these standards play is that by requiring various aspects to be specified, they help ensure that the analyst does not forget some major property. Here we discuss the organization proposed in the IEEE guide to software requirements specifications [92].

The IEEE standards recognize the fact that different projects may require their requirements to be organized differently, that is, there is no one method that is suitable for all projects. It provides different ways of structuring the SRS. The first two sections

of the SRS are the same in all of them. The general structure of an SRS is given in Figure 3.

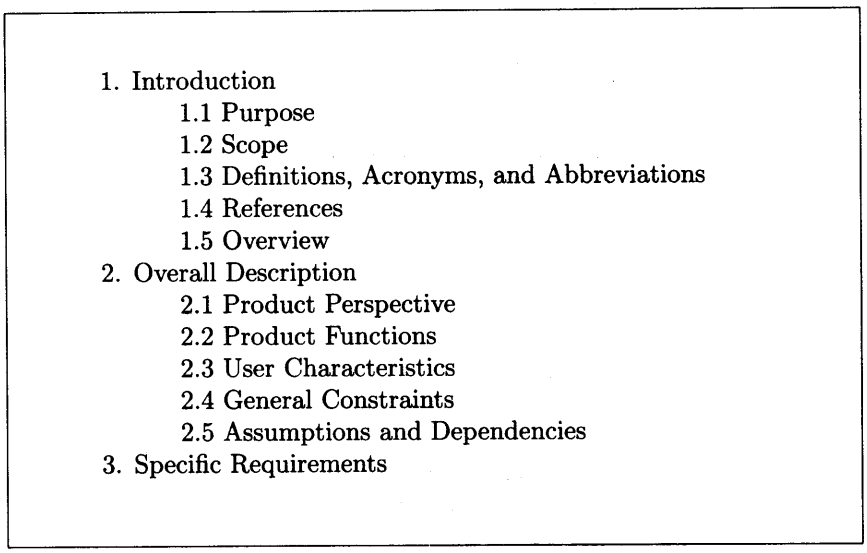


Figure 3.13: General structure of an SRS.

The introduction section 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. Section 2 describes the general factors that affect the product and its requirements. Specific requirements are not mentioned, 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 (section 3 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. For this section, different organizations have been suggested in the standard. These requirements can be organized by the modes of operation, user class, object, feature, stimulus, or functional hierarchy [92]. One method to organize the specific requirements is to first specify the external interfaces, followed by functional requirements, performance requirements, design constraints, and system attributes. This structure is shown in Figure 3 [92].

3. Specific Requirements
    3.1 External Interface Requirements
        3.1.1 User Interfaces
        3.1.2 Hardware Interfaces
        3.1.3 Software Interfaces
        3.1.4 Communication Interfaces
    3.2. Functional Requirements
        3.2.1 Mode 1
            3.2.1.1 Functional Requirement 1.1
            :
            3.2.1.n Functional Requirement 1.n
        :
        3.2.m Mode m
            3.2.m.1 Functional Requirement m.1
            :
            3.2.m.n Functional Requirement m.n
    3.3 Performance Requirements
    3.4 Design Constraints
    3.5 Attributes
    3.6 Other Requirements

Figure 3.14: One organization for specific requirements.

The external interface requirements section specifies all the interfaces of the software: to people, other softwares, 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.

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 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).

There are three other outlines proposed by the IEEE standard for organizing "specific requirements." However, these outlines are essentially guidelines. There are other ways a requirements document can be organized. 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. For different projects many of these sections may not be needed and can be omitted. Especially for smaller projects, some of the sections and subsections may not be necessary to properly specify the requirements.

When use cases (discussed next) are employed, then the functional requirements section of the SRS is replaced by use case descriptions. (The format of a use case description is discussed later.) And the product perspective part of the SRS may provide an overview or summary of the use cases.

3.4 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. We will focus on describing the behavior of software systems that are to be built.

Though use cases are primarily for specifying behavior, they can also be used effectively during analysis. Later when we discuss how to develop use cases, we will see how they can help in eliciting requirements also.

Use cases drew attention after they were used as part of the object-oriented modeling approach proposed by Jacobson [95]. Due to this connection with an object-oriented approach, use cases are sometimes viewed as part of an object-oriented approach to

software development. However, they are a general method for describing the interaction of a system (even non-IT systems.) The discussion of use cases here is based on the concepts and processes discussed in [39].

3.4.1 Basics

A software system (whose requirements are being uncovered) may be used by many users. However, in addition to users, the software system may also be used by other systems. In use case terminology, an *actor* is a person or a system which uses the system being built for achieving some goal. Note that actors need not be people only. Also, as an actor interacts for achieving some goal, it is a logical entity that represents a group of users (people or system) who behave in a similar manner. Different actors represent groups with different goals. So, it is better to have a "receiver" and a "sender" actor rather than having a generic "user" actor for a system in which some messages are sent by users and received by some other users.

A *primary actor* is the main actor that initiates a use case (UC) for achieving a goal, and whose goal satisfaction is the main objective of the use case. The primary actor is a logical concept and though we assume that the primary actor executes the use case, some agent may actually execute it on the behalf of the primary actor. For example, a VP may be the primary actor for *get sales growth report by region* use case, though it may actually be executed by an assistant. We consider the primary actor as the person who actually uses the outcome of the use case and who is the main consumer of the goal. Time driven trigger is another example of how a use case may be executed on behalf of the primary actor (in this situation the report is generated automatically at some time.)

Note, however, that although the goal of the primary actor is the driving force behind a use case, the use case must also fulfill any goals that other stakeholders might also have for this use case. That is, the main goal of a use case is to describe behavior of the system that results in satisfaction of the goals of all the stakeholders, although the use case may be driven by the goals of the primary actor. For example, a use case "withdraw money from the ATM" has a customer as its primary actor and will normally describe the entire interaction of the customer with the ATM. However, the bank is also a stakeholder of the ATM system and its interests may include that all steps are logged, money is given only if there are sufficient funds in the account, and no more than some amount is given at a time, etc. Satisfaction of these goals should also be shown by the use case "Withdraw money from the ATM."

For describing interaction, use cases use scenarios. A *scenario* describes a set of actions that are performed to achieve a goal under some specified conditions. The set of actions is generally specified as a sequence (as that is the most convenient way to express it in text), though in actual execution the actions specified may be executed in parallel or in some different order. Each step in a scenario is a logically complete

action performed either by the actor or the system. Generally, a step is some action by the actor (e.g., enter information), some logical step that the system performs to progress towards achieving its goals (e.g., validate information, deliver information), or an internal state change by the system to satisfy some goals (e.g., log the transaction, update the record.)

A use case always has a *main success scenario*, which describes the interaction if nothing fails and all steps in the scenario succeed. There may be many success scenarios. Though the UC aims to achieve its goals, different situations can arise while the system and the actor are interacting which may not permit the system to achieve the goal fully. For these situations, a use case has *extension scenarios* which describe the system behavior if some of the steps in the main scenario do not complete successfully. Sometimes they are also called *exception scenarios.* A use case is a collection of all the success and extension scenarios related to the goal. The terminology of use cases is summarized in Table 3.

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 3.2: Use Case terms.

To achieve the desired goal, a system can divide it into sub-goals. Some of these sub-goals may be achieved by the system itself, but they may also be treated as separate use cases executed by supporting actors, which may be another system. For example, suppose for verifying a user in "Withdraw money from the ATM" an authentication service is used. The interaction with this service can be treated as a separate use case. A scenario in a use case may therefore employ another use case for performing some of the tasks. In other words, use cases permit a hierarchic organization.

It should be evident that the basic system model that use cases assume is that a system primarily responds to requests from actors who use the system. By describing

the interaction between actors and the system, the system behavior can be specified, and through the behavior its functionality is specified. A key advantage of this approach is that use cases focus on external behavior, thereby cleanly avoiding doing internal design during requirements, something that is desired but not easy to do with many modeling approaches.

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. We will not discuss use case diagrams further.

3.4.2 Examples

Let us illustrate these concepts with a few use cases, which we will also use to explain other concepts related to use cases. 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.

In this system, though we have the same people who might be buying and selling, we dont have "users" as actors. Instead we have "buyers" and "sellers" as separate logical actors, as both have different goals to achieve. Besides these, the auction system itself is a stakeholder and an actor. The financial system is another. Let us first consider the main use cases of this system—"put some item for auction," "make a bid," and "complete an auction." These use cases are given in Fig 3.

The use cases are self-explanatory. This is the great value of use cases—they are natural and story-like which makes them easy to understand by both an analyst and a layman. This helps considerably in minimizing the communication gap between the developers and other stakeholders.

Some points about the use case are worth discussing. The use cases are generally numbered for reference purposes. The name of the use case specifies the goal of the primary actor (hence there is no separate line specifying the goal). The primary actor can be a person or a system—for UC1 and UC2 they are persons but for UC3, it is a system. The primary actor can also be another software which might request a service.

- ***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 3.15: 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 3.15: Main use cases in an auction system (contd.)

The *precondition* of a use case specifies what the system will ensure before allowing the use case to be initiated. Common preconditions are "user is logged in," "input data exists in files or other data structures," etc. For an operation like delete it may be that "item exists," or for a tracking use case it may be that the "tracking number is valid."

It is worth noting that the use case description lists contains some actions that are not necessarily tied to the goals of the primary actor. For example, the last step in UC 2 is to update the bid price of other bidders. This action is clearly not needed by the current bidder for his goal. However, as the system and other bidders are also stakeholders for this use case, the use case has to ensure that their goals are also satisfied. Similar is the case with the last item of UC1.

The exception situations are also fairly clear. We have listed only the most obvious ones. There can be many more, depending on the goals of the organization. For example, there could be one "user does not complete the transaction," which is a failure condition that can occur anywhere. What should be done in this case has to then be specified (e.g., all the records are cleaned).

A use cases can employ other use cases to perform some of its work. For example, in UC2 actions like "block the necessary funds" or "Debit bidder's account and credit seller's" are actions that need to be performed for the use case to succeed. However, they are not performed in this use case, but are treated as use cases themselves whose behavior has to be described elsewhere. If these use cases are also part of the system being built, then there must be descriptions of these in the requirements document. If they belong to some other system, then proper specifications about them will have to be

obtained. The financial actions may easily be outside the scope of the auction system, so will not be described in the SRS. However, actions like "search" and "browse" are most likely part of this system and will have to be described in the SRS.

This allows use cases to be hierarchically organized and refinement approach can be used to define a higher level use case in terms of lower services and then defining the lower services. However, these lower-level use cases are proper use cases with a primary actor, main scenario, etc. The primary actor will often be the primary actor of the higher level use case. For example, the primary actor for the use case "find an item" is the buyer. It also implies that while listing the scenarios, new use cases and new actors might emerge. In the requirements document, all the use cases that are mentioned in this one will need to be specified if they are a part of the system being built.

3.4.3 Extensions

Besides specifying the primary actor, its goal, and the success and exceptional scenarios, a use case can also specify a scope. If the system being built has many subsystems, as is often the case, sometimes system use cases may actually be capturing the behavior of some subsystem. In such a situation it is better to specify the scope of that use case as the subsystem. For example, a use case for a system may be log in. Even though this is a part of the system, the interaction of the user with the system described in this use case is limited to the interaction with the "login and authentication" subsystem. If the architecture of the system has identified "login and authentication" as a subsystem or a component, then it is better to specify it as the scope. Generally, a business use case has the enterprise or the organization as the scope; a system use case has the system being built as the scope; and a component use case is where the scope is a subsystem.

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. (E.g., for processing an application the HR department has to do some things, the travel department has to arrange the travel and lodging, and the technical department has to conduct the interview.) The system and subsystem use cases are generally of the type that can be completed in one relatively short sitting. All the three use cases above are system use cases. As mentioned before, we will focus on describing the behavior of the software system we are interested in building. However, the enterprise level UCs provide the context in which the systems operate. Hence, 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.

For example, let us describe the overall use case of performing an auction. A possible use case is given below in Fig 3. This use case is not a one-sitting use case and is really a business process, which provides the context for the earlier use cases. It is this use case that the earlier three use cases exist. Though this use case is also largely done by the system and is probably part of the system being built, frequently such use cases

may not be completely part of the software system and may involve manual steps as well. For example, in the "auction an item" use case, if the delivery of the item being auctioned was to be ensured by the auctioning site, then that will be a step in this use case and it will be a manual step.

- ***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 3.16: A summary level use case.

Use cases may also specify post conditions for the main success scenario, or some minimal guarantees they provide in all conditions. For example, in some use cases, atomicity may be a minimal guarantee. That is, no matter what exceptions occur either the entire transaction will be completed and the goal achieved, or the system state will be as if nothing was done. With atomicity, there will be no partial results and any partial changes will be rolled back.

3.4.4 Developing Use Cases

UCs not only document requirements, as their form is like story telling and uses text, both of which are easy and natural with different stakeholders, they also are a good medium for discussion and brainstorming. Hence, UCs can also be used for requirements elicitation and problem analysis. While developing use cases, informal or formal models may also be built, though they are not required.

UCs can be evolved in a stepwise refinement manner with each step adding more details. This approach allows UCs to be presented at different levels of abstraction. Though any number of levels of abstractions are possible, four natural levels emerge:

- **Actors and goals.** The actor-goal list enumerates the use cases and specifies the actors for each goal. The name of the use case is generally the goal. This table

may be extended by giving a brief description of each of the use cases. At this level, the use cases specify the scope of the system and give an overall view of what it does. Completeness of functionality can be assessed fairly well by reviewing these.

- **Main success scenarios.** For each of the use cases, the main success scenarios are provided at this level. With the main scenarios, the system behavior for each use case is specified. This description can be reviewed to ensure that interests of all the stakeholders are met and that the use case is delivering the desired behavior.

- **Failure conditions.** Once the success scenario is listed, all the possible failure conditions can be identified. At this level, for each step in the main success scenario, the different ways in which a step can fail form the failure conditions. Before deciding what should be done in these failure conditions (which is done at the next level), it is better to enumerate the failure conditions and reviewed for completeness.

- **Failure handling.** This is perhaps the most costly and difficult part of writing a use case. Often the focus is so much on the main functionality that people do not pay attention to how failures should be handled. Determining what should be the behavior under different failure conditions will often identify new business rules or new actors.

The different levels can be used for different purposes. For discussion on overall functionality or capabilities of the system, actors and goal level description is very useful. Failure conditions, on the other hand, are very useful for understanding and extracting detailed requirements and business rules under special cases.

The four levels can also guide the analysis activity. First just identify the actors and their goals and get an agreement with the concerned stakeholders on that. The actor-goal list will clearly define the scope of the system and will provide an overall view of what the system capabilities are. Then the main success scenario for each UC can be evolved, giving more details about the main functions of the system. Interaction and discussion are the primary means to uncover these scenarios though models may be built, if required. When the main success scenario for a use case is agreed upon and the main steps in its execution are specified, then the failure conditions can be examined. Enumerating failure conditions is an excellent method of uncovering special situations that can occur and which must be handled by the system. Finally, what should be done for these failure conditions should be examined and specified. As details of handling failure scenarios can require a lot of effort and discussion, it is better to first enumerate the different failure conditions and then get the details of these scenarios. Very often, when deciding the failure scenarios, many new business rules of how to deal with these scenarios get uncovered. Note that during this process an analyst may have to go back

to earlier steps as during some detailed analysis new actors may emerge or new goals and new use cases may get uncovered. That is, using use cases for analysis is also an interactive task.

What should be the level of detail in a use case? There is no one answer to to a question like this; the actual 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. For example, if there is a small co-located team building the system, it is quite likely that use cases which list the main exception conditions and give a few key steps for the scenarios will suffice. On the other hand, for a project whose development is to be subcontracted to some other organization, it is better to have more detailed use cases.

For writing use cases, general technical writing rules apply. Use simple grammar, clearly specify who is performing the step, and keep the overall scenario as simple as possible. Also, when writing steps, for simplicity, it is better to combine some steps into one logical step, if it makes sense. For example steps "user enters his name," "user enter his SSN," and "user enters his address" can be easily combined into one step "user enters personal information."

3.5 Validation

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. Furthermore, as we have seen, the longer an error remains undetected, the greater the cost of correcting it. Hence, it is extremely desirable 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 room 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" (some desirable quality objectives are given later).

Before we discuss validation, let us consider the type of errors that typically occur in an SRS. Many different types of errors are possible, but the most common errors that occur 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.

Some projects have collected data about requirement errors. In [47] the effectiveness of different methods and tools in detecting requirement errors in specifications for a data processing application is reported. On an average, a total of more than 250 errors were detected, and the percentage of different types of errors was:

Omission	Incorrect Fact	Inconsistency	Ambiguity
26%	10%	38%	26%

In [8] the errors detected in the requirements specification of the A-7 project (which deals with a real-time flight control software) were reported. A total of about 80 errors were detected, out of which about 23% were clerical in nature. Of the remaining, the distribution with error type was:

Omission	Incorrect Fact	Inconsistency	Ambiguity
32%	49%	13%	5%

Though the distribution of errors is different in these two cases, reflecting the difference in application domains and the error detection methods used, they do suggest that the major problems (besides clerical errors) are omission, incorrect fact, inconsistency, and ambiguity. If we take the average of the two data tables, it shows that all four classes of errors are very significant, and a good fraction of errors belong to each of these types. This implies, that besides improving the quality of the SRS itself (e.g., no clerical errors), the validation should focus on uncovering these types of errors.

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. We have discussed the general procedure of inspections in an earlier chapter. Here we only discuss some aspects relevant to requirements reviews.

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 [52]:

- 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. The data in [8] about the A-7 project shows that about 33% of the total requirement errors detected were detected by review processes, and about 45% of the requirement errors were detected during the design phase when the requirement document is used as a reference for design. This clearly suggests that if requirements are reviewed then not only a substantial fraction of the errors are detected by them, but a vast majority of the remaining errors are detected soon afterwards in the design activity.

Though requirements reviews remain the most commonly used and viable means for requirement validation, other possibilities arise if some special purpose tools for modeling and analysis are 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 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.

3.6 Metrics

As we stated earlier, the basic purpose of metrics at any point during a development project is to provide quantitative information to the management process so that the information can be used to effectively control the development process. Unless the metric is useful in some form to monitor or control the cost, schedule, or quality of the project, it is of little use for a project. There are very few metrics that have been defined for requirements, and little work has been done to study the relationship between the metric values and the project properties of interest. This says more about the state of the art of software metrics, rather than the usefulness of having such metrics. In this section, we will discuss some of the metrics and how they can be used.

3.6.1 Size: Function Points

A major problem after requirements are done is to estimate the effort and schedule for the project. For this, some metrics are needed that can be extracted from the requirements and used to estimate cost and schedule (through the use of some model). As the primary factor that determines the cost (and schedule) of a software project is its size, a metric that can help get an idea of the size of the project will be useful for estimating cost. This implies that during the requirement phase measuring the size of the requirement specification itself is pointless, unless the size of the SRS reflects the effort required for the project. This also requires that relationships of any proposed size measure with the ultimate effort of the project be established before making general use of the metric.

A commonly used size metric for requirements is the size of the text of the SRS. The size could be in *number of pages*, *number of paragraphs*, *number of functional requirements*, etc. As can be imagined, these measures are highly dependent on the authors of the document. A verbose analyst who likes to make heavy use of illustrations may produce an SRS that is many times the size of the SRS of a terse analyst. Similarly, how much an analyst refines the requirements has an impact on the size of the document. Generally, such metrics cannot be accurate indicators of the size of the project. They are used mostly to convey a general sense about the size of the project.

Function points [2] are one of the most widely used measures of software size. The basis of function points is that the "functionality" of a system, that is, what the system performs, is the measure of the system size. And as functionality is independent of

how the requirements of the system are specified, or even how they are eventually implemented, such a measure has a nice property of being dependent solely on the system capabilities. In function points, the system functionality is calculated in terms of the number of functions it implements, the number of inputs, the number of outputs, etc.—parameters that can be obtained after requirements analysis and that are independent of the specification (and implementation) language.

The original formulation for computing the function points uses the count of five different parameters, namely, external input types, external output types, logical internal file types, external interface file types, and external inquiry types. According to the function point approach, these five parameters capture the entire functionality of a system. However, two elements of the same type may differ in their complexity and hence should not contribute the same amount to the "functionality" of the system. To account for complexity, each parameter in a type is classified as *simple*, *average*, or *complex*. The definition of each of these types and the interpretation of their complexity levels is given later [2].

Each unique input (data or control) type that is given as input to the application from outside is considered of *external input type* and is counted. An external input type is considered unique if the format is different from others or if the specifications require a different processing for this type from other inputs of the same format. The source of the external input can be the user, or some other application, files. An external input type is considered *simple* if it has a few data elements and affects only a few internal files of the application. It is considered *complex* if it has many data items and many internal logical files are needed for processing them. The complexity is *average* if it is in between. Note that files needed by the operating system or the hardware (e.g., configuration files) are not counted as external input files because they do not belong to the application but are needed due to the underlying technology.

Similarly, each unique output that leaves the system boundary is counted as an *external output type*. Again, an external output type is considered unique if its format or processing is different. Reports or messages to the users or other applications are counted as external input types. The complexity criteria are similar to those of the external input type. For a report, if it contains a few columns it is considered *simple*, if it has multiple columns it is considered *average*, and if it contains complex structure of data and references many files for production, it is considered *complex*.

Each application maintains information internally for performing its functions. Each logical group of data or control information that is generated, used, and maintained by the application is counted as a *logical internal file type*. A logical internal file is *simple* if it contains a few record types, *complex* if it has many record types, and *average* if it is in between.

Files that are passed or shared between applications are counted as *external interface file type*. Note that each such file is counted for all the applications sharing it. The complexity levels are defined as for logical internal file type.

A system may have queries also, where a query is defined as an input-output combination where the input causes the output to be generated almost immediately. Each unique input-output pair is counted as an *external inquiry type*. A query is unique if it differs from others in format of input or output or if it requires different processing. For classifying the query type, the input and output are classified as for external input type and external output type, respectively. The query complexity is the larger of the two.

Each element of the same type and complexity contributes a fixed and same amount to the overall function point count of the system (which is a measure of the functionality of the system), but the contribution is different for the different types, and for a type, it is different for different complexity levels. The amount of contribution of an element is shown in Table 3 [2, 113].

Function type	Simple	Average	Complex
External input	3	4	6
External output	4	5	7
Logical internal file	7	10	15
External interface file	5	7	10
External inquiry	3	4	6

Table 3.3: Function point contribution of an element.

Once the counts for all five different types are known for all three different complexity classes, the raw or unadjusted function point (UFP) can be computed as a weighted sum as follows:

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} C_{ij}$$

where i reflects the row and j reflects the column in Table 3; w_{ij} is the entry in the ith row and jth column of the table (i.e., it represents the contribution of an element of the type i and complexity j); and C_{ij} is the count of the number of elements of type i that have been classified as having the complexity corresponding to column j.

Once the UFP is obtained, it is adjusted for the environment complexity. For this, 14 different characteristics of the system are given. These are data communications, distributed processing, performance objectives, operation configuration load, transaction rate, on-line data entry, end user efficiency, on-line update, complex processing logic, re-usability, installation ease, operational ease, multiple sites, and desire to facilitate change. The degree of influence of each of these factors is taken to be from 0 to 5, rep-

resenting the six different levels: not present (0), insignificant influence (1), moderate influence (2), average influence (3), significant influence (4), and strong influence (5). The 14 degrees of influence for the system are then summed, giving a total N (N ranges from 0 to 14*5=70). This N is used to obtain a complexity adjustment factor (CAF) as follows:

$$CAF = 0.65 + 0.01N.$$

With this equation, the value of CAF ranges between 0.65 and 1.35. The delivered function points (DFP) are simply computed by multiplying the UFP by CAF. That is,

$$\text{Delivered Function Points} = CAF * \text{Unadjusted Function Points}.$$

As we can see, by adjustment for environment complexity, the DFP can differ from the UFP by at most 35%. The final function point count for an application is the computed DFP.

Function points have been used as a size measure extensively and have been used for cost estimation. Studies have also been done to establish correlation between DFP and the final size of the software (measured in lines of code.) For example, according to one such conversion given in www.theadvisors.com/langcomparison.htm, one function point is approximately equal to about 125 lines of C code, and about 50 lines of C++ or Java code. By building models between function points and delivered lines of code (and existing results have shown that a reasonably strong correlation exists between DFP and KLOC so that such models can be built), one can estimate the size of the software in KLOC, if desired.

As can be seen from the manner in which the functionality of the system is defined, the function point approach has been designed for the data processing type of applications. For data processing applications, function points generally perform very well [106] and have now gained a widespread acceptance. For such applications, function points are used as an effective means of estimating cost and evaluating productivity. However, its utility as a size measure for nondata processing types of applications (e.g., real-time software, operating systems, and scientific applications) has not been well established, and it is generally believed that for such applications function points are not very well suited.

A major drawback of the function point approach is that the process of computing the function points involves subjective evaluation at various points and the final computed function point for a given SRS may not be unique and can depend on the analyst. Some of the places where subjectivity enters are: (1) different interpretations of the SRS (e.g., whether something should count as an external input type or an external interface type; whether or not something constitutes a logical internal file; if two reports differ in a very minor way should they be counted as two or one); (2) complexity estimation of a user function is totally subjective and depends entirely on the analyst (an analyst may classify something as complex while someone else may classify it as average) and complexity can have a substantial impact on the final count as the weighs for simple

and complex frequently differ by a factor of 2; and (3) value judgments for the environment complexity. These factors make the process of function point counting somewhat subjective. Organizations that use function points try to specify a more precise set of counting rules in an effort to reduce this subjectivity. It has also been found that with experience this subjectivity is reduced [113]. Overall, despite this subjectivity, use of function points for data processing applications continues to grow.

The main advantage of function points over the size metric of KLOC, the other commonly used approach, is that the definition of DFP depends only on information available from the specifications, whereas the size in KLOC cannot be directly determined from specifications. Furthermore, the DFP count is independent of the language in which the project is implemented. Though these are major advantages, another drawback of the function point approach is that even when the project is finished, the DFP is not uniquely known and has subjectivity. This makes building of models for cost estimation hard, as these models are based on information about completed projects (cost models are discussed further in the next chapter). In addition, determining the DFP—from either the requirements or a completed project—cannot be automated. That is, considerable effort is required to obtain the size, even for a completed project. This is a drawback compared to KLOC measure, as KLOC can be determined uniquely by automated tools once the project is completed.

3.6.2 Quality Metrics

As we have seen, the quality of the SRS has direct impact on the cost of the project. Hence, it is important to ensure that the SRS is of good quality. For this, some quality metrics are needed that can be used to assess the quality of the SRS. Quality of an SRS can be assessed either directly by evaluating the quality of the document by estimating the value of one or more of the quality attributes of the SRS, or indirectly, by assessing the effectiveness of the quality control measures used in the development process during the requirements phase. Quality attributes of the SRS are generally hard to quantify, and little work has been done in quantifying these attributes and determining correlation with project parameters. Hence, the use of these metrics is still limited. However, process-based metrics are better understood and used more widely for monitoring and controlling the requirements phase of a project.

Number of errors found is a process metric that is useful for assessing the quality of requirement specifications. Once the number of errors of different categories found during the requirement review of the project is known, some assessment can be made about the SRS from the size of the project and historical data. This assessment is possible if the development process is under statistical control. In this situation, the error distribution during requirement reviews of a project will show a pattern similar to other projects executed following the same development process. From the pattern of errors to be expected for this process and the size of the current project (say, in function

points), the volume and distribution of errors expected to be found during requirement reviews of this project can be estimated. These estimates can be used for evaluation.

For example, if much fewer than expected errors were detected, it means that either the SRS was of very high quality or the requirement reviews were not careful. Further analysis can reveal the true situation. If too many clerical errors were detected and too few omission type errors were detected, it might mean that the SRS was written poorly or that the requirements review meeting could not focus on "larger issues" and spent too much effort on "minor" issues. Again, further analysis will reveal the true situation. Similarly, a large number of errors that reflect ambiguities in the SRS can imply that the problem analysis has not been done properly and many more ambiguities may still exist in the SRS. Some project management decision to control this can then be taken (e.g., build a prototype or do further analysis).

Clearly, review data about the number of errors and their distribution can be used effectively by the project manager to control quality of the requirements. From the historical data, a rough estimate of the number of errors that remain in the SRS after the reviews can also be estimated. This can be useful in the rest of the development process as it gives some handle on how many requirement errors should be revealed by later quality assurance activities.

Requirements rarely stay unchanged. Change requests come from the clients (requesting added functionality, a new report, or a report in a different format, for example) or from the developers (infeasibility, difficulty in implementing, etc.). *Change request frequency* can be used as a metric to assess the stability of the requirements and how many changes in requirements to expect during the later stages.

Many organizations have formal methods for requesting and incorporating changes in requirements. We have earlier seen a requirements change management process. Change data can be easily extracted from these formal change approval procedures. The frequency of changes can also be plotted against time. For most projects, the frequency decreases with time. This is to be expected; most of the changes will occur early, when the requirements are being analyzed and understood. During the later phases, requests for changes should decrease.

For a project, if the change requests are not decreasing with time, it could mean that the requirements analysis has not been done properly. Frequency of change requests can also be used to "freeze" the requirements—when the frequency goes below an acceptable threshold, the requirements can be considered frozen and the design can proceed. The threshold has to be determined based on experience and historical data.

3.7 Summary

The main goal of the requirements phase is to produce the software requirements specification (SRS), which accurately captures the client's requirements and which forms the basis of software development and validation. The basic reason for the difficulty in specifying software requirements comes from the fact that there are three interested parties—the client, the end users, and the software developer. The requirements document has to be such that the client and users can understand it easily and the developers can use it as a basis for software development. Due to the diverse parties involved in software requirements specification, a communication gap exists. This makes the task of requirements specification difficult.

There are three basic activities in the requirements phase. The first is problem or requirement analysis. The goal of this activity is to understand such different aspects as the requirements of the problem, its context, and how it fits within the client's organization. The second activity is requirements specification, during which the understood problem is specified or written, producing the SRS. And the third activity is requirements validation, which is done to ensure that the requirements specified in the SRS are indeed what is desired.

There are three main approaches to analysis; unstructured approaches rely on interaction between the analyst, customer, and user to reveal all the requirements (which are then documented). The second is the modeling-oriented approach, in which a model of the problem is built based on the available information. The model is useful in determining if the understanding is correct and in ensuring that all the requirements have been determined. Modeling may be function-oriented or object-oriented. The third approach is the prototyping approach in which a prototype is built to validate the correctness and completeness of requirements.

To satisfy its goals, an SRS should possess characteristics like completeness, consistency, unambiguous, verifiable, modifiable, etc. A good SRS should specify all the functions the software needs to support, performance of the system, the design constraints that exist, and all the external interfaces.

One method for specifying the functional specifications that has become popular is the use case approach. With this approach the functionality of the system is specified through use cases, with each use case specifying the behavior of the system when a user interacts with it for achieving some goal. Each use case contains a normal scenario, as well as many exceptional scenarios, thereby providing the complete behavior of the system. Though use cases are meant for specification, as they are natural and story-like, by expressing them at different levels of abstraction they can also be used for problem analysis.

For validation, the most commonly used method is reviewing or inspecting the requirements. In requirements inspections, the team of reviewers also includes a representative of the client to ensure that all requirements are captured.

The main metric of interest for requirements is some quantification of system size, as it can be used to estimate the effort requirement of the project. The most commonly used size metric for requirements is the function points. The function point metric attempts to quantify the functionality of the system in terms of five parameters and their complexity levels which can be determined from the requirements of the system. Based on the count of these five parameters for different complexity levels, and the value of fourteen different environmental factors, the function point count for a system is obtained. The function point metric can be used for estimating the cost of the system.

Exercises

1. Is it possible to have a system that can automatically verify completeness of an SRS document? Explain your answer.
2. Construct an example of an inconsistent (incomplete) SRS.
3. How can you specify the "maintainability" and "user friendliness" of a software system in quantitative terms?
4. For a complete and unambiguous response time requirement, the environmental factors on which the response time depends must be specified. Which factors should be considered, and what units should be chosen to specify them?
5. The basic goal of the requirements activity is to get an SRS that has some desirable properties. What is the role of modeling in developing such an SRS? List three major benefits that modeling provides, along with justifications, for achieving the basic goal.
6. Make a friend of yours as the client. Perform structured analysis and object-oriented analysis for the following:
   (a) An electronic mail system.
   (b) A simple student registration system.
   (c) A system to analyze a person's diet.
   (d) A system to manage recipes for a household.
   (e) A system to fill tax forms for the current year tax laws.
7. Write the SRS for the restaurant example whose analysis is shown in the chapter.
8. Write the functional requirements for the restaurant example using use cases.

9. Develop a worksheet for calculating the function point for a given problem specification.
10. Compute the function points for the restaurant example (can use the worksheet).

Case Studies

We introduce our two running case studies here. We give the problem description and discuss the problem analysis of these case studies. The detailed SRS for both these case studies are available from the Web site.

Case Study 1—Course Scheduling

Problem Description

The computer science department in a university offers many courses every semester, which are taught by many instructors. These courses are scheduled based on some policy directions of the department. Currently the scheduling is done manually, but the department would like to automate it. We have to first understand the problem and then produce a requirements document based on our understanding of the problem.

Problem Analysis

We do the problem analysis here—the requirements specification document is available from the Web site of the book. For analysis, we first identify the parties involved.

Client: Chairman of the computer science department.
End Users: Department secretary and instructors.

Now we begin to study the current system. After speaking with the instructors, the department chairman, and the secretary, we find that the system operates as follows. Each instructor specifies, on a sheet of paper, the course he is teaching, expected enrollment, and his preferences for lecture times. These preferences must be valid lecture times, which are specified by the department. These sheets are given to the department secretary, who keeps them in the order they are received. After the deadline expires, the secretary does the scheduling. Copies of the final schedule are sent to the instructors. The overall DFD for the system is shown in Figure 3.17.

This DFD was discussed with the chairman and the department secretary and approved by them. We now focus on the scheduling process, which is our main interest. From the chairman we found that the two major policies regarding scheduling are: (1) the post-graduate (PG) courses are given preference over undergraduate (UG) courses, and (2) no two PG courses can be scheduled at the same time.

The department secretary was interviewed at length to find out the details of the scheduling activity. The schedule of the last few semesters, together with their respective inputs (i.e., the sheets) were also studied. It was found that the basic process is as

follows. The sheets are separated into three piles—one for PG courses with preferences, one for UG courses with preferences, and one for courses with no preference. The order of the sheets in the three piles was maintained. First the courses in the PG pile were scheduled and then the courses in the UG pile were scheduled. The courses were scheduled in the order they appeared in the pile. During scheduling no backtracking was done, i.e., once a course is scheduled, the scheduling of later courses has no effect on its schedule. After all the PG and UG courses with preferences were processed, courses without any preferences were scheduled in the available slots. It was also found that information about classrooms and the department-approved lecture times was tacitly used during the scheduling. The DFD for the schedule process is shown in Figure 3.18.

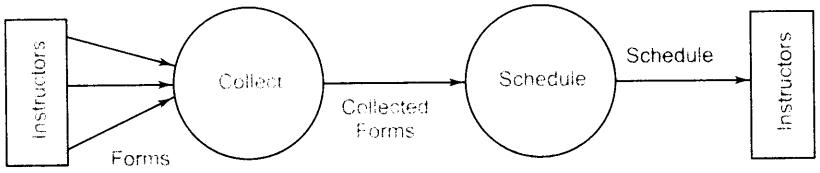


Figure 3.17: Top-level DFD for the current scheduling system.

The secretary was not able to explain the algorithm used for scheduling. It is likely that some hit-and-miss approach is being followed. However, while scheduling, the following was being checked:

1. Classroom capacity is sufficient for the course.
2. A slot for a room is never allotted to more than one course.

The two basic data flows are the sheets containing preferences and the final schedule. The data dictionary entry for these is:

collected_forms = [instructor_name +
course_number + [preferences]*]
schedule = [course_number class_room lecture_time]*

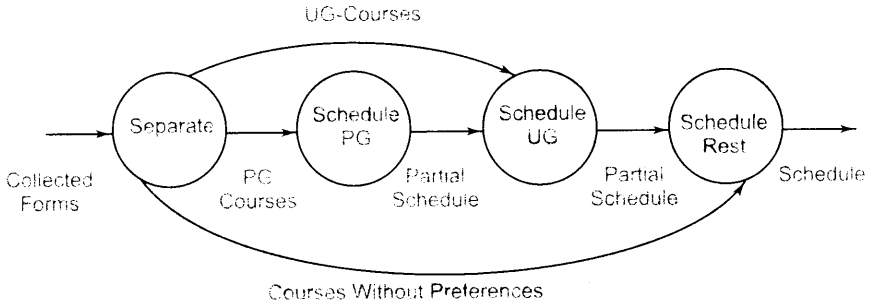


Figure 3.18: The DFD for the schedule process.

Now we have to define the DFD for the new or future automated system. Automating scheduling can affect the preference collection method, so boundaries of change include the entire system. After discussion with the chairman, the instructors, and the secretary, the following decisions were made regarding what the automated system should do and what the new environment should be:

1. The preferences will be electronically mailed to the secretary by the instructors. The secretary will put these preferences for different courses in a file in the order in which they are received. The secretary will also make entries for all courses for which no response has been given before the deadline. Entries for these courses will have no preferences.

2. The format for each course entry should be similar to the one currently being used.

3. Entries might have errors, so the system should be able to check for errors.

4. The current approach for scheduling should be followed. However, the system should make sure that scheduling of UG courses does not make a PG course without any preference unschedulable. This is not being done currently, but is desired.

5. A reason for unschedulability should be given for the preferences that are not satisfied or for courses that cannot be scheduled.

6. Information about department courses, classrooms, and valid lecture times will be kept in a file.

The DFD for the new logical system (one with automation) is shown in Figure 3.19. The two important data entities are the two files in the DFD. The data dictionary entry for these is:

prefs_file = [pref]*
pref = course_number + enrollment + [preferences]*
dept_DB = [class_rooms]* + dept_course_list + [valid_lecture_time]*
class_rooms = room_no + capacity

It is decided that the scheduling process will be automated. The rest (such as combining preferences) will be done manually. Based on the format currently used in the sheets, a detailed format for the course entries was decided and approved by the

instructors. A detailed format for the dept_DB file was also chosen and approved. The final formats and the requirements are given in the requirements document.

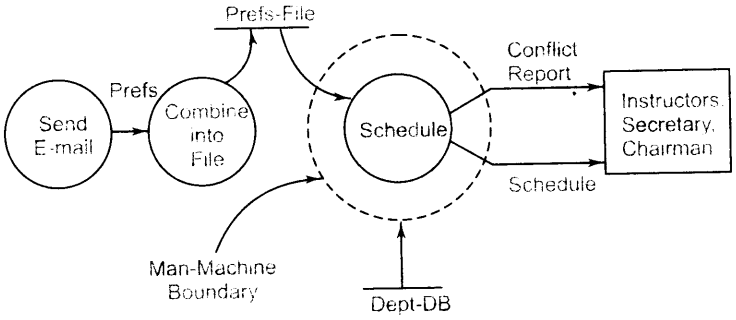


Figure 3.19: DFD for the new system.

The complete SRS of this case study, which specifies the functional requirements by enumerating the functions of the system, is available from the Web site of the book.

Case Study 2—Personal Investment Management System

Problem Description

Many people invest their money in a number of securities (shares). Generally, an investor has multiple portfolios of investments, each portfolio having investments in many securities. From time to time an investor sells or buys some securities and gets dividends for the securities. There is a current value of each security—many sites give this current value. It is proposed to build a personal investment management system (PIMS) to help investors keep track of their investments, as well as determine the rate of returns he/she is getting on the individual investments as well as on the overall portfolio. The system should also allow an investor to determine the net-worth of the portfolios.

Problem Analysis

This project started with the above problem statement. During analysis, discussion with the clients were held to clarify various issues. After discussion, the following clarifications emerged.

- An investor can have multiple portfolios of investments. A portfolio can have many investments.
- In each investment, the investor invests some moneys from time to time, and withdraws some funds from time to time. The amount invested withdrawn and the dates are provided by the investor. Any number of investments withdrawals can be made.
- There is a current value of each investment. As a default, the previously given current value can be chosen. Provision should be made to get the current value from some recognized site on the Web. If for some reason the net is down then the user should be able to specify the current value of the shares.
- An investor may also invest in instruments which have a maturity date and/or a fixed rate of return. Such investments should also be handled by the system. In addition, for such investments, the system should provide a provision of alerts (e.g., on maturity).
- The investor should be allowed to save information about his portfolio investments, etc.
- The investor should be allowed to edit entered data.

- An investor should be able to view any of his portfolios in summary [illegible] detailed form.
- Data being stored is very personal: even though the system is to work on a [illegible] it should provide some security.
- For each investment, the investor can determine the rate of return he is getting. Besides the rate of return on each investment, the investor should be given the overall rate of return for each portfolio as well as total investments. Information like how much money invested, how much has been earned, etc. can also be shown.
- Rates of return can be computed on a monthly basis. For example, month is the smallest unit for computing returns. The yearly return will be computed from this monthly return using monthly compounding (i.e., yearly return = (1+ monthly return)**12 - 1.)

During the discussions, the scope of the project also got defined.

In Scope

- Managing investment of a single user, which would include maintaining bookkeeping information about entities like Portfolio, Security, and Transaction
- Computation of Net-Worth and Rate of Investment (ROI) of the Investor
- Giving alerts to the user, if it is requested
- Downloading the current prices of shares from the Web
- User authentication

Out of Scope

- Features for actual purchasing and selling of securities. That is, actually buying and selling of shares, securities is done outside PIMS.
- Tax computations for gains, losses
- Any market related prediction

Key Use Cases

For this project, a use-case-based requirement analysis and specification is done. That is, use cases are used for analysis as well as specification. The main actors for PIMS are the user and the system. During analysis, first the major use cases categories and key use cases in each category are identified. The broad categories and the use cases in each category are given in the table below.

Use Case Category	Use Cases
Installation	Installation
System authorization	Login, Change Password
Portfolio related	Create portfolio, Rename portfolio, Delete portfolio
Securities related	Create security, Rename security, Delete security
Transaction related	Add transaction, Edit transaction, Delete transaction
Information display	Display investment, Display portfolio, Display security
Computation	Compute net-worth, Compute ROI
Share prices	Get current share price, Edit share price
Alerts	Set alerts, Show alerts, Delete alerts

Once the main use cases were identified and agreed, details of the use cases were uncovered. As discussed earlier, first we defined the main success scenario for the use cases, and then we identified the exception scenarios.

The complete SRS for this case study is available from the book's Web site.

Chapter 4

Software Architecture

A system is an entity that provides some behavior to its environment, where the environment can consist of people or other systems. In the previous chapter we saw that expected behavior of a proposed software system is defined through a software requirement specification (SRS) document. For building the specified software system, designing the software architecture is a key step, and is the topic of this chapter.

Any complex system is composed of sub systems that interact under the control of system design such that the system provides the expected behavior. While designing such a system, therefore, the logical approach is to identify the sub-systems that should compose the system, the interfaces of these subsystems, and the rules for interaction between the subsystems. This is what software architecture aims to do.

Software architecture is a relatively recent area. As the software systems increasingly become distributed and more complex, architecture becomes an important step in building the system. Due to a wide range of options now available for how a system may be configured and connected, carefully designing the architecture becomes very important. It is during the architecture design where choices like using some type of middleware, or some type of back end database, or some type of server, or some type of security component are made. It is not possible to design the details of the system and then try to accommodate these choices—the architecture must be created such that these decisions have been incorporated suitably in the system structure. Architecture is also the earliest place when properties like reliability and performance can be evaluated for the system, a capability that is increasingly becoming important.

In this chapter, we will focus primarily on architecture concepts and some notation for describing architecture. The issue of methodology, that is, how architecture should be created, is not discussed as architecture is a high-level creative activity for which methodologies do not really exist. However, we will discuss some architecture styles, which suggest some forms of architectures. A combination of some variation of these styles is likely to be useful for many systems. We also discuss some issues relating to software architectures like documentation, relationship to design, etc., and discuss

one approach for analyzing architectures. We end the chapter with a dsicussion of architectures for the two case studies.

4.1 Role of Software Architecture

What is architecture? We must have a clear answer to this before we further discuss what its role is in building a software system and how we can go about creating and representing architecture.

At a top level, architecture is a design of a system which gives a very high level view of the parts of the system and how they are related to form the whole system. That is, architecture partitions the system in logical parts such that each part can be comprehended independently, and then describes the system in terms of these parts and the relationship between these parts.

Any complex system can be partitioned in many different ways, each providing an useful view and each having different types of logical parts. The same holds true for a software system—there is no unique structure of the system that can be described by its architecture; there are many possible structures.

Due to this possibility of having multiple structures, one of the most widely accepted definitions of software architecture is that *the software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [9].* This definition implies that for elements in an architecture, we are only interested in those abstractions that specify those properties that other elements can assume to exist and that are needed to specify relationships. These properties could be about the functionality or services the component provides, or the performance and other quality properties it provides. Details on how these properties are supported are not needed for architecture. This is an important capability that allows architecture descriptions to represent a complex system in a succinct form that is easily comprehended. The definition also implies that the behavior of the elements is part of the architecture; hence any architecture documentation must clearly specify the behavior. Finally, as with most definitions, this definition does not say anything about whether an architecture is good or bad—this determination has to be done through some analysis.

An architecture description of a system will therefore describe the different structures of the system. The next natural question that arises is what are these structures in an architecture description good for? Why should a team building a software system for some customer be interested in creating and documenting the structures of the proposed system. Some of the important uses that software architecture descriptions play are [9, 35, 93].

1. *Understanding and communication*: An architecture description is primarily to communicate the architecture to its various stakeholders, which include the users who will use the system, the clients who commissioned the system, the builders

who will build the system, and, of course, the architects. An architecture description is an important means of communication between these various stakeholders. Through this description the stakeholders gain an understanding of some macro properties of the system and how the system intends to fulfill the functional and quality requirements. As the description provides a common language between stakeholders, it also becomes the vehicle for negotiation and agreement amongst the stakeholders, who may have conflicting goals.

Clearly, to facilitate communication, software architecture descriptions must facilitate the understanding of systems. An architecture description of the proposed system describes how the system will be composed, when it is built. By partitioning the system into parts, and presenting the system at a higher level of abstraction as composed of subsystems and their interactions, detailed level complexity is hidden. This facilitates the understanding of the system and its structure.

Though we are focusing on new systems being created, it should be pointed out that architecture descriptions can also be used to understand an existing system— by specifying different high level views of the system structure, a system description is simplified with details about how parts are implemented hidden away. The reduction of system to a few parts and how they work together is a tremendous aid in understanding, as it reduces the complexity and allows a person to deal with a limited complexity at a given time.

2. *Reuse*: Architecture descriptions can help software reuse. Reuse is considered one of the main techniques by which productivity can be improved, thereby reducing the cost of software. The software engineering world has, for a long time, been working towards a discipline where software can be assembled from parts that are developed by different people and are available for others to use. If one wants to build a software product in which existing components may be reused, then architecture becomes the key point at which reuse at the highest level is decided. The architecture has to be chosen in a manner such that the components which have to be reused can fit properly and together with other components that may be developed, they provide the features that are needed.

   Architecture also facilitates reuse among products that are similar and building product families such that the common parts of these different but similar products can be reused. Architecture helps specify what is fixed and what is variable in these different products, and can help minimize the set of variable elements such that different products can share software parts to the maximum. Again, it is very hard to achieve this type of reuse at a detail level.

3. *Construction and Evolution.* As architecture partitions the system into parts, some architecture provided partitioning can naturally be used for constructing the system, which also requires that the system be broken into parts such that

different teams (or individuals) can separately work on different parts. A suitable partitioning in the architecture can provide the project with the parts that need to be built to build the system. As, almost by definition, the parts specified in an architecture are relatively independent (the dependence between parts coming through their relationship), they can be built independently. Not only does an architecture guide the development, it also establishes the constraints—the system should be constructed in a manner that the structures chosen during the architecture creation are preserved. That is, the chosen parts are there in the final system and they interact in the specified manner.

The construction of a software system usually does not end in delivery of the product—a software system also evolves with time. During evolution, often new features are added to the system. The architecture of the system can help in deciding where to add the new features with minimum complexity and effort, and what the impact on the rest of the system might be of adding the new features. Also, if some changes have to be made to the existing functionality, then architecture can help determine which are the parts of the system that will be affected by this change—an exercise that is extremely important in ensuring that the change is made properly without any unforeseen side effects.

4. *Analysis.* It is highly desirable if some important properties about the behavior of the system can be determined before the system is actually built. This will allow the designers to consider alternatives and select the one that will best suit the needs. Many engineering disciplines use models to analyze design of a product for its cost, reliability, performance, etc. Architecture opens such possibilities for software also. It is possible (thought the methods are not fully developed or standardized yet) to analyze or predict the properties of the system being built from its architecture. For example, the reliability or the performance of the system can be analyzed. Such an analysis can help determine whether the system will meet the quality and performance requirements, and if not, what needs to be done to meet the requirements. For example, while building a Web site for shopping, it is possible to analyze the response time or throughput for a proposed architecture, given some assumptions about the request load and hardware. It can then be decided whether the performance is satisfactory or not, and if not, what new capabilities should be added (for example, a different architecture or a faster server for the back end) to improve it to a satisfactory level.

One can easily think of other uses of architecture as well. However, not all of these uses may be significant in a project and which of these uses is pertinent to a project depends on the nature of the project. In some projects communication may be very important, but a detailed performance analysis may be unnecessary (because the system is too small or is meant for only a few users). In some other systems, performance analysis may be the primary use of architecture.

4.2 Architecture Views

There is a general view emerging that there is no unique architecture of a system. The definition that we have adopted (given above) also expresses this sentiment. Consequently, there is no one architecture drawing of the system. The situation is similar to that of civil construction a discipline that is the original user of the concept of architecture and from where the concept of software architecture has been borrowed. For a building, if you want to see the floor plan, you are shown one set of drawings. If you are an electrical engineer and want to see how the electricity distribution has been planned, you will be shown another set of drawings. And if you are interested in safety and firefighting, another set of drawings is used. These drawings are not independent of each other—they are all about the same building. However, each drawing provides a different view of the building, a view that focuses on explaining one aspect of the building and tries to a good job at that, while not divulging much about the other aspects. And no one drawing can express all the different aspects—such a drawing will be too complex for to be of any use.

Similar is the situation with software architecture. In software, the different drawings are called views. A view represents the system as composed of some types of *elements* and *relationships* between them. Which elements are used by a view, depends on what the view wants to highlight. Different views expose different properties and attributes, thereby allowing the stakeholders and analysts to properly evaluate those attributes for the system. By focusing only on some aspects of the system, a view reduces the complexity that a reader has to deal with at a time, thereby aiding system understanding and analysis.

A view describes a structure of the system. We will use these two concepts—views and structures—interchangeably. We will also use the term architectural view to refer to a view. Many types of views have been proposed. Most of the proposed views generally belong to one of these three types [35, 9]:

- Module
- Component and connector
- Allocation

In a module view, the system is viewed as a collection of code units, each implementing some part of the system functionality. That is, the main elements in this view are modules. These views are code-based and do not explicitly represent any runtime structure of the system. Examples of modules are packages, a class, a procedure, a method, a collection of functions, and a collection of classes. The relationships between these modules are also code-based and depend on how code of a module interacts with another module. Examples of relationships in this view are "is a part of" (i.e., module B is a part of module A), "uses or depends on" (a module A uses services of module

B to perform its own functions and correctness of module A depends on correctness of module B,) and "generalization or specialization" (a module B is a generalization of a module A.)

In a component and connector (C&C) view, the system is viewed as a collection of runtime entities called components. That is, a component is a unit which has an identity in the executing system. Objects (not classes), a collection of objects, and a process are examples of components. While executing, components need to interact with others to support the system services. Connectors provide means for this interaction. Examples of connectors are pipes and sockets. Shared data can also act as a connector. If the components use some middleware to communicate and coordinate, then the middleware is a connector. Hence, the primary elements of this view are components and connectors.

An allocation view focuses on how the different software units are allocated to resources like the hardware, file systems, and people. That is, an allocation view specifies the relationship between software elements and elements of the environments in which the software system is executed. They expose structural properties like which processes run on which processor, and how the system files are organized on a file system.

An architecture description consists of views of different types, with each view exposing some structure of the system. Module views show how the software is structured as a set of implementation units, C&C views show how the software is structured as interacting runtime elements, and allocation views show how software relates to non-software structures. These three types of view of the same system form the architecture of the system, as represented in Figure 4.1.

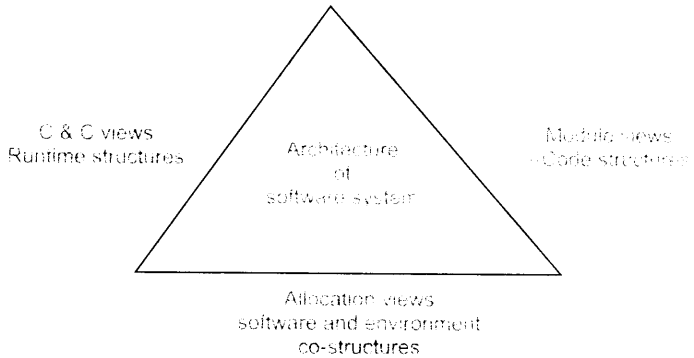


Figure 4.1: Views of Software Architecture.

Note that the different views are not unrelated. They all represent the same system. Hence, there are relationships between elements in one view and elements in another view. These relationships may be simple or may be complex. For example, the relationship between modules and components may be one to one in that one module

implements one component. On the other hand, it may be quite complex with a module being used by multiple components, and a component using multiple modules. While creating the different views, the designers have to be aware of this relationship.

The next question is what are the standard views that should be expressed for describing the architecture of a system? For answering this question, the analogy with buildings may again help. If one is building a simple small house, then perhaps there is no need to have a separate view describing the emergency and the fire system. Similarly, if there is no air conditioning in the building, there need not be any view for that. On the other hand, an office building will perhaps require both of these views, in addition to other views describing plumbing, space, wiring, etc.

The situation with software is similar which views are needed for a project depends on the project and the system being built. Depending on the needs of the project, it can be decided which views are needed. For example, if performance analysis is to be done, then the architecture must describe some component and connector view to capture the runtime structure of the system, as well as describe the allocation view to specify what hardware the different components run on. If it is to be used for planning the development, then a module view must be provided so that the different programmers or teams can be assigned different modules. In general, a large and complex project where a lot of money is at stake will require many different views so it can be analyzed from many different angles, and the risks of failures is reduced by doing so. On the other hand, for a smaller project, maybe a single view, or a couple of views, will suffice.

However, despite the fact that there are multiple drawings showing different views of a building, there is one view that predominates in construction—that of physical structure. This view forms the basis of other views in that other views cannot really be completed unless this view can be done. Other views may or may not be needed for constructing a building, depending on the nature of the project. Hence, in a sense, the view giving the building structure may be considered as the primary view in that it is almost always used, and other views rely on this view substantially. The view also captures perhaps the most important property to be analyzed in the early stages, namely, that of space organization.

The situation with software architecture is also somewhat similar. As we have said, depending on what properties are of interest, different views of the software architecture are needed. However, of these views, the component and connector (C&C) view has become the de-facto primary view, one which is almost always prepared when an architecture is designed (some definitions even view architecture only in terms of C&C views.) In this chapter, we will focus primarily on the C&C view, and will discuss the other two types only briefly. The module view will get discussed further in later chapters when discussing high level design, which focuses on identifying the different modules in the software.

4.3 Component and Connector View

Component and Connector (C&C) architecture view of a system has two main elements—components and connectors. Components are usually computational elements or data stores that have some presence during the system execution. Connectors define the means of interaction between these components. A C&C view of the system defines the components, and which component is connected to which and through what connector. A C&C view describes a runtime structure of the system—what components exist when the system is executing and how they interact during the execution. The C&C structure is essentially a graph, with components as nodes and connectors as edges.

C&C view is perhaps the most common view of architecture and most box-and-line drawings representing architecture attempt to capture this view. Most often when people talk about the architecture, they refer to the C&C view. Most architecture description languages also focus on the C&C view.

4.3.1 Components

Components are generally units of computation or data stores in the system. A component has a name, which is generally chosen to represent the role of the component or the function it performs. The name also provides a unique identity to the component, which is necessary for referencing details about the component in the supporting documents, as a C&C drawing will only show the component names.

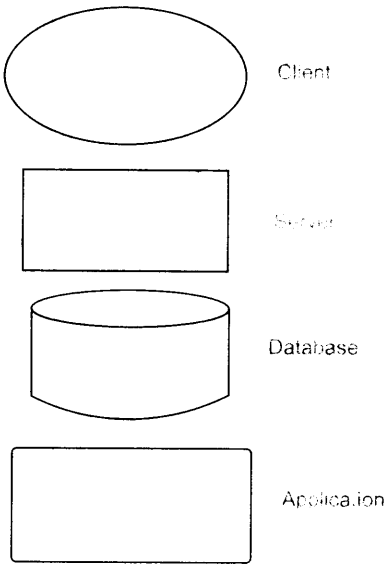


Figure 4.2: Component examples.

A component is of a component-type, where the type represents a generic component, defining the general computation and the interfaces a component of that type must have. Note that though a component has a type, in the C&C architecture view, we have components (i.e., actual instances) and not types. Examples of these types are clients, servers, filters, etc. Different domains may have other generic types like controllers, actuators, and sensors (for a control system domain.)

In a diagram representing a C&C architecture view of a system, it is highly desirable to have a different representation for different component types, so the different types can be identified visually. In a box-and-line diagram, often all components are represented as rectangular boxes. Such an approach will require that types of the components are described separately and the reader has to read the description to figure out the types of the components. It is much better to use a different symbol/notation for each different component type. If there are multiple components of the same type, then each of these components will be represented using the same symbol they will be distinguished from each other by their names.

Components use interfaces to communicate with other components. The interfaces are sometimes called ports. A component must clearly specify its ports. In a diagram, this is typically done by putting suitable marks on the edges of the symbol being used for the component.

It would be useful if there was a list of standard symbols that could be used to build an architecture diagram. However, as there is no standard list of component types, there is no such standard list. Some of the common symbols used for representing commonly found component types are shown in Figure 4.2.

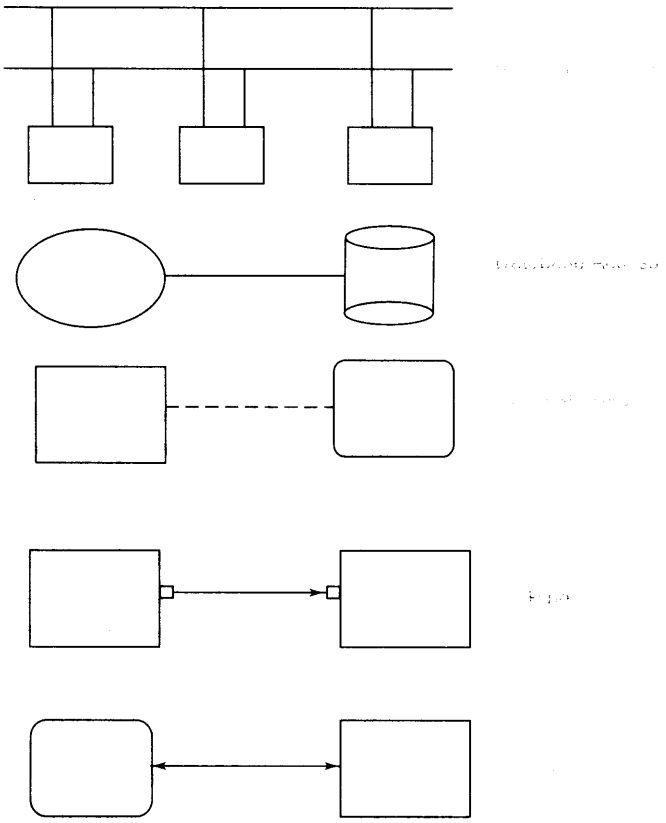
As there are no standard notations for different component types and an architect can use his own symbols, the type information cannot be obtained by a reader from the symbols used. To make sure that the meanings of the different symbols is clear to the reader, it is therefore necessary to have a key of the different symbols to describe what type of component a symbol represents.

A component is essentially a system in its own right providing some behavior at defined interfaces (i.e., ports) to its environment. Like any system, a component may be complex and have a structure of its own, which can be determined by decomposing the component. In many situations, particularly for systems that are not too large, there may not be a need to decompose the components to determine their internal architecture. We will mostly work with atomic components, that is, components whose internal structure is not needed for describing or analyzing an architecture view.

4.3.2 Connectors

The different components of a system are likely to interact while the system is in operation to provide the services expected of the system. After all, components exist to provide parts of the services and features of the system, and these must be combined to

deliver the overall system functionality. For composing a system from its components, information about the interaction between components is necessary.



· Interaction between components may be through a simple means supported by the underlying process execution infrastructure of the operating system. For example, a component may interact with another using the procedure call mechanism (a connector,) which is provided by the runtime environment for the programming language. However, the interaction may involve more complex mechanisms as well. Examples of such mechanisms are remote procedure call, TCP/IP ports, and a protocol like HTTP. These mechanisms requires a fair amount of underlying runtime infrastructure, as well as special programming within the components to use the infrastructure. Consequently, it is extremely important to identify and explicitly represent these connectors. Specification of connectors will help identify the suitable infrastructure needed to implement an architecture, as well as clarify the programming needs for components using them.

Without a proper understanding of the connectors, a realization of the components using the connectors may not be possible.

Note that connectors need not be binary and a connector may provide a n-way communication between multiple components. For example, a broadcast bus may be used as a connector, which allows a component to broadcast its message to all the other components. (Of course, how such a connector will be implemented is another issue that must be resolved before the architecture can be implemented. Generally, while creating an architecture, it is wise for the architect to use the connectors which are available on the systems on which the software will be deployed. Otherwise, there must be plans to build those connectors, or buy them, if they are available.)

A connector also has a name that should describe the nature of interaction the connector supports. A connector also has a type, which is a generic description of the interaction, specifying properties like whether it is a binary or n-way, types of interfaces it supports, etc. Sometimes, the interaction supported by a connector is best represented as a protocol. A protocol implies that when two or more components use the connector using the protocol to communicate, they must follow some conventions about order of events or commands, order in which data is to be grouped for sending, error conditions etc. For example, if TCP ports are to be used to send information from one process to another (TCP ports are the connector between the two components of process type), the protocol requires that a connection must first be established and a port number obtained before sending the information, and that the connection should be closed in the end. A protocol description makes all these constraints explicit, and defines the error conditions and special scenarios. If a protocol is used by a connector type, it should be explicitly stated.

Just like with components, in a C&C architecture diagram of a system, it is best to use a different notation for the different connector type. It is a common mistake to use a simple line or an arrow to represent all types of connectors, forcing the reader to obtain the information about type from elsewhere. However, multiple instances of the same connector type need not be always distinguished through naming, as often the components being connected can provide the unique identification. As in components, as there are no commonly accepted notations, it is best to provide a key of the notations used. Some examples of connectors are shown in Figure 4.3.

It is worth pointing out that the implementation of a connector may be quite complex and may be distributed. For example, a middleware like CORBA provides connectors that may be used by objects for interaction. However, there is a lot of code in the form of ORB (object request broker) that is needed to support this connector. It is the ORB software that does the format translations between the sender and the receiver components and performs all the communication between them (using a protocol called IIOP), besides providing a host of other services that may be needed by the objects to cooperate. Hence, explicit representation of connectors is important, particularly

in distributed systems where connectors play roles that cannot be easily changed to implicit language and OS mechanisms.

If the connector is provided by the underlying system, then the components just have to ensure that they use the connectors as per their specifications. If, however, the underlying system does not provide a connector used in an architecture, then as mentioned above, the connector will have to be implemented as part of the project to build the system. That is, during the development, not only will the components need to be developed, but resources will have to be assigned to also develop the connector. (This situation might arise for a specialized system that requires connectors that are specific to the problem domain.)

4.3.3 An Example

We have now discussed the two key elements of a C&C architecture view of a software system, and how they work together. Let us now discuss an example, putting these concepts together.

Suppose we have to design and build a simple system for taking an on-line survey of students on a campus. There is a set of multiple choice questions, and the proposed system will provide the survey form to the student, who can fill and submit it on-line. We also want that when the user submits the form, he/she is also shown the current result of the survey, that is, what percentage of students so far have filled which options for the different questions.

The system is best built using the Web; this is the likely choice of any developer. For this simple system, a traditional 3-tier architecture is proposed. It consists of a client which will display the form that the student can fill and submit, and will also display the results. The second component is the server, which processes the data submitted by the student, and saves it on the database, which is the third component. The server also queries the database to get the outcome of the survey and sends the results in proper format (HTML) back to the client, which then displays the result. A figure giving the C&C view is shown in Figure 4.4.

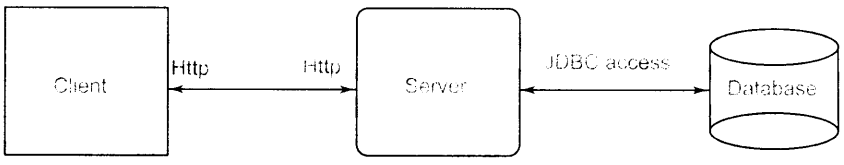


Figure 4.4: Architecture of the survey system.

Note that the client, server, and the database are all different types of components, and hence are shown using different symbols. Note also that the connectors between

the components are also of different types. The diagram makes the different types clear, making the diagram stand alone and easy to comprehend.

Note that at the architecture level, a host of details are not discussed. How is the URL of the survey set? What are the modules that go in building these components and what language they are written in? Questions like these are not the issues at this level.

Note also that the connector between the client and the server explicitly says that http is to be used. And the diagram also says that it is a Web client. This implies that it is assumed that there will be a Web browser running on the machines from which the student will take the survey. Having the http as the connector also implies that there is a proper http server running, and that the server of this system will be suitably attached to it to allow access by clients. In other words, the entire infrastructure of browser and the http server, for the purposes of this application, mainly provides the connector between the client and the server (and a virtual machine to run the client of the application).

There are some implications of choice of this connector on the components. The client will have to be written in a manner that it can send the request using http (this will imply using some type of scripting language or HTML forms). Similarly, it also implies that the server has to take its request from the http server in the format specified by the http protocol. Furthermore, the server has to send its results back to the client in the HTML format. These are all constraints on implementing this architecture. Hence, when discussing it and finally accepting it, the implications for the infrastructure as well as the implementation should be fully understood and actions should be taken to make sure that these assumptions are valid.

Extension 1

The above architecture has no security and a student can take the survey as many times as he wishes. Furthermore, even a non-student can take the survey. Now the Dean of students wants that this system be open only to registered students, and that each student is allowed to take the survey at most once. To identify the students, it was explained that each student has an account, and their account information is available from the main proxy server of the institute.

Now the architecture will have to be quite different. The proposed architecture now has a separate login form for the user, and a separate server component which does the validation. For validation, it goes to the proxy for checking if the login and password provided are valid. If so, the server returns a cookie to the client (which stores it as per the cookie protocol). When the student fills the survey form, the cookie information validates the user, and the server checks if this student has already filled the survey. The architecture for this system is shown in Figure 4.5.

Note that even though we are saying that the connection between the client and the server is that of http, it is somewhat different from the connection in the earlier

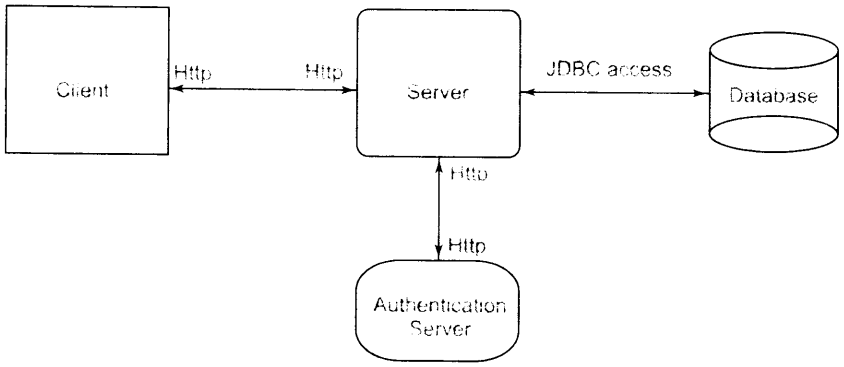


Figure 4.5: Architecture for the survey system with authentication.

architecture. In the first architecture, plain http is sufficient. In this one, as cookies are also needed, the connector is really http + cookies. So, if the user disables cookies, the required connector is not available and this architecture will not work.

Extension II

Now suppose, we want the system to be extended in a different way. It was found that the database server is somewhat unreliable, and is frequently down. It was also felt that when the student is given the result of the survey when he submits the form, a somewhat outdated result is acceptable, as the results are really statistical data and a little inaccuracy will not matter. We assume that the survey result can be outdated by about 5 data points (even if it does not include data of 5 surveys, it is OK). What the Dean wanted was to make the system more reliable, and provide some facility for filling the survey even when the database is down.

To make the system more reliable, the following strategy was thought. When the student submits the survey, the server interacts with the database as before. The results of the survey, however, are also stored in the cache by the server. If the database is down or unavailable, the survey data is stored locally in a cache component, and the result saved in the cache component is used to provide the result to the student. (This can be done for up to 5 requests, after which the survey cannot be filled.) So, now we have another component in the server called the cache manager. And there is a connection between the server and this new component of the call/return type. This architecture is shown in Figure 4.6.

It should be clear that by using the cache, the availability of the system is improved. The cache will also have an impact on performance. These extensions shows how architecture affects both availability and performance, and how properly selecting or tuning the architecture can help meet the quality goals (or just improve the quality of the