Software Engineering - I

An Introduction to Software Construction Techniques for Industrial Strength Software

Chapter 3 – Requirement Engineering

Requirement Engineering

We recall from our previous discussion that software development is not simply coding – it is a multi-activity process. The process of software construction encompasses and includes answers to the following questions:

- What is the problem to be solved?
- What are the characteristics of the entity that is used to solve the problem?
- How will the entity be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors that were made in the design and construction of the entity?
- How will the entity be supported over the long term when users of the entity request corrections, adaptations, and enhancements?

These questions force us to look at the software development process from different angles and require different tools and techniques to be adopted at different stages and phases of the software development life cycle. Hence we can divide the whole process in 4 distinct phases namely vision, definition, development, and maintenance. Each one of these stages has a different focus of activity. During the vision phases, the focus is on why do we want to have this system; during definition phase the focus shifts from why to what needs to be built to fulfill the previously outlined vision; during development the definition is realized into design and implementation of the system; and finally during maintenance all the changes and enhancements to keep the system up and running and adapt to the new environment and needs are carried out.

Requirement engineering mainly deals with the definition phase of the system. Requirement engineering is the name of the process when the system services and constraints are established. It is the starting point of the development process with the focus of activity on what and not how.

Software Requirements Definitions

Before talking about the requirement process in general and discussing different tools and techniques used for developing a good set of requirements, let us first look at a few definitions of software requirements.

Jones defines software requirements as a statement of needs by a user that triggers the development of a program or system.

Alan Davis defines software requirements as a user need or necessary feature, function, or attribute of a system that can be sensed from a position external to that system.

According to Ian Summerville, requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.

IEEE defines software requirements as:

- 1. A condition or capability needed by user to solve a problem or achieve an objective.
- 2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
- 3. A documented representation of a condition or capability as in 1 or 2.

As can be seen, these definitions slightly differ from one another but essentially say the same thing: a software requirement is a document that describes all the services provided by the system along with the constraints under which it must operate.

Importance of Requirements

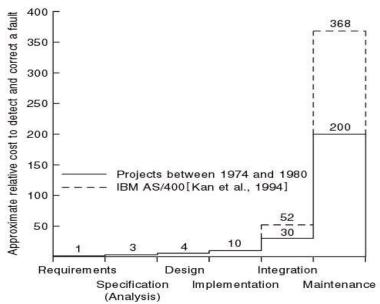
Many of the problems encountered in SW development are attributed to shortcoming in requirement gathering and documentation process. We cannot imagine start building a house without being fully satisfied after reviewing all the requirements and developing all kinds of maps and layouts but when it comes to software we really do not worry too much about paying attentions to this important phase. This problem has been studied in great detail and has been noted that 40-60% of all defects found in software projects can be traced back to poor requirements.

Fred Brooks in his classical book on software engineering and project management "The Mythical Man Month" emphasizes the importance of requirement engineering and writes:

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the system if done wrong. No other part is more difficult to rectify later."

Let us try to understand this with the help of an analogy of a house. If we are at an advanced stage of building a house, adding a new room or changing the dimensions of some of the rooms is going to be very difficult and costly. On the other hand if this need is identified when the maps are being drawn, one can fix it at the cost of redrawing the map only. In the case of a software development, we experience the exact same phenomenon - if a problem is identified and fixed at a later stage in the software development process, it will cost much more than if it was fixed at and earlier stage.

This following graph shows the relative cost of fixing problem at the various stages of software development.



Boehm (1981) has reported that correcting an error after development costs 68 times more. Other studies suggest that it can be as high as 200 times. Since cost is directly related with the success or failure of projects, it is clear from all this discussion that having sound requirements is the most critical success factor for any project.

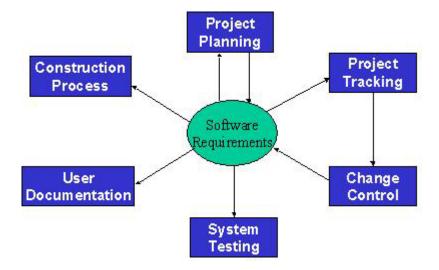
Role of Requirements

Software requirements document plays the central role in the entire software development process. To start with, it is needed in the project planning and feasibility phase. In this phase, a good understanding of the requirements is needed to determine the time and resources required to build the software. As a result of this analysis, the scope of the system may be reduced before embarking upon the software development.

Once these requirements have been finalized, the construction process starts. During this phase the software engineer starts designing and coding the software. Once again, the requirement document serves as the base reference document for these activities. It can be clearly seen that other activities such as user documentation and testing of the system would also need this document for their own deliverables.

On the other hand, the project manager would need this document to monitor and track the progress of the project and if needed, change the project scope by modifying this document through the change control process.

The following diagram depicts this central role of the software requirement document in the entire development process.

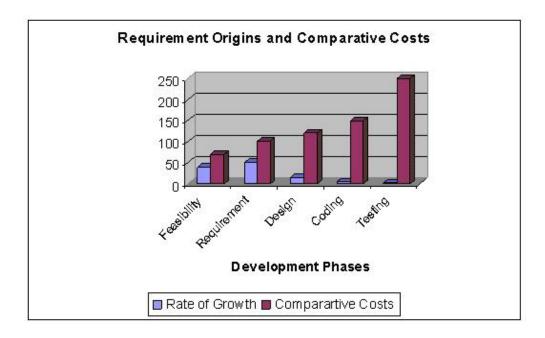


Some Risks from Inadequate Requirement Process

From the above discussion, it should be clear that the requirements play the most significant role in the software development process and the success and failure of a system depends to a large extent upon the quality of the requirement documents. Following is a list of some of the risks of adopting an inadequate requirement process:

- 1. Insufficient user involvement leads to unacceptable products.

 If input from different types of user is not taken, the output is bound to lack in key functional areas, resulting in an unacceptable product. Overlooking the needs of certain user classes (stake holders) leads to dissatisfaction of customers.
- 2. Creeping user requirements contribute to overruns and degrade product quality. Requirement creep is one of the most significant factors in budget and time overruns. It basically means identifying and adding new requirements to the list at some advanced stages of the software development process. The following figure shows the relative cost of adding requirements at different stages.



3. Ambiguous requirements lead to ill-spent time and rework.

Ambiguity means that two different readers of the same document interpret the requirement differently. Ambiguity arises from the use of natural language. Because of the imprecise nature of the language, different readers interpret the statements differently. As an example, consider the following Urdu Phrase: "Rooko mut jane doo". Now, depending upon where a reader places the comma in this statement, two different readers may interpret it in totally different manner. If a comma is palced after "Rooko", the sentence will become "Rooko, mut jane doo", meaning "don't let him go". On the other hand if the comma id placed after "mut", the sentence will become "Rooko mut, jane doo", meaning "let him go". Ambiguous requirements therefore result in misunderstandings and mismatched expectations, resulting in a wasted time and effort and an undesirable product.

Let us consider the following requirement statement:

The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security VDU and deposited in the login file when an operator logs into the system.

This is an example of ambiguous requirement as it is not clear what is meant by "it" in the second sentence and what should be displayed on the VDU. Does it refer to the operator identity as a whole, his name, or his password?

4. Gold-plating by developers and users adds unnecessary features.

Gold-plating refers to features are not present in the original requirement document and in fact are not important for the end-user but the developer adds them anyway thinking that they would add value to the product. Since these features are outside the initial scope of the product, adding them will result in schedule and budget overruns.

5. Minimal specifications lead to missing key requirements and hence result in an unacceptable product.

As an example, let us look at the following requirement. The requirement was stated as: "We need a flow control and source control engineering tool." Based upon this requirement, system was built. It worked perfectly and had all the functionality needed for source control engineering tool and one could draw all kinds of maps and drawings. The system however could not be used because there was there was no print functionality.

Let us now look at the following set of requirement statements for another system:

- The system should maintain the hourly level of reservoir from depth sensor situated in the reservoir. The values should be stored for the past six months.
- AVERAGE: Average command displays the average water level for a particular sensor between two times.

This is another case of minimal requirements – it does not state how the system should respond if we try to calculate the average water level beyond the past six months.

6. Incompletely defined requirements make accurate project planning and tracking impossible.

Levels of Software Requirements

Software requirements are defined at various levels of detail and granularity. Requirements at different level of detail also mean to serve different purposes. We first look at these different levels and then will try to elaborate the difference between these with the help of different examples.

1. Business Requirements:

These are used to state the high-level business objective of the organization or customer requesting the system or product. They are used to document main system features and functionalities without going into their nitty-gritty details. They are captured in a document describing the project vision and scope.

2. User Requirements:

User requirements add further detail to the business requirements. They are called user requirements because they are written from a user's perspective and the focus of user requirement describe tasks the user must be able to accomplish in order to fulfill the above stated business requirements. They are captured in the requirement definition document.

3. Functional Requirements:

The next level of detail comes in the form of what is called functional requirements. They bring-in the system's view and define from the system's perspective the software functionality the developers must build into the product to enable users to accomplish their tasks stated in the user requirements - thereby satisfying the business requirements.

4. Non-Functional Requirements

In the last section we defined a software requirement as a document that describes all the services provided by the system along with the constraints under which it must operate. That is, the requirement document should not only describe the functionality needed and provided by the system, but it must also specify the constraints under which it must operate. Constraints are restrictions that are placed on the choices available to the developer for design and construction of the software product. These kinds of requirements are called Non-Functional Requirements. These are used to describe external system interfaces, design and implementation constraints, quality and performance attributes. These also include regulations, standards, and contracts to which the product must conform.

Non-functional requirement play a significant role in the development of the system. If not captured properly, the system may not fulfill some of the basic business needs. If proper care is not taken, the system may collapse. They dictate how the system architecture and framework. As an example of non-functional requirements, we can require software to run on Sun Solaris Platform. Now it is clear that if this requirement was not captured initially and the entire set of functionality was built to run on Windows, the system would be useless for the client. It can also be easily seen that this requirement would have an impact on the basic system architecture while the functionality does not change.

While writing these requirements, it must always be kept in mind that all functional requirements must derive from user requirements, which must themselves be aligned with business requirements. It must also be remembered that during the requirement engineering process we are in the definition phase of the software development where the focus is on what and not how. Therefore, requirements must not include design or implementation details and the focus should always remain on what to build and not how to build.

Let us now look at an example to understand the difference between these different types of requirements.

Let us assume that we have a word-processing system that does not have a spell checker. In order to be able to sell the product, it is determined that it must have a spell checker. Hence the business requirement could be stated as: *user will be able to correct spelling errors in a document efficiently*. Hence, the Spell checker will be included as a feature in the product.

In the next step we need to describe what tasks must be included to accomplish the above-mentioned business requirement. The resulting user requirement could be as follows: finding spelling errors in the document and decide whether to replace each misspelled word with the one chosen from a list of suggested words. It is important to note that this requirement is written from a user's perspective.

After documenting the user's perspective in the form of user requirements, the system's perspective: what is the functionality provided by the system and how will it help the user to accomplish these tasks. Viewed from this angle, the functional requirement for the same user requirement could be written as follows: the spell checker will find and highlight misspelled words. It will then display a dialog box with suggested replacements. The user will be allowed to select from the list of suggested replacements. Upon selection it will replace the misspelled word with the selected word. It will also allow the user to make global replacements.

Finally, a non-functional requirement of the system could require that *it must be integrated into the existing word-processor that runs on windows platform.*

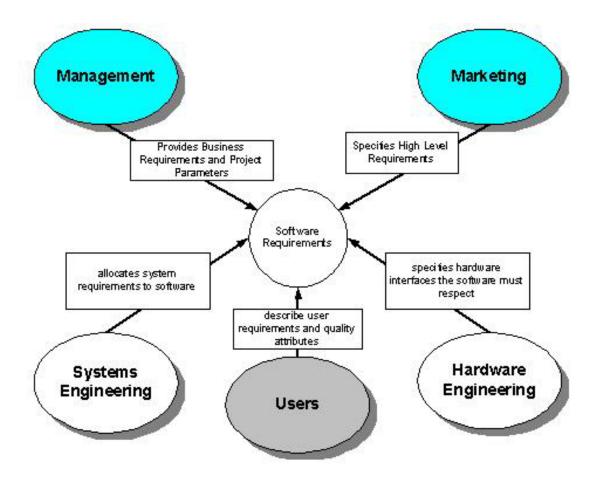
Stakeholders

As mentioned earlier, in order to develop a good requirement document, it is imperative to involve all kinds of user in the requirement engineering process. The first step in fulfillment of this need is the identification of all the stakeholders in the system. Stakeholders are different people who would be interested in the software. It is important to recognize that management carries a lot of weight, but they usually are not the actual users of the system. We need to understand that it is the actual user who will eventually use the system and hence accept or reject the product. Therefore, ignoring the needs of any user class may result in the system failure.

A requirement engineer should be cognizant of the fact that stakeholders have a tendency to state requirements in very general and vague terms. Some times they trivialize things. Different stakeholders have different requirements – sometimes even conflicting. On top of that internal politics may influence requirements.

The role of stakeholders cannot be overemphasized. A study of over 8300 projects revealed that the top two reasons for any project failure are lack of user input and incomplete requirements.

The following diagram shows the role of different stakeholders in the setting the system requirements.



Requirement Statement and Requirement Specification Documents

Different levels of software requirements are documented in different documents. The two main documents produced during this phase are Requirement Statement and Requirement Specification. They are also called Requirement Definition and Functional Specification and are used to document user requirements and functional requirements respectively.

Requirement Statement Characteristics

A good Requirements statement document must possess the following characteristics.

- Complete Each requirement must fully describe the functionality to be delivered.
- **Correct** Each requirement must accurately describe the functionality to be built.
- **Feasible** It must be possible to implement each requirement within the known capabilities and limitations of the system and its environment.

- **Necessary** -Each requirement should document something that the customer really need or something that is required for conformance to an external system requirement or standard.
- **Prioritized** An implementation priority must be assigned to each requirement, feature or use case to indicate how essential it is to a particular product release.
- Unambiguous All readers of a requirement statement should arrive at a single, consistent interpretation of it.
- **Verifiable** User should be able to devise a small number of tests or use other verification approaches, such as inspection or demonstration, to determine whether the requirement was properly implemented.

Requirement Specification Characteristics

A good Requirements specification document should possess the following characteristics.

- **Complete** No requirement or necessary information should be missing.
- **Consistent** No requirement should conflict with other software or higher-level system or business requirements.

Let us try to understand this with the help of some examples. The following set of (non-functional) requirements was stated for a particular embedded system.

- All programs must be written in Ada
- The program must fit in the memory of the embedded micro-controller

These requirements conflicted with one another because the code generated by the Ada compiler was of a large footprint that could not fit into the micro-controller memory.

Following is another set of (functional) requirements that conflicted with one another:

- System must monitor all temperatures in a chemical reactor.
- System should only monitor and log temperatures below -20° C and above 400° C.

In this case the two requirements clearly conflict with each other in stating what information needs to be monitored and stored.

• **Modifiable** - One must be able to revise the Software Requirement Specification when necessary and maintain a history of changes made to each requirement.

• Traceable - One should be able to link each requirement to its origin and to the design elements, source code, and test cases that implement and verify the correct implementation of the requirement.

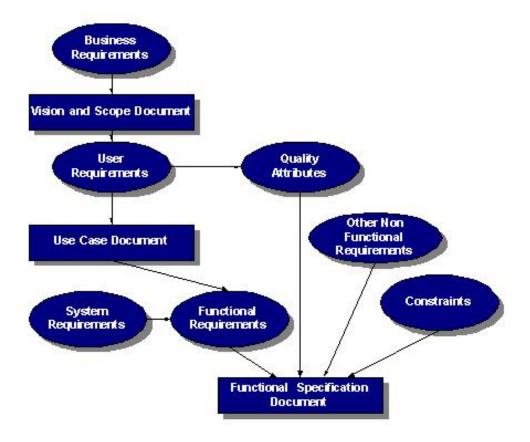
Mixed level of Abstraction

It is important to recognize that all requirements in a requirement document are stated at a uniform level of abstraction. This difference in detail falsely implies the relative importance of these requirements and hence misguides all involved in the development process. The following set of requirements clearly demonstrates violation of this principle:

- *The purpose of the system is to track the stock in a warehouse.*
- When a loading clerk types in the withdraw command he or she will communicate the order number, the identity of the item to be removed, and the quantity removed. The system will respond with a confirmation that the removal is allowable.

Relationship of Several components of Software Requirements

The following figure depicts the relationship between different documents produced during the requirement engineering phase.



Business Requirements

Business requirements collected from multiple sources might conflict. For example, consider a kiosk product with embedded software that will be sold to retail stores and used by the store's customers. The kiosk developer's business objectives include the following:

- leasing or selling the kiosk to the retailers
- selling consumables through the kiosk to the customer
- attracting customer to the brand
- modifying the nature of the historical developer-customer relationship

The retailer's business interest could include:

- making money from customer use of kiosk
- attracting more customers to the store
- saving money if the kiosk replaces manual operations

The developer might want to establish a high-tech and exciting new direction for customers, while the retailer wants a simple solution and the customer wants convenience and features. The tension among these three parties with their different goals, constraints, and cost factors can lead to conflicting business requirements, which must be resolved before the kiosk's software requirements are detailed.

You can also use the business requirements to set implementation priorities for use cases and their associated functional requirements. For example, a business requirement to generate maximum revenue from the kiosk would imply the early implementation of features directly associated with selling more products or services to the customer, rather than glitzy features that appeal to only a subset of customers.

The Vision Statement

The vision statement should reflect a balanced view that will satisfy the need of diverse customers. It can be somewhat idealistic but should be grounded in the realities of existing or anticipated customer markets, enterprise architectures, organizational strategic directions, and resource limitations.

Chemical Tracking System

The chemical tracking system will allow scientists to request containers of chemicals to be supplied by chemical stockroom or by vendors. The location of every chemical container within the company, the quantity of the material remaining in it, and the complete history of each container's location and usage will be known by the system at all times. The company will save 25% on chemical costs by fully exploiting chemicals already available within the company, by disposing of fewer partially used or expired containers, and by using a standard chemical purchasing process. The chemical tracking system will also generate all reports required to comply with federal and state government regulations that require the reporting of chemical usage, storage, and disposal.

Assumptions and Dependencies

All assumptions that were made when conceiving the project have to be recorded. For example, the management sponsor for the chemical tracking system assumed that it would replace the existing chemical stockroom inventory system and that it would interface to the appropriate purchasing department applications

Scope

Project scope defines the concept and range of the proposed solution, and limitations identify certain capabilities that the product will not include. Clarifying the scope and limitations helps to establish realistic stakeholder's expectations. Sometimes customers request features that are too expansive or do not lie within the intended project scope. Propose requirements that are out of scope must be rejected, unless they are so beneficial that the scope should be enlarged to accommodate them (with accompanying changes in budget, schedule, and staff). Keep a record of these requirements and why they were rejected, as they have a way of reappearing.

Scope and Initial Release

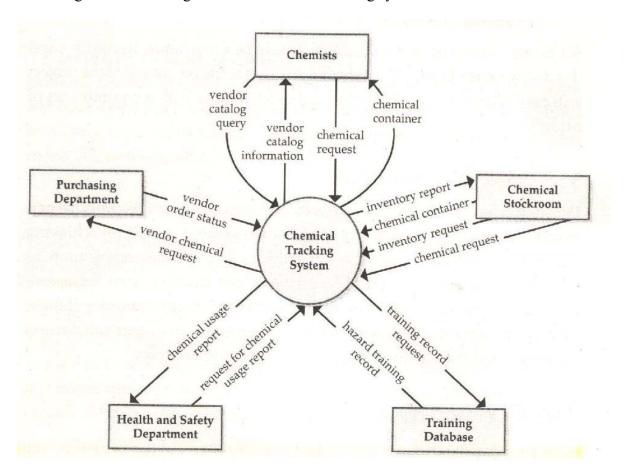
The major features that will be included in the initial release of the project should be summarized. Describe the quality characteristics that will enable the product to provide the intended benefits to its various customer communities.

Requirements need to be prioritized and a release schedule should be made.

The Context Diagram

The scope description establishes the boundary between the system we are developing and everything else in the universe. The context diagram graphically illustrates this boundary by showing the connections between the system being developed or the problem being addressed, and the outside world. The context diagram identifies the entities outside the system that interface to it in some way (called terminators or external entities), as well as the flow of data and material between each external entity and the system. The context diagram is used as the top level abstraction in a dataflow diagram developed according to principles of structured analysis. The context diagram can be included in the vision and scope document, in the SRS, or as part of a dataflow model of the system.

Following is a context diagram of the chemical tracking system.



Use Cases and Customer-Developer Relationship

It has been mentioned earlier on, excellent software products are the result of a well-executed design based on excellent requirements and high quality requirements result from effective communication and coordination between developers and customers. That is, good customer-developer relationship and effective communication between these two entities is a must for a successful software project. In order to build this relationship and capture the requirements properly, it is essential for the requirement engineer to learn about the business that is to be automated.

It is important to recognize that a software engineer is typically not hired to solve a computer science problem – most often than not, the problem lies in a different domain than computer science and the software engineer must understand it before it can be solved. In order to improve the communication level between the vendor and the client, the software engineer should learn the domain related terminology and use that terminology in documenting the requirements. Document should be structured and written in a way that the customer finds it easy to read and understand so that there are no ambiguities and false assumption.

One tool used to organize and structure the requirements is such a fashion is called use case modeling.

It is modeling technique developed by Ivar Jacobson to describe what a new system should do or what an existing system already does. It is now part of a standard software modeling language known as the Unified Modeling Language (UML). It captures a discussion process between the system developer and the customer. It is widely used because it is comparatively easy to understand intuitively – even without knowing the notation. Because of its intuitive nature, it can be easily discussed with the customer who may not be familiar with UML, resulting in a requirement specification on which all agree.

Use Case Model Components

A use case model has two components, use cases and actors.

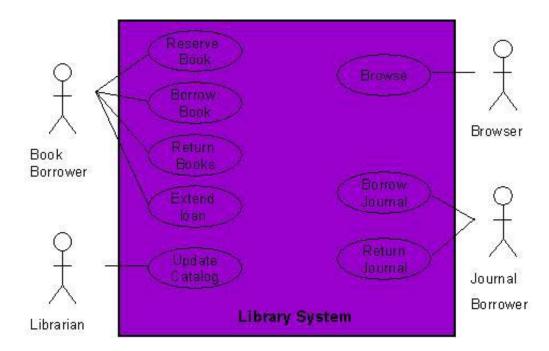
In a use case model, boundaries of the system are defined by functionality that is handled by the system. Each use case specifies a complete functionality from its initiation by an actor until it has performed the requested functionality. An actor is an entity that has an interest in interacting with the system. An actor can be a human or some other device or system.

A use case model represents a use case view of the system – how the system is going to be used. In this case system is treated as a black box and it only depicts the external interface of the system. From an end-user's perspective it and describes the functional requirements of the system. To a developer, it gives a clear and consistent description of what the system should do. This model is used and elaborated throughout the

development process. As an aid to the tester, it provides a basis for performing system tests to verify the system. It also provides the ability to trace functional requirements into actual classes and operations in the system and hence helps in identifying any gaps.

Use Diagram for a Library System

As an example, consider the following use case diagram for a library management system. In this diagram, there are four actors namely Book Borrower, Librarian, Browser, and Journal Borrower. In addition to these actors, there are 8 use cases. These use cases are represented by ovals and are enclosed within the system boundary, which is represented by a rectangle. It is important to note that every use case must always deliver some value to the actor.



With the help of this diagram, it can be clearly seen that a Book Borrower can reserve a book, borrow a book, return a book, or extend loan of a book. Similarly, functions performed by other users can also be examined easily.

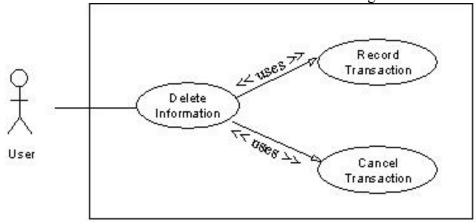
Creating a Use Case Model

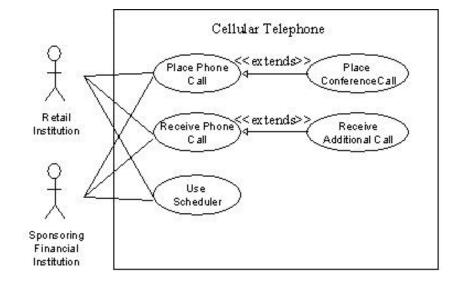
Creating a use case model is an iterative activity. The iteration starts with the identification of actors. In the next step, use cases for each actor are determined which define the system. After that, relationships among use cases are defined. It must be understood that these are not strictly sequential steps and it is not necessary that all actors must be identified before defining their use cases. These activities are sort of parallel and concurrent and a use case model will evolve slowly from these activities. This activity stops when no new use cases or actors are discovered. At the end, the model is validated.

Relationship among Use Cases

The UML allows us to extend and reuse already defined use cases by defining the relationship among them. Use cases can be reused and extended in two different fashions: extends and uses. In the cases of "uses" relationship, we define that one use case invokes the steps defined in another use case during the course of its own execution. Hence this defines a relationship that is similar to a relationship between two functions where one makes a call to the other function. The "extends" relationship is kind of a generalization-specialization relationship. In this case a special instance of an already existing use case is created. The new use case inherits all the properties of the existing use case, including its actors.

Let is try to understand these two concepts with the help of the following diagrams. In the case of the first diagram, the *Delete Information* use case is using two already existing use cases namely *Record Transaction* and *Cancel Transaction*. The direction of the arrow determines which one is the user and which use case is being used.

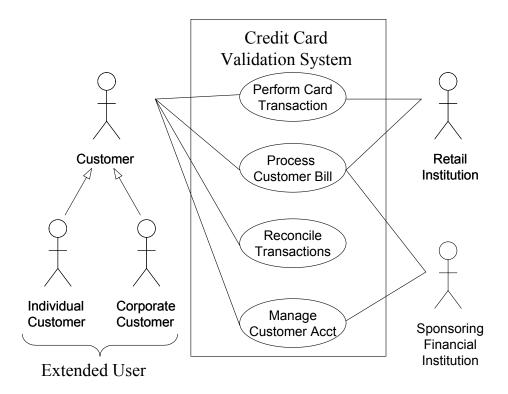




The second diagram demonstrates the concept of reuse by extending already existing use cases. In this case *Place Conference Call* use case is a specialization of *Place Phone Call* use case. Similarly, *Receive Additional Call* is defined by extending *Receive Phone Call*. It may be noted here that, in this case, the arrow goes from the new use case that is being created (derived use case) towards the use case that is being extended (the base use case).

This diagram also demonstrates that many different actors can use one use case. Additionally, the actors defined for the base use case are also defined by default for the derived use case.

The concept of reusability can also be used in the case of actors. In this case, new classes of actors may be created by inheriting from the old classes of actors.



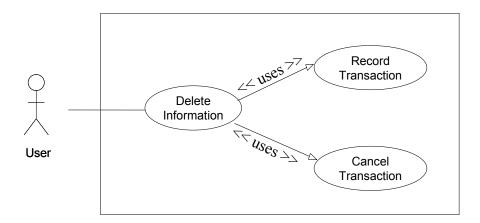
In this case two new classes, *Individual Customer* and *Corporate Customer*, are being created by extending *Customer*. In this case, all the use cases available to *Customer* would also be available to these two new actors.

Elaborated Use Cases

After the derivation of the use case model, each use is elaborated by adding detail of interaction between the user and the software system. An elaborated use case has the following components:

- Use Case Name
- Implementation Priority: the relative implementation priority of the use case.
- Actors: names of the actors that use this use case.
- Summary: a brief description of the use case.
- Precondition: the condition that must be met before the use case can be invoked.
- Post-Condition: the state of the system after completion of the use case.
- Extend: the use case it extends, if any.
- Uses: the use case it uses, if any.
- Normal Course of Events: sequence of actions in the case of normal use.
- Alternative Path: deviations from the normal course.
- Exception: course of action in the case of some exceptional condition.
- Assumption: all the assumptions that have been taken for this use case.

As an example, the *Delete Information* use case is elaborated as follows:



Use Case Name: Delete Information

Priority: 3

Actors: User

Summary: Deleting information allows the user to permanently remove information from the system. Deleting information is only possible when the information has not been used in the system.

Preconditions: Information was previously saved to the system and a user needs to permanently delete the information.

Post-Conditions: The information is no longer available anywhere in the system.

Uses: Record Transactions, Cancel Action

Extends: None

Normal Course of Events:

- 1. The use case starts when the user wants to delete an entire set of information such as a user, commission plan, or group.
- 2. The user selects the set of information that he/she would like to delete and directs the system to delete the information. Exception 1, 2
- 3. The system responds by asking the user to confirm deleting the information.
- 4. The user confirms deletion.
- 5. Alternative Path: Cancel Action
- 6. A system responds by deleting the information and notifying the user that the information was deleted from the system.
- 7. Uses: Record Transaction
- 8 This use case ends

Alternative Path - The user does not confirm Deletion

- 1. If the user does not confirm deletion, the information does not delete.
- 2. Uses: Cancel Action

Exceptions:

- 1. The system will not allow a user to delete information that is being used in the system.
- 2. The system will not allow a user to delete another user that has subordinates.

Assumptions:

- 1. Deleting information covers a permanent deletion of an entire set of data such as a commission plan, user, group etc. Deleting a portion of an entire set constitutes modifying the set of data.
- 2. Deleted information is not retained in the system.
- 3. A user can only delete information that has not been used in the system.

Alternative Ways of Documenting the Use Case

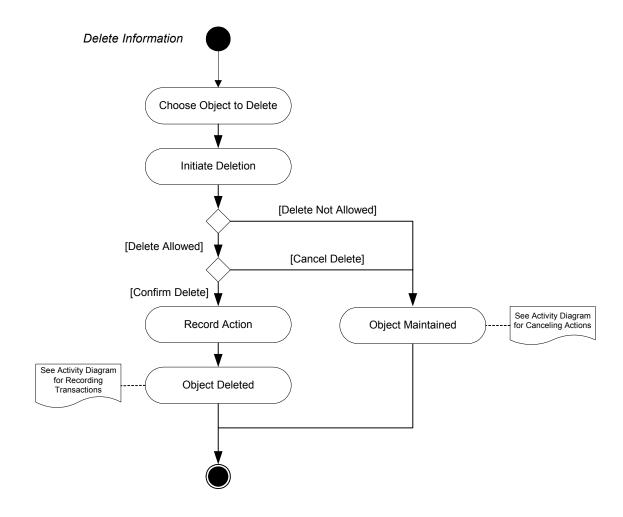
Many people and organizations prefer to document the steps of interaction between the use and the system in two separate columns as shown below.

User Action	System Reaction
1. The use case starts when the user wants	
to delete an entire set of information	
such as a user, commission plan, or	
group.	
2. The user selects the set of information	3. The system responds by asking the user
that he/she would like to delete and	to confirm deleting the information.
directs the system to delete the	
information Exception 1, 2	
4. The user confirms deletion.	5. A system responds by deleting the
	information and notifying the user that
	the information was deleted from the
	system.

It is a matter of personal and organizational preference. The important thing is to write the use case in proper detail.

Activity Diagrams

Activity diagrams give a pictorial description of the use case. It is similar to a flow chart and shows a flow from activity to activity. It expresses the dynamic aspect of the system. Following is the activity diagram for the *Delete Information* use case.



Limitations of Use Cases

Use cases alone are not sufficient. There are kinds of requirements (mostly non-functional) that need to be understood. Since use cases provide a user's perspective, they describe the system as a black box and hide the internal details from the users. Hence, in a use case, domain (business) rules as well as legal issues are not documented.

The non-functional requirements are also not documented in the use cases. As examples of those, consider the following requirements.

- Usability
 - Color blind people should not have any difficulty in using the system color coding should take care of common forms of color blindness.
- Reliability
 - o The system needs to support 7 x 24 operation
- Performance
 - o Authorization should be completed within 1 minute 90% of the time.
 - o Average authorization confirmation time should not exceed 30 seconds.
- Portability
 - The system should run on Windows 98 and above as well as Sun Solaris 7.0 and above.
- Access
 - System should be accessible over the internet hidden requirement security

Because of this shortcoming, use cases must be augmented by additional information.

Source and sink analysis

Once requirements are documented using any of these analysis models, an independent verification is needed to verify completeness and consistency of requirements captured through these models. The process of verifying requirements involves careful analysis of sources as well as the sinks of information.

Source

A stakeholder describes requirements (needs, constraints) to be included as system functionality. These can be processes that generate certain information that the system may have to process or maintain. Sources of requirements are the origins from where the corresponding business process is initiated. By this concept, one has to trace from a requirement back to its origins to see who is involved in its initiation. Be it a person, an organization or an external entity that initiate some action and system responds back by completing that action.

Sink

Sink is the consumer of certain information. It is that entity which provides a logical end to a business process. Thus, 'sinks of requirements' is a concept that helps in identifying persons, organizations or external systems that gets certain functionality from the system. These are logical ends of requirements, or where all the requirements are consumed. For example, we may consider a user of a software application that retrieves a report from the system. In this case, user when reviews the report, becomes the sink of that report. Thus when analyzing the sink of the requirement of implementing a report, the analyst would naturally point towards the user who would get that report.

In source and sink analysis the analyst determines all the sources of requirements and where do these requirements consume (sinks). Now evaluate a report which displays certain information, the source of this report is the data (and who enters it) that is input to be retrieved later in the form of the report. Similarly, whoever needs this report become the sink of the report.

In a similar manner, at times we gather data in our application that is not used anywhere. So the question really is what to do with that kind of unused data or the missing requirement. Is it really redundant or is something really missing from these requirements? How to figure it out?

For example, we are having certain inputs (sources) to a process against which we do not know about the corresponding outputs (sinks). Such inputs are redundant if there is found no corresponding outputs. Thus these inputs can be removed as redundant. If we probe out corresponding outputs, which could not be recorded initially, that mean these inputs were not redundant rather a few (output related) requirements were missing that we discovered during the sink analysis.

A stakeholder may have required the development team to develop certain report for his use. It means we are sure of its use (sink) but not about its sources, from where the

Software Engineering-I (CS504)

required information will be provided? Who will input that information and using what mechanism?

A requirement statement that describe the report but do not list down its sources, will be an incomplete statement and the software engineer who is involved in validating such requirements, should identify all the sources against sinks or vice versa to determine complete end-to-end requirements.