CHAPTER 2

REQUIREMENTS ENGINEERING

2.1 INTRODUCTION

Requirements Engineering is concerned with defining the problem that will be solved by a software system. It can be considered as the foundation on which a software product is built. Below are some representative definitions of Requirements Engineering.

Lamsweerde says that “Requirements Engineering is concerned with the elicitation of high-level goals to be achieved by the envisioned system, the refinement of such goals and their operationalization into specifications of services and constraints, and the assignment of responsibilities for the resulting requirements to such as humans, devices and software.” Nuseibeh offers another perspective: “Broadly speaking, Software Systems Requirements Engineering is the process of discovering the purpose for which the Software System is intended, by identifying stakeholders and their needs, documenting these in a form that is amenable to analysis, communication and subsequent implementation.” Zave includes factors external to the software product: “Requirements Engineering is the branch of Software Engineering concerned with the real world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families.” Ross and Schuman emphasized the practical nature of Requirements Engineering, when they
wrote that Requirements Engineering is “A careful assessment of the needs that a system should fulfill. It must say a system is needed, based on current and foreseen conditions, which may be internal operations or an external market. It must say what the system features will serve and satisfy this context. And it must say how the system is to be constructed.” Brooks adds yet another dimension to the understanding of Requirements Engineering when talking about the “iterative extraction and refinement of the product requirements.”

The key words highlighted in the above definitions are representative of the varied scope of Requirements Engineering. Requirements Engineering is about High-level Goals, operationalization, assignment of responsibilities to humans, devices and software, stake holder’s needs, environment and constraints under which the software should operate, hard practical actions to deliver solutions, iteratively getting to the desired solution.

The earliest approaches to software development were ad hoc or programming centered, which considered development of software primarily as a programming effort. Such approaches resulted in the projects being late, costly, and unreliable. Moreover, the maintenance on the delivered product was significant – adding additional expenses to the project.

In the 1960s the introduction of third-generation computer hardware enabled large software systems to be built (Sommerville 1996). The problems encountered in building large software systems were not simply scaled up versions of the problems of developing small computer programs. As a result, the cost of building large software systems increased dramatically (Bell 1976). One answer to the increased cost of software development was “software engineering”, first introduced in the late 1960s at a conference held
to discuss the “software crisis” (Sommerville 1996). Software engineering is formally defined as (IEEE 1987):

...a systematic approach to the development, operation, maintenance, and retirement of software.

The objective of software engineering is to identify methods and procedures for the development of software that can scale up for large projects, and be used to consistently produce high-quality software at low cost (Jalote 1999). Thus, the key objectives are consistency, low cost, high quality, and scalability.

The initial model of the Software Development Life Cycle (SDLC) was the waterfall model produced by Royce (1970). Since then, several approaches such as prototyping and spiral have improved upon the waterfall model. This has resulted in more robust methodologies for the development of software. In spite of the differences in the approaches to the SDLC, most of the models encompass the same “macro” development phases – requirements analysis, software design, coding and unit testing, system testing, and maintenance (Figure 2.1).

![Figure 2.1 Main phases in the SDLC](image)

Despite the number of approaches proposed, software development was still facing the earlier problems of cost and schedule delays because the phases of the models lacked the necessary support in terms of guidelines,
tools and methods. As a result, emphasis was placed on improving the individual phases. In spite of being “first” in the sequence of the SDLC phases, requirements analysis has been the last to be re-examined and refined by the software engineering community (Sidky 2002). The refinement of the phases proposed by Royce took place in a backward direction starting with testing. Several tools, methods, and guidelines were developed for the testing phase, such as black box and white box testing techniques. Coding was also enhanced with the integrated development environment (IDE), coding guidelines and error detectors, and a number of other tools. The design phase was improved with notations such as UML and supporting tools. However, in spite of these advances in the various SDLC phases, the product often failed to meet the user’s intent. The main reason was the lack of a requirements generation framework and suitable guidelines. It is only in recent years that the importance of requirements has been recognized and efforts are being made to overcome the problems in the requirements analysis phase. Moreover, since the name requirements analysis was inappropriate to cover all requirements activities, it was changed to requirements engineering, which includes the activities – elicitation, analysis, specification, verification and management. The following section highlights the importance of requirement engineering in the SDLC.

2.2 IMPORTANCE OF REQUIREMENTS ENGINEERING

Success in software development is measured by the quality of the product delivered to the customer. Requirements engineering is a critical phase for the success of a project since this phase ensures that the software system reflects the customer needs. The requirements engineering phase focuses on deciding precisely what to build and this is the most difficult part of building a software system (Brooks 1987).
A good set of requirements cannot be obtained by an *ad hoc* process; instead, they need to be engineered through a systematic and well-defined process (Bell 1976). Therefore, as mentioned in the previous section, the requirements analysis phase in the SDLC has evolved into the requirements engineering, which is defined as “a systematic process of developing requirements through an iterative co-operative process of: analyzing the problem, documenting the resulting observations in a variety of representation formats, and checking the accuracy of the understanding gained” (Macaulay 1996).

The objective of the requirements engineering phase is to obtain a complete and clear software requirements specification (SRS), which captures the user’s intent. However, if the requirements engineering process fails to generate quality requirements, the potential impact is substantial. Errors committed during the requirements phase often remain latent and are not detected until well after the stage in which they are made. The later in the development life cycle that a software error is detected, the more expensive it is to repair.

In addition to increasing cost and schedule, errors in the requirements often lead to the development of a wrong product, which does not meet the user expectations. A number of studies have been conducted to quantify the impact of requirements errors in the software development process. For example, three companies (GTE, TRW, and IBM) performed independent studies to measure the expenses of repairing errors in different phases of the SDLC. Even though these studies were conducted independently, they all reached roughly the same results depicted in Figure 2.2 (Davis 1993). The figure illustrates that the cost to fix requirements errors increase rapidly in the later development phases. The relative costs are
calculated by making the assumption that a unit cost of detecting and repairing a software error in the coding phase is assigned a value of one.

The surveys show a very important fact that detecting an error in the requirements phase could save a software repair cost 200 times than if it is detected in the maintenance phase.

![Relative cost to repair a defect at difference software life cycle phases](image)

**Figure 2.2 Relative cost to repair a defect at difference software life cycle phases**

In recent years, an increased awareness of requirements engineering in the software industry is becoming more apparent because of certain alarming facts. For instance, the CHAOS report by the Standish Group reveals the significance of the requirements engineering in the software development. According to the CHAOS report, three top challenging factors in the software development are lack of user input, incomplete requirements specifications and changing requirements (Standish 1995). Furthermore, the report emphasizes the importance of involving users in the requirements generation process, and clearly stating requirements as part of top project success factors.
Another large-scale survey conducted by the ESPITI (European Software Process Improvement Training Initiative) indicates that two largest problems in the software industry are (1) requirements specifications, and (2) managing customer requirements (ESPITI 1995). Detailed results are indicated in Figure 2.3.

![Figure 2.3 Major problems in software development](image)

Over the past few years, the software industry has realized the importance of requirements engineering, and a number of research efforts are being directed towards generating quality requirements.

### 2.3 CURRENT DEVELOPMENTS IN REQUIREMENTS ENGINEERING

The requirements engineering process is divided into five major activities, and current research is focused on improving each individual activity as well as integration among them. A brief description of the requirement engineering activities adapted from (Thayer 1997) are listed below:
- Requirements elicitation – The customers and developers of a software system discover, review, articulate, and understand the user needs and constraints on the software and the development activity.

- Requirements analysis – The process of analyzing the customer and user needs to arrive at a definition of software requirements. The customer assesses the acceptable level of risks regarding completeness, correctness, and technical and cost feasibility.

- Requirements specification – The requirements elicited and analyzed in the preceding activities are documented in the form of a formal document, often referred to as the Software Requirements Specification (SRS).

- Requirements verification – The process of ensuring that the requirements elicited and documented in the SRS comply with the system requirements and customer needs. The requirements are also verified for conformance to document standards and adequate basis for the architectural (preliminary) design phase.

- Requirements management – It is “a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system” (Leffingwell 2000). The management activity pervades the entire requirements engineering process ensuring the planning and control of the requirements elicitation, analysis, specification, and verification.
Significant research is being conducted in the requirements engineering field; however, the industry still fails to produce quality requirements. Errors attributed to the SRS are typically incorrect facts, omissions, inconsistencies, and ambiguities (Figure 2.4) (Basili 1981).

![Figure 2.4 Distribution of errors in the SRS](image)

The reasons for incorrect and incomplete requirements are mainly due to the intrinsic problems in the requirements generation process. For example, communication with the customer/user is vital in the requirements generation process because the requirements engineer has a responsibility to bridge the user application domain and the solution domain. Communicating in natural language can easily lead to ambiguity and/or misunderstanding. Often times the requirements engineer does not have the adequate knowledge about the problem domain; therefore, understanding user domain terminologies and ensuring that the real user needs are reflected in the software requirements is challenging. Furthermore, a lack of user participation has been a major problem (Standish 1995) due to both users and
developers having insufficient understanding about the importance of user input in generating quality requirements, which in turn drives the success of the software system.

The Researchers have been improving the requirements engineering process through various solutions. Developments have been made in the individual phases of the requirements generation process, such as more effective elicitation techniques (Goguen 1993), risk analysis (Davis 1999), verification and validation techniques (McGraw 1997), and so on. In this section, we give a brief overview on current approaches that address the above-mentioned problems. Well-known methodologies to coordinate user participation in the requirements elicitation process are Participatory Design (PD) and Joint Application Design (JAD).

Both approaches concentrate on facilitated interaction between users and developers, but they differ in participant selection and facilitator roles. PD and JAD are valuable approaches for requirements elicitation because they focus on group discussions and emphasizing customer input. However, they lack the holistic approach that encompasses the entire requirements engineering process, as well as guidelines for the recording of requirements and ultimately generating the SRS.

The importance of understanding the application domain and analyzing the root cause of the problem is noted by both (Davis 1993) and (Leffingwell 2000). The problem analysis process includes stating the problem, understanding root causes, and defining the system boundary. Although the detailed steps of the problem analysis process and survey of techniques are provided, the approaches lack the integration of problem analysis to other parts of the requirements generation process. Moreover, the
models do not provide guidelines on documents produced during the problem analysis and their content and format.

Documenting requirements is vital throughout the requirements engineering process to ensure correct and complete representation of requirements, and an adequate basis for the future developments of the system. The requirements engineer has a challenging task to incorporate multiple viewpoints from different users and document communicated ideas in an unambiguous and comprehensive way. Two types of requirements representation are (1) natural language and (2) formal specifications. Expressing information in a natural language is easy to comprehend for all parties involved in the development project, but the information expressed can be ambiguous, resulting in multiple interpretations of a particular document. On the other hand, a formal specification overcomes the ambiguity and correctness verification. However, formal techniques are complicated to apply for developers and difficult to understand for the customers (Sutcliffe 2002). Given the two approaches of representing requirements, natural language is most commonly used. The problems inherent in the natural language description can be satisfactorily overcome with proper guidelines for content and format, and templates for the documents. These aspects are addressed in this thesis.

Most of the current research is focused on individual areas of the requirements engineering process. There is, therefore, the need for a framework that organizes and integrates all requirements generation activities, including elicitation, analysis, specification, verification and validation, and management. Several models have been recently developed, including the RGM (Arthur 1999), the Knowledge-Level Process Model (Herlea 1999), the Requirements Triage (Davis 1999), and the Win-Win Spiral Model (Beohm 1998). Each of these models provides structure to the requirements generation
process based on a particular approach emphasizing certain areas such as systematic elicitation, importance of scenarios, detailed risk analysis, and requirements negotiation. A significant point often overlooked by these models is the importance of well-defined requirements documents and their evolution throughout the requirements generation process.

2.4 REQUIREMENTS DEFINITION AND CLASSIFICATION

2.4.1 Definitions

The literature provides a number of definitions of the term “requirement”. IEEE defines requirements as (Macaulay 1996):

1. A condition or capacity needed by a 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 documents.

This definition highlights two important characteristics:

1. Requirements are reflections of the user needs.
2. Requirements are abstract descriptions of what the system will do without referring to how it will accomplish it.

The literature includes several other definitions (Thayer 1997, Sommerville 1996, Lawrence 1996) and so on.
A distinctive definition of “requirement” has been proposed by Jackson (1995):

A requirement is a desired relationship among the phenomena of the problem context, to be satisfied by the software/machine.

According to Jackson, software development is synonymous with building a machine by describing it. He defines requirements as a phenomenon of the application domain, not the solution/machine domain (Jackson 1995). Hence, the requirements specification describes the external behavior of the machine, including requirements of the shared phenomena of the application domain and the machine. Therefore, it is necessary to understand both the application domain (problem domain) and machine domain (solution domain) to generate a well-defined requirements specification. The relationships between the requirements, specifications, and both machine and application domain phenomena are illustrated in Figure 2.5.

![Diagram of relationships between application and machine domains]

Figure 2.5 Relationships of requirements, specifications, and domains

In the process of deriving the requirements specification, a number of requirements documents, which are various representations of
requirements, are generated and analyzed. *Requirements evolution* is a process of transforming the content and format of requirements documents as we progress through the requirements engineering process.

### 2.4.2 Classifications of Requirements

Several classifications of requirements have been proposed in the requirements engineering literature. Here, we present a brief overview of how requirements are classified. IEEE classifies requirements into five categories – functional, interface, performance, quality, and design constraints (IEEE 1990). In addition, different categorizations are proposed by Davis (1990), Leffingwell (2000) and Lauesen (2002). All these classifications of requirements can be condensed into a single categorization comprising of two types of requirements:

- **Functional requirements** – also referred to as behavioral requirements, functional requirements define precisely what inputs the system expects, what outputs will be generated by the software, and the details of the transformational function existing between those inputs and outputs (Davis 1990). Some authors classify the input and output requirements as separate from the functions and refer to them as *data requirements* (Lauesen 2002).

- **Non-functional requirements** – define the overall qualities or characteristics of the resulting system. They place restrictions on the product being developed, the development process, and they also specify external constraints that the product must meet (Kotonya 1998). Non-functional requirements can be further divided into three sub categories – product, process,
and external requirements (Figure 2.6). While product requirements detail the desired characteristics a system or subsystem must possess, process requirements specify constraints on the development process. External requirements are constraints derived from the environment in which the system is developed (Kotonya 1998).

Figure 2.6 Categories of non-functional requirements

Another way of classifying requirements is according to the level of satisfying the customers (Zultner 1992).

- Normal requirements – The objectives and goals of a system that are elicited during meetings with the customer/user. If these requirements are implemented, the customer is satisfied.

- Expected requirements – These are often so fundamental and basic that the customers may not mention them. Their absence causes a significant dissatisfaction.
Exciting requirements – These features are beyond the customer expectations, and their presence is very pleasing.

All types of requirements presented in this section need to be generated through a structured process to ensure completeness and correctness. The subsequent sections discuss how requirements generation and documentation is addressed in the existing SDLC models and the requirements generation models.

2.5 THE SDLC MODELS

SDLC spans the complete product life cycle, beginning with the initial conception of the idea, and continuing through to the day the product is withdrawn from the market or service. Ever since Royce introduced the first SDLC model in 1970 (Royce 1970), a number of models have been proposed to improve the software engineering process. This section presents a brief description of the industry accepted SDLC models and concentrates on how these models address the issue of requirements generation and documentation.

2.5.1 Waterfall Model

The waterfall model (Royce 1970) is known as the basic or “traditional” software engineering model. The model proposes a linear, sequential approach to software development consisting of five phases - analysis, design, coding, testing, and maintenance as shown in Figure 2.7.
Figure 2.7 Waterfall model

Requirements analysis is the first phase in the waterfall model, and its main objective is determining the scope of the problem and identifying the requirements of the system. The analysis phase begins with a careful study of the project feasibility (Jalote 1999). Project feasibility is followed by a thorough analysis of the problem and solution space to determine the requirements. The requirements engineer must have a good understanding of the application domain in order to document the required functions, interface, and constraints in the SRS, the final work product of this phase. It is necessary to verify and validate the SRS to ensure that the specification is complete and correct.

The waterfall model provides the much needed structure and framework for successful software development. In addition, this model emphasizes the importance of verification by stipulating the verification activity at the end of each phase. On the downside, this model is criticized because of its non-iterative nature which fails to reflect the real world software development scenario (Hanna 1995). However, in spite of the
criticisms of the waterfall model for being inflexible, variations of this model are still the most popular in the industry (Neill 2003, Holt 1997).

In the waterfall model, the only requirements document generated is the SRS. Even though the model emphasizes understanding the application domain and eliciting required functions from the users, it does not provide guidelines on how to document them. In addition, this model performs V and V only on the completed SRS, resulting in longer period of checking and correcting the specification.

2.5.2 Prototyping Model

Prototyping evolved out of the need to overcome weaknesses of the waterfall model and to obtain user feedback early in the development process by conducting user evaluations of a “quick” implementation of the system. As depicted in Figure 2.8 prototyping begins by gathering initial requirements. Meetings between the developer and customer are conducted to determine overall system objectives, functions, and performance. The preliminary version of the requirements specification document is produced based on the elicitation (Gomma 1981). The developer then uses a set of tools to implement a quick design and a working model (prototype) in order to understand the proposed system, or certain aspects of it, and to clarify requirements (Maude 1991). Prototypes have little or no underlying functionality, with the sole purpose being to capture user feedback in a short time frame. Minimal cost and time is spent to design and refine the prototype since it is not used as a part of the final system (Brooks 1995). Therefore, the process is often referred to as rapid prototyping. The users evaluate the functions and recommends changes to reflect their actual needs. The requirements phase is iterative, and emphasizes redesigning the prototype until an acceptable model is derived and the SRS is generated. Based on the
prototype and the SRS, the developer builds the real product by applying the steps described in the waterfall model.

Figure 2.8 Rapid prototyping model

An alternative way of using prototypes is evolutionary prototyping, where the prototype is iteratively refined and becomes the final product. Thus, the prototypes are not discarded as in the rapid prototyping process. Through multiple iterations, the prototype is expanded and polished to obtain the product satisfying the user’s needs. A difficulty of evolutionary prototypes is keeping the code efficient and error-free, while continuously adding new code or making changes to the existing code.

The prototyping model emphasizes understanding the user requirements, especially user interface and transaction-oriented functions, to generate a quality SRS (Dorfman 1997). Prototyping allows the software engineer to refine requirements that are not well understood. Therefore, the prototyping approach addresses the volatile nature of the requirements, and through iterations, reduces risk of developing the wrong product (Jalote 1999). Furthermore, prototyping enables the creation of a specification with precise details about the user interface and functionality of the system. In addition, prototyping is also very useful for eliciting requirements for systems which have not been identified earlier.
However, demonstrating an early design without analyzing requirements in detail can cause unnecessary design constraints. Another inherent problem is when the user sees the prototype, what appears to be a fully working system, he/she mistakenly believes that the prototype can be easily transformed into a final product. Therefore, users tend to demand a short duration for releasing the working product. Moreover, a risk of producing “spaghetti code” is apparent in the development is evolutionary prototypes. Developers often make implementation compromises in order to get a working prototype rapidly. This may result in inappropriate design decisions and inefficient algorithms. Thus, effective management of the prototyping model is crucial.

### 2.5.3 Incremental Model

The incremental development combines features of the sequential model with the iterative philosophy of prototyping. The software product is designed, implemented, integrated, and tested as a series of incremental deliverables which build on the previous increments of the software (McDermid 1993). The first increment is usually the core product, which implements only the basic requirements (Pressman 2001). The subsequent increments implement the additional features of the system. After the development of each increment, the product is evaluated, and a plan for the next increment is created. The plan addresses the modification of the core product and the delivery of additional features and functionality. This process is iterated following the implementation of each increment, until the complete product is developed. The iterative behavior of the incremental model is illustrated in Figure 2.9.
The requirements engineering phase in the incremental model produces a single requirements document – the SRS. In the original concept of the incremental model, requirements are assumed to be stable (Dorfman 1997). However, this concept is altered by allowing the SRS to be modified in each increment, since in practice requirements in the later phases can be changed based on evaluation of earlier increments or technology advancement (Dorfman 1997). Therefore, the SRS is produced in increments, with the initial specification containing detailed requirements to be implemented in the first increment. Features to be included in the subsequent increments are noted in the SRS but not detailed. After each increment, the feedback obtained from the users is collected and analyzed for revision to the SRS. In addition, new features can be included as a result of the user needs.

The incremental model emphasizes continuous user feedback to obtain the SRS. Even though this model is iterative like the prototyping model, the incremental model focuses on the delivery of an operational product with each increment. This model is specially suited for situations where the required staff is not available for a complete implementation by the
deadline established from business needs (Pressman 2001). The incremental model lowers the risk of project failure by allowing partial delivery of the system. Determining the number of iterations and its duration is very important to the success of the project. If the software product is broken down into too few builds, the model becomes a build-and-fix approach. On the other hand, if the project has too many builds, it can incur excessive overhead. In addition, the requirements change management process is not explicitly represented in this model.

2.5.4 **Spiral Model**

The spiral model as defined by Boehm (1988) consists of six task regions: customer communication, planning, risk analysis, engineering, construction and release, and customer evaluation. Each phase in the development life cycle iterates one or more times through the six sections. The objectives of each region are given below (Pressman 2001):

- Customer communication – determines the scope and requirements with active participation of the customer.
- Planning – determines objectives and possible solution approaches.
- Risk analysis – along with the customer, determines the risk involved in each alternate solution approach.
- Engineering – creates the design of the system and describes the necessary algorithms.
- Construction and release – codes the application based on the design, and conducts testing and release.
- Customer evaluation – determines if product is satisfactory and what changes or changes are necessary.
The spiral model (Figure 2.10) combines the best features of the waterfall and prototype models, and includes the evaluation of risk. Before each development phase begins, the risks involved in the project are analyzed and mitigated, if possible. If the risks cannot be resolved, then the project may be discontinued by the decision of the management (Schach 1996).

Figure 2.10  Spiral model

The spiral model uses prototyping as a means of reducing the risk in the project. Each of the cycles in the spiral model is iterative and can be executed several times until the objectives are achieved. The SRS is the final product of the requirements phase, and it is reviewed and validated several times until it is complete. In the initial cycle, only the known requirements are
elicited and prototyped. The risks are evaluated and the plan for the next iteration is prepared. Before conducting the next cycle, the requirements are validated by the customer, and the feedback is used to make any needed modifications to the SRS. Thus after several iterations, a complete and precise SRS is generated.

The spiral model focuses on eliminating errors and unattractive alternatives early in the development process through the use of risk analysis. Defining the system through an iterative approach allows the stakeholders to remain focused on the smaller manageable issues within each cycle. Thus, this model is better suited for large projects as compared to the other models. A strong focus of the spiral model is on communications with the customer. The software engineer works together with the customer to (1) decide on solution alternatives, (2) determine the risk of the system/solution, and (3) evaluate the delivered system.

The disadvantage of the spiral model is that the process is not well documented and fails to provide clear guidance to the requirements engineer. The model needs further elaboration of each spiral, and lacks detailed checklists and guidelines necessary for consistent interpretation and use of the spiral model. Furthermore, risk analysis is a vital part of the process; therefore, the success of the project is largely dependent on the risk-assessment expertise.

2.5.5 RUP Process

The Rational Unified Process (RUP) is a further refinement of the Spiral Model. It is a general framework that can be used to describe specific development processes. According to The Unified Software Development Process by Jacobson et al., every 'software life cycle' is 'a cycle over four
phases in the following order: inception, elaboration, construction, and transition’. A 'phase' is 'the span of time between two major milestones in the software development process'. And 'major milestones' can be thought of '... as synchronization points where a well-defined set of objectives is met, artifacts are completed, decisions are met to move or not to into the next phase, and where the managerial and the technical realm conjuncts' And the core of the [phases] is state-based, and the state is determined by what fundamental questions you are trying to answer:

- Inception - do you and the Customer have a shared understanding of the system?
- Elaboration - do you have architecture to be able to build the system?
- Construction - are you developing product?
- Transition - are you trying to get the Customer to take ownership of the system?

The inception phase contains whatever workflows are necessary to get the stakeholders to agree (the keyword there is agree, and not "have sure knowledge") on the objectives, rough architecture, and rough schedule of the project. If the stakeholders are extremely knowledgeable, then little analysis will be required. If they need to be severely educated, then lots of analysis will be required.

The essence of RUP is iteration. And the essence of iteration is that each iteration ends in a deliverable -- preferably one that executes. Even in inception, you are going to want a few iterations that show growing functionality. During inception you will gather a significant (a half? a third? it
depends...) fraction of the use cases. You will focus on those that seem to be central or key.

During elaboration you will tighten up your architecture and your plan. The nature of the iterations won't necessarily change much; but the longevity of the software produced will certainly increase. Early iterations (usually in the inception phase) have a tendency to get thrown out. During elaboration you will discover the rest of the use cases (or at least their first approximations) and will implement the minimal set.

During construction you will drive towards giving the customer the minimum system that they need. The nature of the iterations won't change much, but your focus will be on identifying the smallest possible deliverable that will still meet at least some of the customers needs. During construction, the use cases will change a bit as the customer sees the growing system and feeds changes back to you.

During transition, you will drive towards fleshing out the functionality of the system, and incorporating the mounds of customer feedback that you are surely to get. The nature of your iterations won't change much. During transition the use cases are likely to undergo drastic changes as the customers actually use the system and realize that it is not exactly what they needed.

Again, the essence of RUP is iteration, and the essence of iteration is the production of executable deliverables. You may also be producing UML diagrams, or some other form of model too. Such models take two forms. One is a model of the architecture, which is seeded during inception and established during elaboration. This model is likely to be a permanent document. The other kind of model is created at the beginning of each
iteration, as a way to plan what the structure of the iteration will look like. These models are most likely temporary documents. You might find a few that are essential and should be retained; but many will be discard able.

The transition from phase to phase is gradual. Management is not done by placing dates upon the phase boundaries. Nor are there phase gate events that mark the transition from one to another. Rather, management is done based upon iterations of the development. The first iterations start in the inception phase. The project plan is pretty straightforward. It contains a list of proposed iterations (which all parties agree is likely to change). Each proposed iteration has an estimate. The proposed iterations are not assigned dates. Rather decision points are identified for example:

1. By week 6, if iteration 1 and 2 are not complete, then hire one additional person.
2. By week 10, if the customer has not accepted iterations 1-4, then remove iteration 8 from the release.
3. By week 15, if the customer has not accepted iterations 1-4, then cancel the project.

A project plan is not a statement of what will be. Rather it is a statement of how risks will be managed. It is a plan of contingencies, as opposed to purely a plan of action. As the project proceeds, we will learn more about how quickly the iterations are developed. We'll learn about the iterations that we forgot when we made the project plan. The customer will see the developing project and will change the requirements. Thus, the plan must be malleable, flexible, and responsive.
2.5.6 XP Model

This is a minimal implementation of RUP, which was called dX. The Principles and practices of dX were identified by Ward Cunningham, Kent Beck, Ron Jeffries, and a host of other developers and methodologists. They have used this process on several projects with significant success. Because of that success, they have gathered quite a following. They call the process Extreme Programming; or for short: XP.

The main objective of XP is to shorten the development time of a project while increasing the interaction with and feedback from the customer (Beck 1999). XP proposes a core set of principles – small releases, pair programming, continuous integration, and on-site customer participation.

Communication and customer involvement is a major component of XP. In fact, XP takes customer interaction to the extreme by including customer participation in all phases of the development life cycle. XP also stresses on product evaluation, which is accomplished by writing and performing test cases (Figure 2.11). Another important principle of XP is pair programming. In pair programming, two developers always work together, one developer on the keyboard and the second giving instructions or simply reading over the shoulder of the typist to ensure coding accuracy.

The requirements engineering phase in XP occurs as the requirements are elicited from the on-site customer, who takes part in the complete development life cycle (Beck 1999). Once the requirements are elicited, the developers then ask the customer to prioritize the requirements, which are implemented according to their priority. The test cases for the prioritized requirements are then prepared by the customer with help from the developer.
Finally, the product is coded and validated against the test cases by the customer. The feedback obtained from the customer is used in the planning of the next increment. No formal requirements document such as SRS is produced in XP since documentation is considered to cause delays to the project schedule.

XP is guideline centric and hence lacks detailed descriptions for producing stories and tasks. The process follows the evolutionary approach of building the product in increments, and thus improving the confidence of the customer in the project. Since the customer is involved at all times in the XP development cycle, the finished project has a better chance of meeting the customer expectations. However, assuming that all business domain concerns can be represented by only one on-site customer is impractical, especially for complex systems (Leite 1998). XP offers faster completion times for the project by avoiding documentation overhead. Even though the development time is shortened by less documentation, problems are faced in maintenance due to lack of reference documents.
2.6 OTHER TECHNIQUES

2.6.1 Modeling Requirements

Modeling is a vital part of Requirements Engineering in order to provide abstract representation of real world activity. Pressman describes the Software Engineering process as a series of modeling tasks that conclude in development of the system and the analysis modeling is the ‘first technical representation’ of a system. Pressman goes on to specify minimum requirements for modeling technique such as ease of analysis, process for tracking interfaces. He also stresses the need for new tools to improve on the narrative text that is currently, an essential component of Requirements documentation.

Modeling is an important objective of this phase, because it involves the basic issues of representing and reasoning about the knowledge and information captured during the elicitation phase. They were among the first who recognized the importance of modeling the requirements and argued that analysis is needed to understand and represent the requirements in a clear and comprehensible manner. Yeh et al (1984) state that the complexity of large computer-based systems demands an additional layer of understanding between the real world and the requirements specification. This is accomplished through modeling, which is generally performed by starting with the representation of the proposed system’s environment, followed by working towards the system and its software component. The models are also used for evaluating the system requirements on the basis of various requirement attributes.
2.6.2 Types of Modeling

Easterbrook describes five types of modeling to describe the system comprehensively. These are:

- Enterprise modeling to capture the organizational conditions in which the system should operate. Issues like business rules, goals and tasks and responsibilities of individual members fall within the purview of enterprise modeling.

- Data modeling attempts to describe a system from the object perspective. Objects are described in terms of their attributes and their relationship with other objects and the processes that transform them. Data Modeling provides answers to questions such as what are the primary data objects to be processed, their composition and attributes, their location, relationship between objects, the processes that transform them, etc. The ERD or Entity-Relationship Diagram illustrates this data modeling aspect of the system.

- In Behaviour modeling the dynamic behaviour of stakeholders and systems (both current and future) are described. A rigorous (though effort-intensive) approach may sometimes call for construction of 3 models: the current physical system, the current logical model to determine the essential functionality and lastly, the future or proposed system. A choice of Tools for behaviour modeling is available ranging from structured to object-oriented methods and from soft to formal methods. Formal methods have many advantages starting with ease of automated reasoning and analysis. However, use of Formal methods requires specialized training,
which is usually outside the scope for the typical software development training and hence difficult to implement. Soft methods on the other hand provide rich representation but are not amenable to automated processes.

- Domain modeling enables understanding of the world in which the system will operate.

- Modeling Non-Functional Requirements (NFRs). Many of these NFRs are in the quality domain and difficult to analyze and measure. A feature of NFRs (Safety, Security, Reliability and Usability) is they apply to the system as a whole and cannot be broken down into corresponding NFRs for the individual components.

Analysis also involves interacting with the users to clarify misunderstandings and determine the priorities and value of requirements. In practice, requirements are negotiated, not just elicited; it is incorrect to assume that complete and well-defined requirements for a system are waiting to be discovered. The analysis phase facilitates a common understanding among all stakeholders about the proposed system.

2.6.3 Formal Methods

Formal Methods involve use of techniques employing formal logic and discrete mathematics in the specification, design, and implementation of software and hardware systems. Lamsweerde defines Formal Specifications as “expressions, in some form and at some level of abstraction, of a collection of properties some system should satisfy. He goes on to define the essential content of Formal Specification, namely it must have syntax, semantics and rules for inferring useful information from the specification.
Formal Methods enable precision in capturing requirements and remove ambiguity. It is also possible to use different logics (temporal, deontic, linear) for different aspects of the system. They also make the specifications amenable to automation with the resulting benefits of automated analysis and reasoning leading to early detection and elimination of errors in Requirements and ultimately, more quality in the end software product.

However, Formal Methods are difficult to understand for the untrained software developer besides having other disadvantages. Widmaier reports an interesting study in which a project was completed in two ways: one using Formal Methods and in the other way, traditional development techniques were used. The formal approach produced better results; however, customer’s reliability criteria were not met in either method. Formal Methods were good for the requirements phase but were not effective in project management, and surprisingly, documentation for re-engineering and maintenance. Code optimization was also sub-optimal. Widmaier concludes by recommending a hybrid approach.

2.7 SUMMARY

Requirements Engineering comprises a series of engineering decisions that lead from recognition of a problem to be solved (commencement) through a detailed specification of that problem and beyond into the region where the usefulness of the software is prolonged by injecting more requirements. A Software Product’s Requirements are never ‘final’ but are prone to change well into deployment and operations.

Requirements Engineering is the most important part of Software Development because: It is the first as well as the last activity in Software
Development; its contribution does not end with commencement of development but goes on after deployment in its role of keeping the software current and useful.

- Thorough Requirements Engineering is vital to the economical development of quality software. A software product can only be as good as the Requirements Engineering work done on it as has been proved through many studies.

- Requirements Engineering requires the deployment of skills from many fields, such as:

  - Computer Science to assess the feasibility and architect the solution from the problem under solution,

  - A variety of Logics are used in Formal Methods that enable precision in capture of requirements,

  - Traditional Systems Analysis,

  - Requirements Engineering is a human-centered process. To understand the impact of the system on humans and organization, it calls for the Requirements Engineer to deploy skills in Sociology, Cognitive psychology, Linguistics, Epistemology, Ontology, etc.

The Importance of Requirements Engineering: A number of studies have been made to determine the impact of Requirements Engineering. They clearly establish the importance or Requirements Engineering for quality software development. These effects have been known and appreciated since quite some time. Boehm as early as 1981, wrote about the amplification effect of errors in Requirements Engineering: late correction of requirement errors
could lead to escalation of 200 times in terms of cost, if left uncorrected at the Requirements stage itself. The primary cause in two major failures in NASA’s programs was identified as errors in functional and interface requirements- Lutz. The Standish Group made a comprehensive study of the progress and satisfaction of nearly 8000 projects in 350 companies. The results were startling: one-third of the projects were never completed, half of the projects were deemed to be partially successful but with major cost over runs and significant delays. Managers in these projects identified the major cause (45%) as poor requirements. Brooks summed it up well when he identified Requirements Engineering as “the single most important task” that a software developer does for the end user of the software product.

Each of the Requirements Processes gave their own particular uses. Even the earliest waterfall process is used for short simple developments. The later processes RUP and XP have spawned many variations and are the largest used in the industry now. New processes are being experimented with in companied for their own situation and in Universities, but no radically different ones have been accepted industry wide yet.