Chapter 1

Software Requirement

Chapter 1: Software Requirement

Introduction

This chapter introduces the notion of software requirements and requirements engineering. In this introductory chapter, we discuss software requirements and the requirements engineering process. The development of computer-based systems has been overwhelmed with problems since the 1960s. Systems may be delivered late, over budget, they do not do what users really want and they are often never used to their full effectiveness by the people who have paid for them. There is rarely a single reason (or a single solution) for these problems but we know that a major factor is problems with system and software requirements. System requirements define what services the system should provide and set out constraints on the system's operation. Common problems which arise with system requirements are that:

1 The requirements do not reflect the real needs of the customer for the system.

2 Requirements are inconsistent and/or incomplete.

3 It is expensive to make changes to requirements after they have been agreed.

4 There are misunderstandings between customers, those developing the system requirements and software engineers developing or maintaining the system.

We are convinced that the best way to reduce these problems is to improve the processes of discovering, understanding, negotiating, describing, validating and managing system requirements. We believe

that the best way to do this is in a gradual way where you introduce new or improved procedures over a period of time.

1.1 The Goal of Software Development

Thousands of software development teams worldwide are engaged right now in developing widely different software applications in widely different industries. But although we work in different industries and speak and write in different languages, we all work with the same technologies, and fortunately, we have the same clear goal: *to develop quality software—on time and on budget—that meets customers' real needs.* However, our customers are quite different.

For some of us, the customer is an external entity, purchase order in hand, whom we must convince to disregard our competitor's claims and to buy *our* wrapped software product because it's easier to use, has more functionality, and, in the final analysis, is just better.

For others of us, the customer is a company that has hired us to develop its software, based on expectations that the software developed will be of the highest quality achievable given today's state of the art and will transform the company into a more competitive, more profitable organization in the marketplace.

For others of us, the customer is sitting down the hall or downstairs or across the country, waiting anxiously for that new application to enter sales orders more efficiently or to use e-commerce for selling the company's goods and services so that the company we *both* work for will ultimately be more profitable and our jobs more rewarding.

1.2 The Root Causes of Project Success and Failure

The first step in resolving any problem is to *understand the root causes*. Fortunately, the Standish Group survey went beyond the assessment phase and asked survey respondents to identify the most significant

factors that contributed to projects that were rated “success,” “challenged” (late and did not meet expectations), and “impaired” (canceled), respectively. The 1994 Standish Group study noted the three most commonly cited factors that caused projects to be “challenged”:

1. Lack of user input: 13 percent of all projects

2. Incomplete requirements and specifications: 12 percent of all projects

3. Changing requirements and specifications: 12 percent of all projects

Thereafter, the data diverges rapidly. Of course, your project could fail because of an unrealistic schedule or time frame (4 percent of the projects cited this), inadequate staffing and resources (6 percent), inadequate technology skills (7 percent), or various other reasons. Nevertheless, to the extent that the Standish figures are representative of the overall industry, it appears that at least *a third of development projects run into trouble for reasons that are directly related to requirements gathering, requirements documentation, and requirements management*.

Although the majority of projects do seem to experience schedule/budget overruns, if not outright cancellation, the Standish Group found that 9 percent of the projects in large companies were delivered on time and on budget; 16 percent of the projects in small companies enjoyed a similar success. That leads to an obvious question: What were the primary “success factors” for those projects? According to the Standish study, the three most important factors were:

1. User involvement: 16 percent of all successful projects

2. Executive management support: 14 percent of all successful projects

3. Clear statement of requirements: 12 percent of all successful projects

1.3 Software Process

A software process is the blueprint that guides every phase of software development — from idea to deployment. You follow each step to ensure the final product turns out just right. A software process is a set of related activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. However, business applications are not necessarily developed in this way. New business software is now often developed by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components. There are many different software processes but all must include four activities that are fundamental to software engineering:

1 Software specification. The functionality of the software and constraints on its operation must be defined.

2 Software design and implementation. The software to meet the specification must be produced.

3 Software validation. The software must be validated to ensure that it does what the customer wants.

4 Software evolution. The software must evolve to meet changing customer needs.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. There is no ideal process and most organizations have developed their own software development processes. Processes have evolved to take advantage of the capabilities of the people in an organization and the specific characteristics of the systems that are being developed.
Sometimes, software processes are categorized as either plan-driven or agile processes. Plan-driven processes are processes where all of the

process activities are planned in advance and progress is measured against this plan. In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.

1.4 Types of Models in Software Engineering

When it comes to building software, there isn't a one-size-fits-all solution. Just like you wouldn't use the same tool for every task, you don't use the same process for every project. A suitable process model can distinguish between a smooth journey and a rough road ahead.

Waterfall Model

The Waterfall Model is one of the oldest and most straightforward process models in software engineering. In this, you follow a sequential, step-by-step process. Once you complete one phase, you move to the next without turning back . Depending on the project, this rigid structure can be both a strength and a limitation. Let's understand the structural process of it in detail:

Requirement Gathering: Gather all the requirements upfront. The goal is to understand the client's needs before any work begins.

Design: Once requirements are precise, design the system architecture and components.

Implementation: This is where the actual coding takes place. Developers build the software based on the design.

Testing: Once the code is complete, it undergoes rigorous testing to ensure it meets the requirements.

Deployment: After testing, the product is deployed.

Maintenance: Post-deployment maintenance ensures the software continues functioning and is updated as needed.

A classic example would be building a simple application or a regulatory-compliant system where the requirements are fixed, and strict design adherence is required. However, if you expect requirements to shift or evolve, you should explore other types of models in software engineering. So, let's explore them.

Agile Methodology

Agile embraces change and adaptability instead of following a rigid, linear path. It's all about breaking down the project into smaller chunks or sprints and delivering incremental updates. This model thrives in environments where requirements evolve, and quick delivery is key.

Agile is based on 12 core principles, but here are the key ones you need to know:

- **Customer Collaboration Over Contract Negotiation**: Agile puts customer feedback at the center of development.
- **Responding to Change Over Following a Plan**: The ability to adjust is prioritized over sticking to a predefined path.
- **Deliver Working Software Frequently**: Regular deliveries of working software keep everyone aligned and provide real-time feedback.
- **Simplicity is Key**: Focus on the essentials and avoid unnecessary features that don't add value.

In an Agile environment, work is done in sprints — short, time-boxed iterations that typically last 1 to 4 weeks. At the end of each sprint, you deliver a working piece of software and gather feedback from stakeholders to refine the next iteration.

Incremental Model

The Incremental Model breaks the project into smaller, more manageable chunks. Instead of delivering the entire system at once, you develop and

deliver it in increments — pieces of functionality built on top of one another.

Each increment is a working version of the software, and with every new release, you gradually add more features. This approach makes it easier to handle changes, and you get something functional earlier than you would with traditional models like Waterfall.

Rapid Application Development (RAD)

In software development, sometimes you need to build something fast, and RAD gives you that power. You create an initial working version of the product (a prototype) early on. You gather user feedback, refine the product, and repeat the process.

Here's how the RAD model works:

- **Prototyping**: You start with a rough product version and enhance it based on honest user feedback. This iterative approach helps refine the system before it's fully developed.
- **Reusable Components**: RAD emphasizes using pre-built, reusable components — saving time and effort while ensuring consistency across the product.
- **Fast Feedback Loops**: User involvement throughout the process allows for quick adjustments and ensures that the final product is aligned with what the users need.

1.5 Software Requirements

The requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and

checking these services and constraints is called requirements engineering (RE).

The term *requirement* is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function.

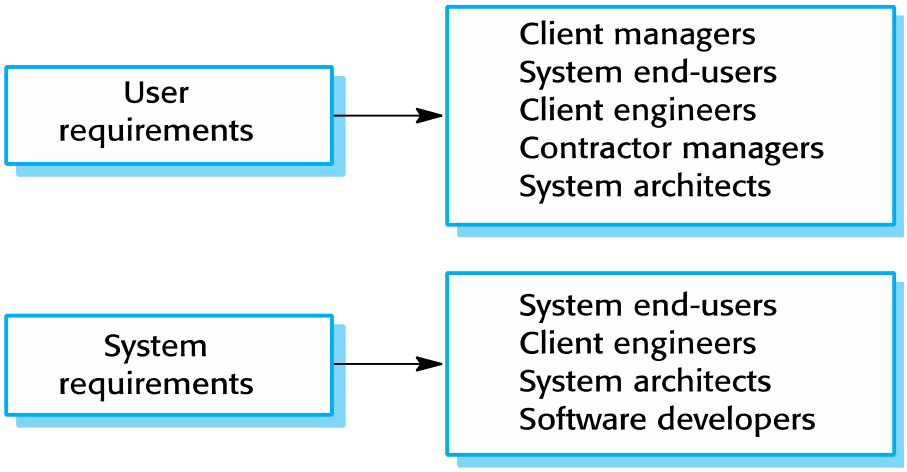
In general, requirements fall into two broad categories: *market-driven* and *customer-specific*. These two broad categories of requirements have different characteristics and are often treated differently within a development process. Some of the key differences reported in an extensive study of requirements engineering projects

Market-Driven Projects	**Customer-Specific Projects**
Requirements are sketchy and informal	Requirements are voluminous and more 'formal'.
Use of techniques from manufacturers rather than Software Engineering, e.g. QFD	Use techniques from Software Engineering
Specification is in the form of a marketing presentation	Specification may be in hundreds of pages of documentation
Not readily identifiable 'customer'. Developers tend to have less experience.	Make use of domain expertise. Developers have in-depth knowledge of domain.
Projects rely on consultants for advice on desirable features	Projects rely on in-house personnel
Less structured approaches adopted.	Structured approach following a particular approach

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. We distinguish between them by using the term *user requirements* to mean the high-level abstract requirements and

system requirements to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services
the system is expected to provide to system users and the constraints under which it must operate. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality.



2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Different kinds of requirement are needed to communicate information about a system to different types of reader. You need to write requirements at different levels of detail because different types of readers use them in different ways. The readers of the user requirements are not usually concerned with how the system will be implemented and

may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

1.6 Software Requirements Types

Software system requirements are often classified as functional or non-functional requirements:

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.
2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole rather than individual system features or services.

Functional Requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.

When expressed as user requirements, functional requirements should be written in natural language so that system users and managers can understand them. Functional system requirements expand the user requirements and are written for system developers. They should describe the system functions, their inputs and outputs, and exceptions in detail.

Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.

Functional requirements, as the name suggests, have traditionally focused on what the system should do. However, if an organization decides that an existing off the-shelf system software product can meet its needs, then there is very little point in developing a detailed functional specification. In such cases, the focus should be on the development of information requirements that specify the information needed for people to do their work. Information requirements specify the information needed and how it is to be delivered and organized. Imprecision in the requirements specification can lead to disputes between customers and software developers. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

Examples:

- **User Authentication**: The system must allow users to log in using a username and password.
- **Search Functionality**: The software should enable users to search for products by name or category.
- **Report Generation**: The system should be able to generate sales reports for a specified date range.

Non-functional requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. These non-functional requirements usually specify or

constrain characteristics of the system as a whole. They may relate to emergent system properties such as reliability, response time, and memory use. Alternatively, they may define constraints on the system implementation, such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

It is often difficult to relate SW components to non-functional requirements. The implementation of these requirements may be diffused throughout the system.

Non-functional requirements may affect the overall architecture of a system rather than the one component . For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.

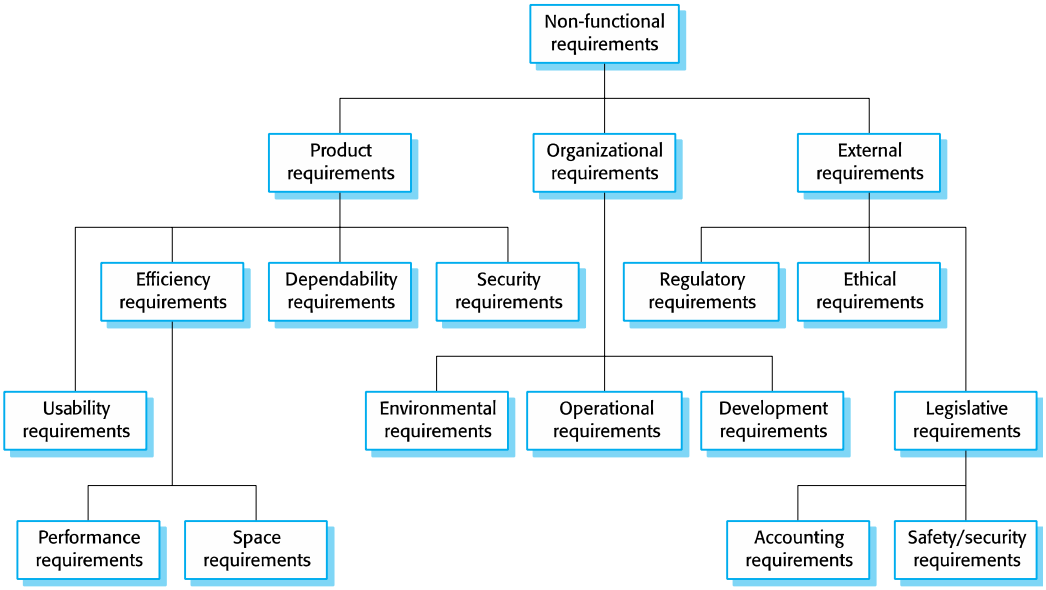
A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.

The non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or external sources:

Product requirements

These requirements specify or constrain the runtime behavior of the software. Examples include performance requirements for how fast the

system must execute and how much memory it requires; reliability requirements that set out the acceptable failure rate; security requirements; and usability requirements.



Organizational requirements

These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organizations. Examples include operational process requirements that define how the system will be used; development process requirements that specify the programming language; the development environment or process standards to be used; and environmental requirements that specify the operating environment of the system.

External requirements

This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include

regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a nuclear safety authority; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

A common problem with non-functional requirements is that users or customers often propose these requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Non-functional requirements are considered more critical than functional requirements for the following reasons:

1- If the non-functional requirements of the system are not met, the whole system may be useless.
2- Non-functional requirements may affect the overall architecture of a system rather than one component.
3- A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.
4- Customers often propose these requirements as general goals which leave scope for interpretation and subsequent dispute

Domain requirements

Constraints on the system from the domain of operation. They may be new functional requirements in their own right, constrain existing functional requirements, or set out how particular computations must be

carried out. Domain requirements are the requirements that are characteristic of a particular category or domain of projects. Domain requirements can be functional or nonfunctional.

Examples:

- **Healthcare**: The software must comply with HIPAA regulations for handling patient data.
- **Finance**: The system should adhere to GAAP standards for financial reporting.
- **E-commerce**: The software should support various payment gateways like PayPal, Stripe, and credit cards.

Domain requirements reflect the unique needs and constraints of a particular industry. They ensure that the software is relevant and compliant with industry-specific regulations and standards.

1.7 Advantages of classifying software requirements include:

1. **Better organization:** Classifying software requirements helps organize them into groups that are easier to manage, prioritize, and track throughout the development process.
2. **Improved communication:** Clear classification of requirements makes it easier to communicate them to stakeholders, developers, and other team members. It also ensures that everyone is on the same page about what is required.
3. **Increased quality:** By classifying requirements, potential conflicts or gaps can be identified early in the development process. This reduces the risk of errors, omissions, or misunderstandings, leading to higher-quality software.
4. **Improved traceability:** Classifying requirements helps establish traceability, which is essential for demonstrating compliance with regulatory or quality standards.

1.8 Requirements Management

Requirements define capabilities that the systems must deliver to a set of requirements often determines the success (or failure) of projects. It makes sense, therefore, to find out what the requirements are, write them down, organize them, and track them in the event that they change. Let's take a closer look at some key concepts.

Anyone who has ever been involved with complex software systems—whether from the perspective of a customer or a developer—knows that a crucial skill is the ability to *elicit* the requirements from users and stakeholders.

Since hundreds, if not thousands, of requirements are likely to be associated with a system, it's important to *organize* them.

Since most of us can't keep more than a few dozen pieces of information in our heads, *documenting* the requirements is necessary to support effective communication among the various stakeholders. The requirements have to be recorded in an accessible medium: a document, a model, a database, or a list on the whiteboard.

What do these elements have to do with managing requirements? Project size and complexity are major factors. Nobody would bother talking about "managing" requirements in a two-person project that had only 10 requirements to fulfill. But to verify 1,000 requirements—a small purchased software product—or 300,000 requirements. It's obvious that we will face problems of organizing, prioritizing, controlling access to, and providing resources for the various requirements.

Homework

Q1) Suggest three possible response for software projects success and failure?

Q2) What are the main activities of software process?

Q3) Compare between user and system requirements?

Q4) What are the advantages of classifying software requirements?

Q5) What are the main types of non-function requirements? Give an example for each type?

Q6) Suggest five functional requirements for each of the following software systems, Giving reasons for your answer.

*An interactive web-based game in which characters move around, across and collect items.

* A student affairs system for the faculty of Computers and Information Sciences.

* An operating system for mobile devices.

* A system that sends out reminders about magazine subscriptions.

Q7) Suggest three non-functional requirements for each of the following software systems, Giving reasons for your answer.

- A software system for online banking services.
- A web-based ticket issuing system for trains.
- A software system to issue electrical bills monthly.
- A system that is used for image editing.
- A software system to control microwave oven.

References

1- Ian Sommerville, " Software Engineering ", Tenth Edition, Pearson Education Limited 2016.
2- Dean Leffingwell, Don Widrig "Managing Software Requirements: A USE CASE APPROACH " Second Edition, Addison-Wesley, 2003
3- P. Loucopoulos & V. Karakostas, "System requirements engineering", McGraw-Hill, 1995.