Requirements Engineering

Is the process of establishing the services that a customer requires from a system, as well as the constraints under which it will operate and be developed. The system requirements are the descriptions of the system services and constraints.

A requirement may range from a high-level abstract statement about a service or system to a detailed technical specification. Requirements also serve a dual function:

- Requirements may form the basis of a bid for a contract
- Requirements may form the basis of a contract

User Requirements: Statements put in natural language that lay out the services that the system provides and how it will operate. Oriented for customers.

System Requirements: A structured and technical document that provides detailed descriptions of the components and functionality of the system and services. Defines what is to be implemented.

User requirements are often meant for client managers, system end-users, client engineers, contractor managers, and system architects. System requirements are often meant for System end-users, client engineers, system architects, and software developers.

System Stakeholder: Any person or group of people who is affected by the system and who has a legitimate interest in the system. These include end-users, system managers, system owners, and external stakeholders.

Agile Requirements: Agile methods argue that developing detailed requirements is a waste of time since they change so quickly. Usually use incremental requirements engineering methods. This is practical for business systems but not suitable for systems that require pre-delivery analysis.

Functional and non-functional requirements

Functional Requirements: Statements about the services that a system will provide, how the system will react to inputs, and how the system will behave in certain situations. They may also state what the system is not allowed to do.

Non-function Requirements: Constraints on how the system itself is to be developed, e.g. development restrictions or standards. These often apply to the system as a whole rather than individual components.

Domain Requirements: Constraints on the system from the domain in which it will operate.

Functional Requirements

Function requirements describe functionality and system services. These depend on the type of software being made, the expected users, and the environment in which the system is being used. Function user requirements are generally high-level statements about what the system will do, while function system requirements are detailed statements about the system services.

Problems can rise when function requirements are not properly stated. For example, ambiguous requirements may be interpreted differently by different parties, causing development conflicts.

Requirements, ideally, should be both consistent and complete. In practice however, this is rarely possible as it is extremely difficult to consider every possibility of a given system.

Non-functional Requirements

Non-functional requirements define system properties and constraints. These may include requirements about how the system may run independent of its implementation, as well as restrictions on the development process itself. Often, non-functional requirements are far more critical than functional requirements.

Non-functional requirements tend to affect the overall architecture of a system rather than individual components of the system. Individual non-functional requirements may generate many different functional requirements. For example, security requirements or communication limits may drastically change how components are organized.

Product Requirements: Requirements that specify how the product will behave.

Organizational Requirements: Requirements that arise from organizational policies, process standards, implementation requirements, etc.

External Requirements: Requirements that arise from factors that are external to the system, e.g. regulations, laws, etc.

It is often very difficult to state non-functional requirements precisely. We often try to use verifiable statements when defining non-functional requirements, but this is not always possible. Goals are helpful in the defining of non-functional requirements.

There are several metrics for specifying non-functional requirements:

- **Speed:** Measures number of occurrences of a certain event in a given time
- **Size:** Digital storage space or physical space
- **Ease Of Use:** How long do users need to take to learn the system?
- **Reliability:** How long between failures?
- **Robustness:** How fast can the system recover from failures?
- **Portability:** How many systems does our software need to support?

Requirements Engineering Process

The processes which are used in requirements engineering vary widely depending on what is being done. However, there are a number of activities common to processes, namely requirements elicitation, analysis, validation, and management. Requirements engineering is often iterative.

Requirements Elicitation

Software engineers work with a range of stakeholders in order to find out more about the application domain, and thus make more effective requirements. These stages include requirements discovery, classification and organization, prioritization and negotiation, and specification.

Stakeholders tend not to know exactly what they want, or tend to express their desires in their own terms rather than in software engineering terms. In addition, different stakeholders may have conflicting desires. Organizational and political factors also influence requirements. Requirements are also continuously changing.

Requirements Discovery: Involves interacting with stakeholders to establish requirements.

Requirements Classification and Organization: The process of organizing requirements into coherent groups.

Requirements Prioritization and Negotiation: Establishes requirement priorities and resolves requirement conflicts.

Requirements Specification: Documents requirements and prepares for next iteration of requirements engineering.

Requirements Discovery

Requirements discovery is the process of gathering information about the requirements of the system. Involves interaction with a range of stakeholders.

Formal and informal interviews with stakeholders are common. Can either be closed interview based on a pre-determined list of questions, or a more open ended one. To be an effective interviewer, it is important to remain open-minded, and to avoid pre-conceived notions about the requirements and stakeholders.

In practice, interviews consist of a mix of closed and open-ended questions. They are useful for getting an overall understanding of what stakeholders want. They also provide opportunities to probe the stakeholders about what they want.

One problem that exists with interviews is that application specialists may not be skilled at communication highly technical domain requirements.

Ethnography: The process of studying people and how they work. This involves both social and organizational factors of groups of people. It has been shown that people are far more complex than suggested by most system models.

Ethnography can be used to generate requirements that more closely align with what people actually want. These requirements tend to be derived from cooperation and awareness of what others are doing. It is also useful for establishing an understanding of a pre-existing system.

Focused ethnography combines general ethnography with technical prototyping. This combination helps fix the flaws of each system in isolation.

Stories and Scenarios: Real life examples of how a system will be used. These describe how a system will be used for particular tasks. These are easier for stakeholders to relate to than technical requirements. Scenarios should include a description of the beginning-state, the flow of events, possible error states, information about concurrent events, and a description of and end-state.

Requirements Specification

Requirements specification is the process of gathering system and user requirements in a requirements document. User requirements should be understandable by stakeholders who don't possess technical knowledge, while system requirements can provide more technical details.

There are several ways of specifying requirements:

- **Natural Language** Using plain language sentences to describe requirements.
- **Structured Natural Language** A more templated form of natural language.
- **Design Description Languages** Like programming languages but more abstract.
- **Graphical Notation** e.g. UML
- **Mathematical Specification** Rigorous mathematical definitions.

In principle, requirements should state what the system does, while the design describes how it does it. In practice however, these two things are inseparable. Requirements may depend on the design or vice versa. The system may also interface with other structures that generate their own requirements.

Natural Language

Requirements written in natural language use plain sentences supplement with graphical illustrations. They tend to be easier to understand for most people. To effectively use natural language, the language used should be consistent, and jargon should be avoided.

Problems with using natural language include lack of clarity, confusion of requirements, and often the amalgamation of different requirements into one.

Structured Language

This approach restricts the writer to using a standard way of writing requirements. This works well for more technical applications, but is too rigid for most business applications.

These requirements define the functionality of the entity. They also describe the inputs and outputs of the system, as well as the information needed for computation and for other entities. They describe the actions the system needs to perform, as well as the pre/post conditions and side effects of said actions.

Tabular Specification: Used to supplement natural language. Often used when multiple possible courses of action need to be described.

Use cases: Identifies the actors that interact with the system in a certain scenario. These should describe all possible interactions with the system. Often tabular specification, alongside UML are used to create these use cases.

Software Requirements Document

This document officially states what is required of the developers. Includes both user and system requirements. Should not include design properties. System customers, managers, system engineers, test engineers, and maintenance engineers are the most common users of this document Documents developed will typically have less detail.

Requirements documents typically have the following structure:

- **Preface**
- **Introduction**
- **Glossary**
- **User Requirements**
- **System Architecture**
- **System Requirements**
- **System Models**
- **System Evolution**
- **Appendix**
- **Index**

Requirements Validation

Requirements validation is concerned with verifying that the requirements align with what the customer wants. Error costs are high so this stage is vital.

Checking the requirements involves considering the validity, consistency, completeness, realism, and verifiability of the requirements. This is often achieved through requirements reviews, prototyping, and generating test cases.

Requirements Reviews: Should be held often and should involve both client and contractors. May be formal or informal, but good communication is important for both. Reviews should check verifiability, comprehensibility, traceability, and adaptability.

Requirements Change

Requirements always change after development begins. For example, stakeholders may be different from the actual users of the system. There may also be many different communities of users who each have different requirements.

Requirements Management

This is the process of managing changing requirements during system development. New requirements emerge all the time, so it is important to keep track of and maintain links between everything.

It is important to establish the level of management detail that is required for the requirements. Decisions have to be made about how to identify requirements, change management processes, traceability policies, and tooling support.

To decide if a change should be accepted, you need to analyze the problem and the new specification, analyze the change that the new requirement will bring, and finally analyze the implementation of the new requirement.

System modeling

System modeling is the process of developing models for a system that describe different views or perspectives of the system. Often uses a graphical notation like UML. System modeling helps analysts and can be used to communicate with clients.

Models of an existing system help clarify what the existing system is supposed to do, while models of a new system are mostly suited towards conveying ideas to stakeholders. You should be able to generate a partial or complete implementation of the system from the system model.

There are different kinds of perspectives you can take while modeling a system:

- **External Perspective:** you model the context in which the system will run
- **Interaction Perspective:** you model the actors that interact with the system
- **Structural Perspective:** you model the overall system organization
- **Behavioral Perspective:** you model how that system reacts to changes or outside events

There are also different types of UML diagrams. These include activity, use case, sequence, class, and state diagrams. Graphical models like these are often used to facilitate discussion of an existing system, or to document new systems. They can also serve as detailed descriptions for parts or all of a system.

Context Models

Context models are used to illustrate the operational environment of a system. They model things that lie outside of the system itself. Social or organizational factors may affect this type of modeling.

System Boundaries: Distinguish the inside from the outside of the system. They describe what other systems depend on this system, and what other systems this system depends on. It is often a political judgement.

Process models expand upon this by showing how the system is used in relation to other systems rather than simply showing the organization. UML is often used in this model.

Interaction Models

These model user interactions with the system. They are important in defining user requirements. They also help in communicating problems that may arise, as well as whether a proposed system is likely to deliver the performance that is required of it.

Use Case Modeling

Use cases are modeled with the actors and with the actions that they perform.

Sequence Diagrams: a big part of UML, used to model the interactions between actors and the objects in a system. They show the sequence of interactions that take place in a use case.

Structural Models

Structural models describe the organization of the system in terms of the components that the system comprises of. They may be static, or may dynamically show how the system acts when it executes.

Class Diagrams: Used in OOP development to show the different classes in a system. It models objects (classes), and associations between objects. Classes often represent things in the real world.

Generalization

We use generalization to manage complexity. We can skip learning the details of any given component of the system in lieu of learning the details of broader classes. This allows us to infer commonalities between different objects more easily. It is often useful to look for ways to generalize classes.

Object Class Aggregation Models

Aggregation models show when classes are composed of other classes, such as through inheritance.

Behavioral Models

These models describe the dynamic behavior of a system while it is executing. They detail how the system will react to different stimuli from its environment. These stimuli are typically either data or events.

Data-driven Modeling

Data-driven models show the sequence of actions necessary for the system to process incoming and outgoing data. Useful for showing end-to-end processing in a system.

Event-driven Modeling

Event-driven models show how a system reacts over time to both internal and external events. Usually based off finite state machines.

State machine models show system behavior as a collection of nodes with lines connecting them that model how the system changes state as certain events occur.

Model-driven Engineering

Model-driven engineering is when models are the principal outputs, with executable systems being generated automatically from the model. This raises the level of abstraction so that engineers need not concern themselves with language details or platform specifics.

Still in its infancy, not as mature as other engineering methods. It might not be good for generating actual implementations, and may cost more.

Model-driven architecture as its precursor. Focuses on models in software design, uses UML to model systems at different levels of abstraction. Types of MDAs include computation independent models, platform independent models, and platform specific models.

This type of engineering claims supports agile methods, as it is an iterative approach. Up-front modeling contradicts the agile method however, so it may not be totally suitable.

MDA's have seen limited adoption, as it is quite specialized and provides limited tool availability. In addition, implementation is rarely the biggest issue that software engineers face. Spotlight has been stolen partially by agile methods as well.

Architectural Design

This type of design is concerned with understanding how a system should be organized. It is the critical link between design a requirements, as it identities main structural components and their interrelationships. Provides an architectural model that describes the organization of the system.

Used in the early stages of agile development, as refactoring architecture is much more expensive.

At the small scale, architecture can focus on individual components, while at a large scale, it can focus on how to organize complex enterprise systems.

Advantages of architectural design include increase stakeholder communication, better system analysis, and large scale reuse.

Architectural Representation

Informal block diagrams are often used to show entities and their relationships, but are often criticized for lacking semantics and not distinguishing different types of relationships between entities.

Stricture architectural models can also be used to facilitate discussion about the system design at a high level, as well as to document existing architectures.

Architectural Design Decisions

This is ultimately a creative process, so these decisions vary drastically based on what system is being developed, but a number of common ones span every process.

Architecture reuse can occur between systems that have similar domain requirements, in product lines where different products use the same core architecture, and as influence from existence architectures.

There are several metric for characterizing software architecture:

- **Performance**
- **Security**
- **Safety**
- **Availability**
- **Maintainability**

Architectural Views

This concerns the views and perspectives that go into designing a particular architecture. Individual models can only show one perspective, so multiple must be used to get a complete picture.

4+1 View Model

Requires a logical view, which shows key abstractions of the system, a process view, which shows how the system behaves at runtime, a development view, which shows how each component will be individually developed, and a physical view, which shows how software components are distributed across system hardware. In addition, use cases and scenarios are included. UML is widely used, but lacks abstractions for higher-level concepts.

Architectural Patterns

A stylized description of a design practice that has been tried and tested. Should have defined applications. For example, MVC pattern.

Layered Architecture

Models how sub-systems interact by organizing the system into layers, each of which provides its own service. Allows for replaceability but is often too artificial.

Repository Architecture

When large amounts of data need to be shared, the repository model is commonly used, as each sub-system maintains its own repository and can access it at any time.

Client-Server Architecture

This distributed model shows how data processing is distributed across components. Server components provide services to clients who call these components.

Pipe and Filter Architecture

Simply passes output from components to the inputs of other components, much like piping in UNIX. Most commonly used for data processing systems.

Application Architectures

Application architectures are designed to meet an organization's needs. Many business' application systems have common architectures. Generic architectures are consequently used. Common uses are as a starting point for design, as a design checklist, as a way to organize the developers, as a way to assess reuse of components, and as a method for discussing about the application itself.

Application types include data, transaction, event, and language process systems. Transaction and language processing systems are the most common.

Transaction Processing System

Processes user requests for information retrieval or updating. Any transaction is intended to satisfy a goal; transactions can be made asynchronously.

Information systems have a generic architecture that can be organized into layers. These transaction based-systems generally interact with databases. They may also be web-based, with the user interacting with the system through a web browser. Often implemented with client-server architectures.

Language Processing Systems

These accept both natural and artificial language as input and generate some representation of that input. Used in situations where a formal description of problem is difficult, e.g. programming, language interpreting, etc.

For example, language compilers are comprised of a lexical analyzer, a symbol table, a syntax analyzer, a syntax tree, a semantic analyzer, and a proper code generator.