4. System Requirements

Introduction

Certain types of requirements are so fundamental as to be common in most situations. Developing answers to a specific group of questions will help you understand these basic requirements.
There are also special kinds of requirements that arise, depending on whether the system is transaction or decision oriented and whether the system cuts across several departments. For instance, the need to inform the inventory manager of an unusually large order that is forthcoming underscores the importance of linking the sales, purchasing and warehouse departments.

4.1 Tools for System Requirements

Conceptually, requirements analysis includes three types of activity:

- ***Eliciting requirements:*** The task of communicating with customers and users to determine what their requirements are.
- ***Analyzing requirements:*** Determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then resolving these issues.
- ***Recording requirements:*** Requirements may be documented in various forms, such as *natural-language documents*, *use cases*, *user stories*, or *process specifications*.

New systems change the environment and relationships between people, so it is important to (1) ***identify all the stakeholders***, take into account (2) ***all their needs*** and (3) ***ensure they understand the implications of the new systems***.

Analysts can employ several techniques to elicit the requirements from the customer. Historically, this has included such things as ***holding interviews***, or ***holding focus groups and creating requirements lists***. More modern techniques include prototyping, and use cases.

4.1.1 Stakeholder Interviews

Stakeholder interviews are a common method used in requirement analysis. Some selection is usually necessary, cost being one factor in deciding whom to interview. These interviews may reveal requirements not previously envisaged as being within the scope of the project, and requirements may be contradictory. However, each stakeholder will have an idea of their expectation or will have visualized their requirements.

4.1.2 Joint Requirements Development (JRD) Sessions

Requirements often have cross-functional implications that are unknown to individual stakeholders and often missed or incompletely defined during stakeholder interviews. These cross-functional implications can be elicited by conducting JRD sessions in a controlled environment, facilitated by a Business Analyst, wherein stakeholders participate in discussions to elicit requirements, analyze their details and uncover cross-functional implications. A dedicated scribe to document the discussion is often useful, freeing the Business Analyst to focus on the requirements definition process and guide the discussion.

4.1.3 Contract-style Requirement Lists

One traditional way of documenting requirements has been <u>*contract style requirement lists*</u>. In a complex system such requirements lists can run to ***hundreds of pages***.

Measurable Goals

Best practices take the composed list of requirements only as clues and repeatedly ask “why?” Until the actual business purposes are discovered. ***Stakeholders*** and ***developers*** can then devise tests to measure what level of each goal has been achieved thus far. Such goals change more slowly than the long list of specific but unmeasured requirements. Once a small set of critical, measured goals has been established, rapid prototyping and short iterative development phases may proceed to deliver actual stakeholder value long before the project is half over.

Prototypes

In the mid-1980s, prototyping was seen as the solution to the requirements analysis problem. Prototypes are mock-ups of an application.

Mock-ups allow users to visualize an application that hasn’t yet been constructed. Prototypes help users get an idea of what the system will look like, and make it easier for users to make design decisions without waiting for the system to be built.

Major improvements in communication between users and developers were often seen with the introduction of prototypes. Early views of applications led to *fewer changes later* and hence *reduced overall costs* considerably.

However, over the next decade, while *proving a useful technique*, prototyping did not solve the requirements problem:

- ***Managers***, once they see a prototype, may have a hard time understanding that the finished design will not be produced for some time.
- ***Designers*** often feel compelled to use patched together prototype code in the real system, because they are afraid to 'waste time' starting again.
- ***Prototypes*** principally help with design decisions and user interface design. However, they can't tell you what the requirements originally were.
- ***Designers*** and end users can focus too much on user interface design and too little on producing a system that serves the business process.

Prototypes can be flat diagrams (referred to as 'wireframes') or working applications using synthesized functionality. Wireframes are made in a variety of graphic design documents, and often remove all color from the software design (i.e. use a greyscale color palette) in instances where the final software is expected to have graphic design applied to it. This helps to prevent confusion over the final visual look and feel of the application.

Uses of System Prototypes

- The major utilization of system prototypes is to assist consumers and developers recognize the requirements for the system.
  - ***Requirements elicitation:*** Users can research with a prototype to observe how the system assists their work.
  - ***Requirements validation:*** The prototype can disclose errors and blunders in the requirements.
- Prototyping can be regarded as a risk lessening activity which decreases requirements risks.

Prototyping Benefits

The benefits of prototyping are given as below:

- Misinterpretations among software users and developers are depicted
- Missing services may be identified and perplexing services may be recognized
- A working system is obtainable early in the procedure
- The prototype may provide as a foundation for obtaining a system specification
- The system can sustain user training and system testing.

Prototyping leads to the following:

- Enhanced system usability
- Closer match to the system required
- Enhanced design quality
- Enhanced maintainability
- Abridged on the whole development attempt

Prototyping in the software process: There are two types of prototyping used in the software process.

- ***Evolutionary prototyping:*** A method to system development where an initial prototype is generated and developed via a number of stages to the concluding system.
- ***Throw-away prototyping:*** A prototype which is generally a practical implementation of the system is generated to aid find out requirements problems and then not needed. The system is then produced by means of some other enlargement process.

The purpose of evolutionary prototyping is to provide a working system to end-users. The development begins with those requirements which are best recognized. Evolutionary prototyping must be utilized for systems where the specification cannot be produced in advance such as AI systems and user interface systems. Evolutionary prototyping is based on techniques which permit rapid system iterations. Verification is impracticable as there is no specification. Validation specifies illustrating the competence of the system

In case of Evolutionary prototyping:

- Specification, design and implementation are entangled
- The system is produced as a series of increment that are provided to the customer
- Methods for rapid system development are used like CASE tools and 4GLs
- User interfaces are generally produced by means of a GUI development tool kit.

The purpose of throw-away prototyping is to authenticate or obtain the system requirements. The prototyping process begins with those requirements which are unsuccessfully recognized. Throw-away prototyping is used to decrease requirements risk. The prototype is produced from an initial specification, provided for experiment then discarded.

The throw-away prototype should not be regarded as a final system because:

- Some system traits may have been left out.

- There is no specification for long-term preservation.
- The system will be badly structured and complicated to preserve.

Use Cases

A **use case** is a technique for documenting the *potential requirements* of a new system or software change. Each use case provides one or more scenarios *that convey how the system should interact with the end user or another system to achieve a specific business goal*.

Use cases typically avoid technical jargon, preferring instead the language of the end user or domain expert. Use cases are often co-authored by requirements engineers and stakeholders.

Use cases are deceptively simple tools for describing the behavior of software or systems. A use case contains a textual description of all of the ways which the intended users could work with the software or system. Use cases do not describe any internal workings of the system, nor do they explain how that system will be implemented. They simply show the steps that a user follows to perform a task. All the ways that users interact with a system can be described in this manner.

During the 1990s, use cases rapidly became the most common practice for capturing functional requirements. This is especially the case within the object-oriented community, where they originated, but their applicability is not restricted to object-oriented systems, because use cases are not object-oriented in nature.

A use case defines interactions between external actors and the system under consideration, to accomplish a business goal. Actors are parties outside the system that interact with the system; an actor can be a class of users, a role users can play, or another system.

Use cases treat the system as a black box, and the interactions with the system, including system responses, are perceived as from outside the system. This is deliberate policy, because it simplifies the description of requirements and avoids the trap of making assumptions about how this functionality will be accomplished.

A use case should:

- describe a business task to serve a business goal
- be at an appropriate level of detail
- be short enough to implement by one software developer in a single release