TASK

System Requirements and Design

Visit our website

Introduction

WELCOME TO THE SYSTEM REQUIREMENTS AND DESIGN TASK!

This task explores the best practice guidelines for defining your product — including gathering and documenting system requirements. It also introduces user interface (UI) and user experience (UX) design guidelines.

REQUIREMENTS

The concept of requirements is fairly self-explanatory: they are what we need a system to be able to do. For example, if we want a system to organise our files on our computer, the system's activities will include reading file names, sorting them (e.g. in alphabetical order) and putting them in neat folders (A-E, F-J, etc.). Part of the requirements is also looking at the system's constraints — what it must not do. For example, we may only want to sort our Word documents, not program files.

There are two categories of requirements:

1. **User requirements:** what the user finds important. These are broad ideas of what the program needs to accomplish. They are written in simple language for anyone to understand.
2. **System requirements:** these are the detailed versions of the user requirements. Here, technical jargon is used so that the developers understand the exact functions and constraints of the system because it is the starting point of system design. These requirements are broken down further:
    a. **Functional requirements:** the activities that are required of the program.
    b. **Non-functional requirements:** characteristics of the system as a whole, including its constraints, but excluding the activities covered in the functional requirements.

FUNCTIONAL REQUIREMENTS

These requirements depend on the type of software being developed, the expected users of the software, and the approach taken by the organisation when writing requirements. For example, if you are developing a payroll system, the

required business uses might include functions such as *generating electronic fund transfers*, *calculating commission amounts*, *calculating payroll taxes*, and *maintaining employee-dependent information*. The new system must handle all of these functions.

Functional requirements are based on the procedures and rules that the organisation uses to run its business. They are sometimes well-documented and easy to identify and describe, however, this is not always the case. Some business rules can be more obtuse or difficult to find.

NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements are requirements that are not directly concerned with the specific activities a system must perform or support. Some non-functional requirements can be more critical than individual functional requirements. When they are not critical, system users often find ways to work around a system function that doesn't meet their non-functional needs.

It is not always easy to distinguish between functional and non-functional requirements, but there is a framework you can use for identifying and classifying requirements. The most widely used framework is called FURPS+. FURPS is an acronym that stands for **F**unctionality, **U**sability, **R**eliability, **P**erformance, and **S**ecurity. Functionality refers to functional requirements, while the remaining categories describe non-functional requirements:

- **Usability:** these requirements describe operational characteristics related to users, such as the user interface, related work procedures, online help, and documentation.

- **Reliability:** these requirements describe the dependability of a system. In other words, reliability is to do with how a system detects and recovers from such behaviours as service outages and incorrect processing.

- **Performance:** these requirements describe operational characteristics related to measures of workload, such as throughput and response time.

- **Security:** these requirements describe how access to the application will be controlled and how data will be protected during storage and transmission.

FURPS+ is an extension of FURPS that adds additional categories — all of which are summarised by the plus sign. Below is a short description of each additional category:

- **Design constraints:** these describe restrictions to which the hardware and software must adhere.

- **Implementation requirements:** these describe constraints such as required programming languages and tools, documentation methods and level of detail, and a specific communication protocol for distributed components.

- **Interface requirements:** these describe interactions among systems.

- **Physical requirements:** these describe the characteristics of the hardware such as size, weight, power consumption, and operating conditions.

- **Supportability requirements:** these describe how a system is installed, configured, monitored, and updated.

SOFTWARE REQUIREMENTS DOCUMENT

Once the user and system requirements have been determined, they are written up in an official requirements document during the requirements specification process (discussed below). This is a universal document that everyone from the users to the system developers will see. Therefore, it is vital that the document is written to cater for all stakeholders — it needs to be general enough for the user to understand but detailed enough for the developers to use as a springboard for development. This is why the user requirements will take the form of the introduction, and the system requirements based on that will become the body of the text.

The IEEE (1998, as cited in Sommerville, 2016, p. 128) outlines the structure of a standard requirements document:

Chapter	Description
Preface	Defines who you expect to read the document and describes its version history, which should include the reason for creating the new version and a summary of the changes made in each version.
Introduction	Describes why the system is needed. It also describes the system's functions and explains how it will work with other systems. Describes how the system fits into the business overall or the strategic objectives of the organisation that commissioned the software.
Glossary	Defines the technical terms used in the document.
User requirements definition	The services provided for the user are defined here. The non-functional requirements are described. Product and process standards that must be followed must be specified.
System architecture	Presents a high-level overview of the system's anticipated architecture by showing the distribution of functions across system modules. Highlights any architectural components that are reused.
System requirements specification	Describes the functional and non-functional requirements in more detail. Interfaces to other systems can also be defined.
System models	Might include graphical system models that show the relationship between system components, the system, and its environment. Some examples of models are object models, data-flow models, and semantic data models.
System evolution	Describes the fundamental assumptions on which the system is based. Describes any anticipated changes due to hardware evolution, changing user needs, etc.
Appendices	Provide detailed, specific information that is related to the application that is being developed. For example, hardware and database descriptions.
Index	Several indexes might be included. This might include a normal alphabetic index, an index of diagrams, an index of functions, etc.

The information that is included in a requirements document depends on the type of software being developed and the approach to development that will be used.

REQUIREMENTS ENGINEERING PROCESS

Requirements engineering is the process of finding, analysing, documenting, and checking the requirements of a system (Chemuturi, 2013). This means that it would be up to the Requirements Engineers to oversee the process from start to finish, beginning with the feasibility study. This, as the name suggests, consists of analysing the market, the user requirements, the system requirements, and the possible constraints to determine if creating this system is a worthwhile venture.

Once it has been decided that the system is worth developing, the engineering process follows three broad steps, according to Sommerville's (2016) model:

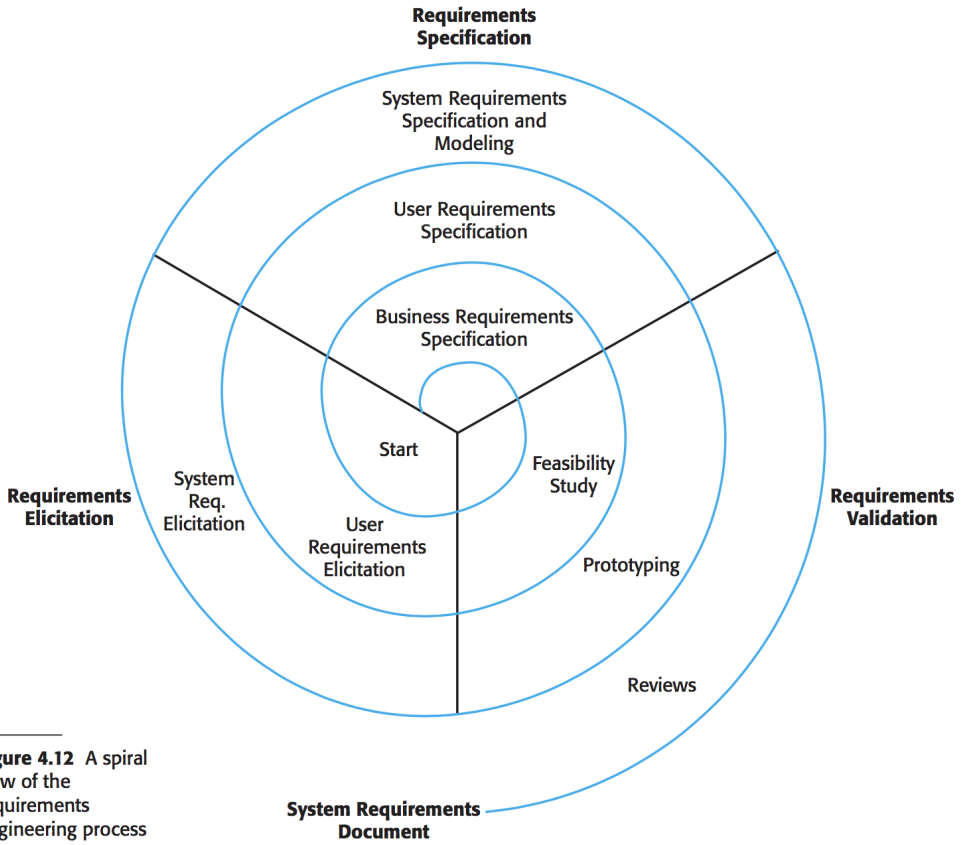


Figure 4.12 A spiral view of the requirements engineering process

A spiral view of the requirements engineering process (Sommerville, 2016, p. 112)

The process begins in the bottom left third in Requirements Elicitation and works its way round in a clockwise direction. Each section is discussed in more detail below:

1. **Requirements elicitation:** determine how the system will be used by the business by interviewing those who will be using the system. Scenarios are also discussed to get a better idea of what the system is required to do and how it will work. This is generally where user requirements are determined.
2. **Requirements specification:** write the requirements determined from elicitation into the requirements document described above. Use cases are also used in this step to illustrate how a user might interact with a system. These interactions are often represented in a Unified Modelling Language (UML) use case diagram.
3. **Requirements validation:** determine that the requirements documented contain everything that the user wants it to do. This is a vital step as sorting out miscommunications before the development process saves both time and money.

Note how the process moves in a spiral through the thirds in an outward direction. This shows the iterative nature of the process — we start by looking at the general user requirements, then we continue going around the spiral to look at the more specific system requirements. This process continues until all stakeholders are happy with the requirements specified.

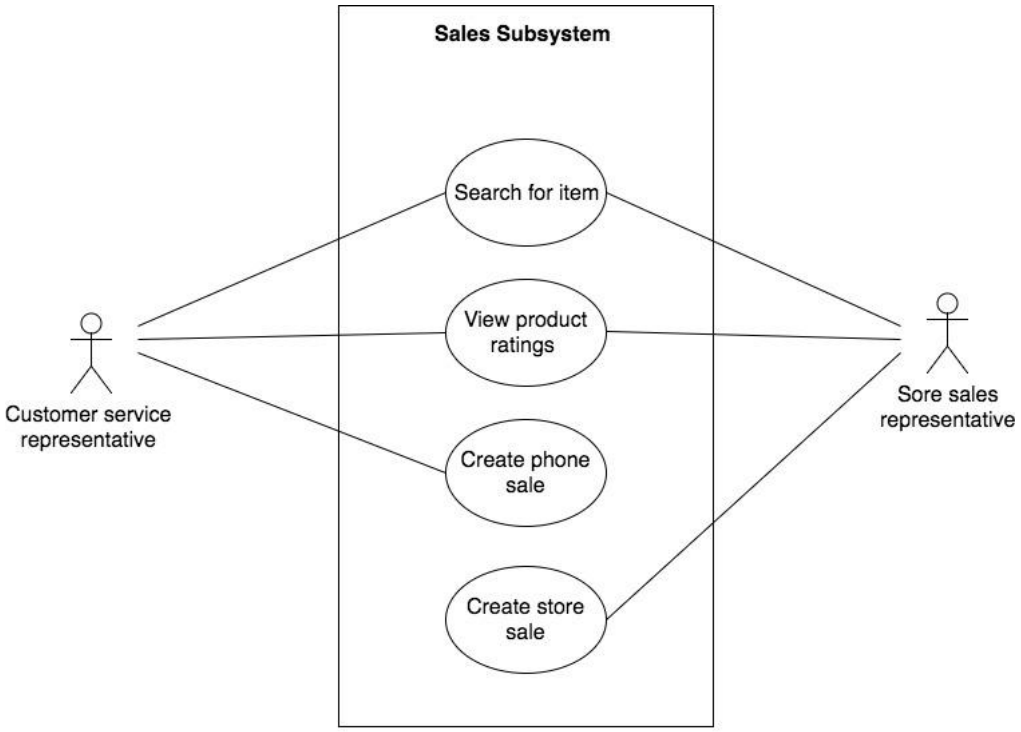
USE CASES

A use case is an activity the system performs, usually in response to a request by a user. It identifies the actors (people or groups external to the system that interact with the system by supplying or receiving data) involved in an interaction and names the type of interaction. The set of use cases represents all of the possible interactions described in the system requirements.

Each use case should be documented with a brief description that can then be used to create other UML diagrams. The table below shows some examples of brief use case descriptions:

Use case	Brief use case description
Create customer account	The actor enters new customer account data and the system assigns the customer an account number, creates a customer record, and creates an account record.
Look up customer	The actor enters the customer's account number, then the system retrieves and displays all customer account data.
Process account adjustment	The actor enters an order number and the system retrieves customer and order data. The actor enters the adjustment amount, and the system creates a transaction record for the adjustment.

Use cases are documented using a UML use case diagram. Actors in the process, who may be human or other systems, are represented as stick figures and the use case itself is represented by an oval with the name of the use case inside. Lines link the actors with the interaction or use case. The following diagram is an example of a UML use case diagram:



There is no clear distinction between scenarios and use cases. Some people consider that each use case is a single scenario, while others encapsulate a set of scenarios in a single use case. Scenarios and use cases are effective techniques for eliciting requirements from stakeholders who interact directly with the system. However, they are not as effective for determining constraints for high-level business and non-functional requirements or for discovering domain requirements, as they focus mainly on interactions with the system.

PROTOTYPING

A prototype is essentially a quickly built, stripped-down version of a system. Its purpose is to be able to show stakeholders a basic mock-up of the system so they can see if development is heading in the right direction. The prototype is developed quickly and cost-effectively so that it does not waste resources, but it needs to be developed enough to use for design experiments and for stakeholders to check if all their requirements for the system are being met. If not, the prototype is developed early enough in the process that it is not overly expensive or time-consuming to change something if need be. Prototyping is an important part of user interface design.

USER INTERFACE DESIGN

One particular non-functional requirement is knowing how the user will interact with the system — the user interface (UI). If you think of using Google Maps on your phone, you don't really care about the inner workings of the app, you just want to know how to get from A to B. That's the user interface — where the user and the system interact. Because most users of a system are not well-experienced in the inner workings of software and technology, it is very important that the system is intuitively designed and easy to use. The designing of this involves anticipating what a user might try to do in order to achieve a specific outcome. For example, will they swipe right if they want the menu? What might they do if they want to cancel their current journey? Would the user be confused if the "start" button were red instead of green? All of these types of questions need to be answered and accounted for to make a system usable.

ELEMENTS OF UI DESIGN

If we go to Google Maps, let us see what UI elements are present (Garrett, 2011):

1. **Input controls:** this is how the user provides information.
    a. Dropdown list that pops up when you start typing
    b. Checkboxes to select what map view you want to see
    c. Button to look at directions
2. **Navigational components:** this helps users navigate through the program.
    a. Search field to search for a location
    b. Icons to show restaurants, attractions, etc.
    c. Image carousel of nearby places to explore
3. **Informational components:** these provide output from the system.
    a. Notifications of potential traffic delays
    b. Progress bar when traffic information is being updated

BEST PRACTICE FOR UI DESIGN

The best programs are designed in such a way that we don't have to learn how to use them. Most of the following may seem obvious when you read it, but they can be what make or break a program's usability:

1. **Keep it simple.** Don't add a whole lot of unnecessary things just to take up space — it can get confusing for the user.
2. **Use common UI elements consistently.** Where would you normally go to close a window on your computer? It's never in the centre of the screen, right? That is because having the close button in one of the top corners of the screen is a common UI element we have all become accustomed to. Having these elements and keeping them consistent means the user can apply prior experience to your program rather than learning something completely new.
3. **Use colours, textures and typography purposefully.** Remember when you first discovered WordArt in Microsoft Word and proceeded to have rainbow headings with a drop shadow for everything? Good UI uses colours

and textures to convey a message — green to start, red to stop, etc. Typography can be useful when showing different levels of hierarchy, e.g. a big heading is the main heading, and smaller headings are subheadings.

4. **Communicate.** There is nothing scarier than making an online payment and then the webpage suddenly freezes. Users feel calm when they know what's going on, so convey the problem to them in a notification.

USER INTERFACE VS USER EXPERIENCE

You may have heard the term "user experience" (UX) before — it is often in a context where it seems interchangeable with "user interface" (UI). While the two do work closely together, they are actually two distinct design processes with different priorities. Put simply, UI deals specifically with the design elements used in the interface — colour, typography, spacing, text boxes, search bars, etc. On the other hand, UX is more abstract: it deals with the *feel* of using the system — its flow.

As elegantly put by Dain Miller (Web Developer):

"UI is the saddle, the stirrups, and the reins. UX is the feeling you get being able to ride the horse."

Take note:

Have you heard of wireframing? This is a basic way that UX designers design how a program is going to work and flow from one part of the program to another. For more information, have a look at this discussion on wireframing **here**.

Compulsory Task 1

Answer the following questions:

- Consider the following statement of requirements for part of a ticket-issuing system:

  An automated ticket-issuing system sells rail tickets. Users select their destination and input a credit card and a personal identification number. The rail ticket is issued and their credit card account is charged. When the user presses the start button, a menu display of potential destinations is activated, along with a message to the user to select a destination. Once a destination has been selected, users are requested to input their credit card. Its validity is checked and the user is requested to input a personal identifier. When the credit transaction has been validated, the ticket is issued.
  Based on this:

  - Discuss any ambiguities or omissions in the statement of requirements for this part of a ticket-issuing system.
  - Write a set of non-functional requirements for the ticket-issuing system above. You are free to make assumptions regarding the system based on ambiguities or omissions you identified previously.

Compulsory Task 2

- Create a text file called **UserInterfaces.txt** and answer the following questions.
- Go to **this website**. Do you think the website has a good UI? Why or why not? Refer to the information given in the task and discuss both good and bad elements in the website's UI.
- Now go to **this website**. Do you think the website has a good UI? Why or why not? Refer to the information given in the task and discuss both good and bad elements in the website's UI.

Completed the task(s)?

Ask an expert code reviewer to review your work!

Review work

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCES

Chemuturi, M. (2013). *Requirements engineering and management for software development projects*. New York: Springer Science & Business Media.

Garrett, J. (2011). *The elements of user experience*. Berkeley: New Riders.

Sommerville, I. (2016). *Software Engineering* (10th ed., pp. 101-137). Essex: Pearson Education Limited.