Requirements

A software **requirement** is a user need the software is meant to fulfill. Once specified, they define what software teams should implement.

requirement: A user need the software is meant to fulfill.

non-functional requirement: A requirement that states how well the software should perform.

functional requirement: A requirement that states what the software should do (its feature set).

2.3 Types of Requirements

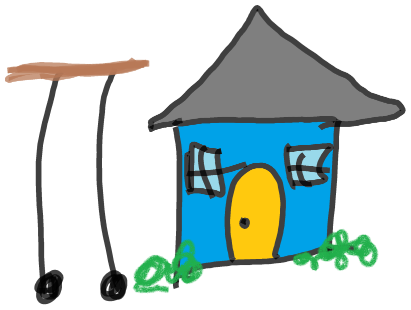
There are two types of requirements:

1. **Non-functional requirements** specify qualities the software should have (e.g., usable, portable, modular, etc.). They answer the question, "How well should the software perform?" This chapter includes a discussion of how quality attributes can be used in specifying non-functional requirements.

2. **Functional requirements** specify the desired functionality of software (e.g., if I click the Log In button, the

Login page appears). They answer the question, “What should the software do?” In this chapter, we’ll talk about specifying functional requirements with user stories and use cases.

User stories and use cases are two different methods for specifying functional requirements. As you will see later, one is more formal than the other.



*This rolling table fails the non-functional requirement of fitting through an average door and the functional requirement of having four legs. **placeholder*

2.4 Why Requirements Matter

The design and implementation of software should, ideally, follow from the requirements. Here are some ways requirements are helpful and reasons they are important:

Requirements keep the development team on track and working together toward creating what the client (and hopefully the users) want.

- When developers aren’t given requirements, they **tend to design and write functionality they personally think is important or fun** to implement. What developers want to implement is not necessarily what will make the project successful.
- When multiple developers are working on the same code, requirements can **help them stay in sync** with one another and **have the same goal**. Without requirements, time, effort, and money can be wasted implementing conflicting code.

Requirements can help protect projects from drift and failure.

- When requirements aren’t specified, it’s easier for project stakeholders (e.g., **clients**, partners, investors, consultants, management, etc.) to **influence the project toward satisfying their own wants or needs**. This can result in the project drifting away from what it was originally intended to do—and can lead to project failure.
- Requirements are **very helpful for communicating** about te software with stakeholders, **keeping track** of everything that needs to get done, and helping you and the

client **decide what really needs to get done** (clients sometimes don't know what they really need).

client (a.k.a. customer): One or more people or organizations who are requesting the software be made and have decision-making authority about the software (e.g., because they are paying for it or otherwise providing resources).

2.5 What Makes a Good Requirement

Good requirements have the following characteristics:

Correct	What they say is right.
Consistent	They aren't contradictory of each other.
Unambiguous	There is only one way to interpret them.
Complete	They cover all that's important.
Relevant	They meet a stakeholder need.
Testable	There's a way to figure out if they're satisfied.
Traceable	It's possible to figure out where they came from.

Sloppy requirements can be useless or worse.

stakeholder: Anyone who is or will be affected by the software or its development (e.g., clients, companies, users, developers, managers, politicians, etc.)

Requirements that fail to have these characteristics can lead developers to making features of software nobody wants, wasting time and other resources and potentially jeopardizing the project.

2.6 Requirements Elicitation

requirements elicitation: The process of gathering requirements from project stakeholders.

The process of gathering requirements is called **requirements elicitation**. Requirements can come from any stakeholder, including clients, managers, users, governments, developers of software your software will integrate with, your development team, and yourself.

Three of the most important, distinct, and universal (common across projects) categories of stakeholders:

- **Clients**: The people who request the software and have most of the authority over its requirements (e.g., because they are paying for it).
- **Users**: The people who will use the software.
- **Developers**: The people who will make the software, including those who manage the software engineers.

Aspects of these stakeholders that can affect the requirements elicitation processes and the software's development and ultimate success:

- **Clients might not have experience or expertise**. Developers can help fill the gap between what the client wants and what is technically feasible and reasonable (e.g., given time, cost, and scope, a.k.a. the **triple constraint**).
- **Clients might not have good ideas**. They may be incorrect about what users will want or will use. Developers sometimes try to guide clients toward better ideas, but developers can also have bad ideas. Techniques such as **focus groups**, **usability testing**, and releasing a **minimum viable product** (MVP) can help with figuring out whether users will use (and pay for) the software.
- **Clients might not know what they want**. They might have a rough idea, or they might have an idea that's at odds with their goals. Developers, through requirements elicitation, can help clients define their goals clearly and reasonable ways for accomplishing those goals.
- **Users might not know what they want or will use**. They may be unaware of their own needs or wants until there's a product in front of them that addresses those needs or wants. Even if 10,000 users tell you, "I would definitely use an app that does X", they might be wrong, they might only use the app once, or they might not be willing to pay for the app. MVP can be a good technique for figuring out early whether users will be interested enough in the software to use or pay for it.
- **Users might want what's bad for them**. You can probably think of multiple examples.
- **Developers have their own tendencies**. They may have technologies, approaches, and ideas they prefer or feel most comfortable with. For better or worse, they bring their own influences to a project.
- **Clients, users, and developers are all humans**. They communicate imperfectly.

Deciding what software to make, and doing so successfully, is a complex and process influenced by human factors affecting all involved.

So how does one elicit requirements? By having conversations or otherwise collecting information from stakeholders. The amount of stakeholder communication can vary by project,

triple constraint: A model of what's important in project management: time, cost, scope.

focus group (in usability engineering): A moderated discussion between researcher and a small number of potential users (usually 6-12) during which the researcher tries to gather information about the participants' attitudes, opinions, motivations, concerns, and problems related to a specific product or topic.(Odimegwu 2000)

usability testing: Observing people while they try to use your software.(Barnum 2020)

minimum viable product (MVP): A low-effort or low-expense effort that accurately tells you whether people will want to use your product—before the product is fully developed.(Olsen 2015)

software process model: A philosophy and/or set of approaches for software development and/or software project management.

project type, the **software process model** being used, and other factors.

2.7 Non-Functional Requirements

Non-functional requirements describe how well the software needs to perform.

non-functional requirement: A requirement that states how well the software should perform.

Examples of non-functional requirements:

- Response time should be a few seconds or less in all operating environments.
- The keylogger must be indetectable to 99.999% of test users.
- The software must be available 24 hours a day, 7 days a week, and must have an uptime of 99%.

Notice that each requirement has a quantity associated with it: That makes it testable (one of the criteria for a good requirement).

2.7.1 Quality Attributes

Quality attributes are characteristics of software used to describe how good it is. They can be used in specifying non-functional requirements.

quality attribute: A characteristic of software used to describe how good it is.

Examples of quality attributes:

- **Reliability**: How often does function X succeed?
- **Efficiency**: How many resources does the software need?
- **Integrity**: How frequently does the software have errors that require a restart?
- **Memorability**: How many times must users learn a function before they no longer need documentation?
- **Flexibility**: How many ways can the software be used?
- **Interoperability**: How well can the software integrate with other software?
- **Reusability**: To what extent can the code be used to solve other problems without being modified?

Each quality attribute can be converted to a scale. For example, the lowest value on a reliability scale could be "the function succeeds 0% of the time" and 100% would of course be the opposite pole. Given this scale, we can specify a non-functional

A quality attribute is not the same as a non-functional requirement. Rather, quality attributes are good for labelling what a non-functional requirement is about.

requirement by defining a performance threshold:

> The function must have high reliability (succeeds >99% of the time).

When you select quality attributes for your software to have, you are prioritizing what qualities matter most to you / your team / the project. Ideally, your team would keep these quality attributes (and the corresponding non-functional requirements) in mind for the duration of the project; If the software is not meeting the non-functional requirements, either the software or the threshold of acceptability needs to change.

2.8 Functional Requirements

functional requirement: A requirement that states what the software should do (its feature set).

Functional requirements describe what functionality the software needs to have.

Example of a functional requirement:

> When a user clicks the “register” button, their information is added to the database and the user is shown a “thank you for registering” screen.

2.8.1 User Stories

User stories are a method for specifying functional requirements. They describe a small piece of the software’s functionality in a simple and easy to read sentence. They are written in plain English so that non-technical people (e.g., users, clients, other stakeholders) can understand them.

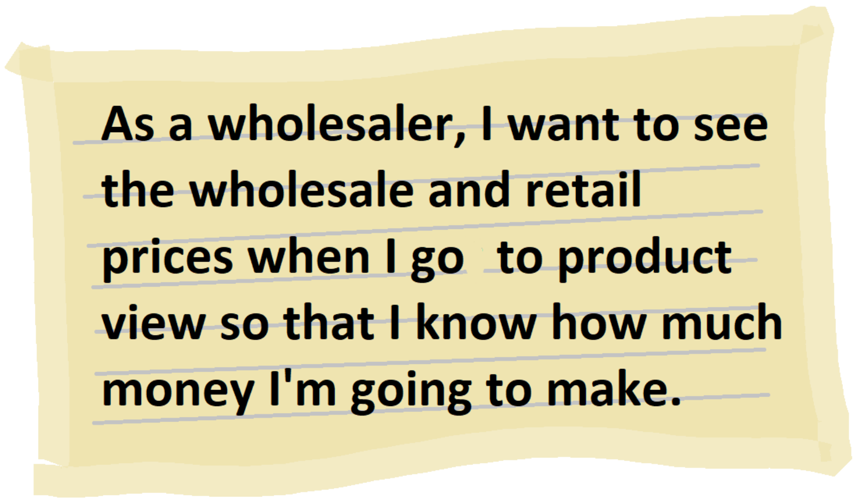
user story: “Short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.” (Cohn n.d.)

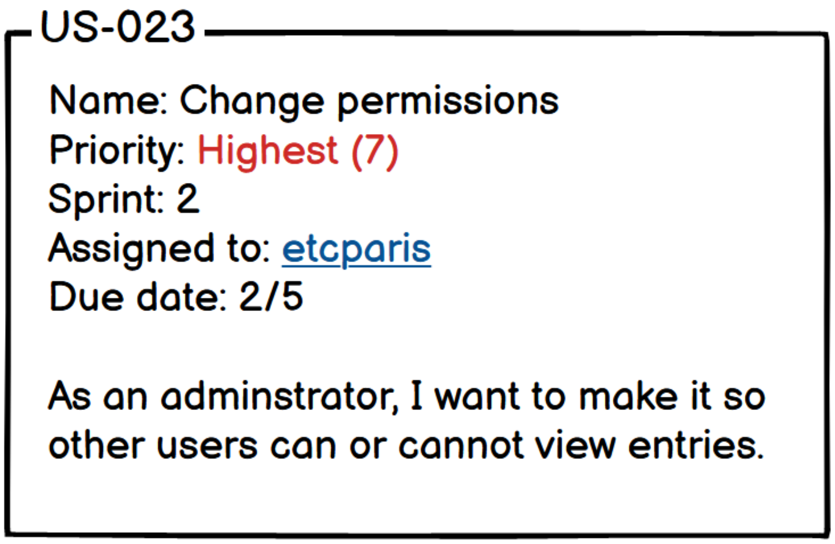
User stories have a title and are commonly written using this format:

> As a ⟨ROLE⟩, I want ⟨SOME FUNCTIONALITY⟩
> so that I get ⟨SOME BENEFIT⟩.

These short sentences are often written on 3x5” index cards and then stuck on a wall or whiteboard. They can also be typed into task and project management systems (e.g., Jira, Asana, etc.).

Examples of what user story cards can look like:


As a wholesaler, I want to see the wholesale and retail prices when I go to product view so that I know how much money I'm going to make.


US-023

Name: Change permissions
Priority: Highest (7)
Sprint: 2
Assigned to: etcparis
Due date: 2/5

As an adminstrator, I want to make it so other users can or cannot view entries.

More examples of user stories: `https://web.archive.org/web/20201124004807/https://www.mountaingoatsoftware.com/agile/user-stories`

Anyone on the team—or any project stakeholder—might come up with user stories. Once the user stories are initially defined, they can be used to start a conversation with the client and others on the team. Clients can guide you on setting priorities for user stories. This conversation is also a good time to get more details about the user stories, which should be added to the card.

INVEST: Characteristics of good user users (independent, negotiable, valuable, estimable, small, testable) (Wake 2003).

Characteristics of good user stories (INVEST):

I	**Independent**: Doesn't depend on other user stories.
N	**Negotiable**: Can be changed during development.
V	**Valuable**: Fulfills a user need.
E	**Estimable**: Can be given a time estimate.
S	**Small**: Can fit into a single development period (e.g., a 2-week Sprint)
T	**Testable**: Possible to determine it's done.

There is some overlap between INVEST and the general characteristics of good requirements mentioned earlier in this chapter.

acceptance criterion: A statement about functionality that, when satisfied, mean the functionality has been satisfactorily implemented.

Definition of Done (DoD): The set of acceptance criteria which, once satisfied, mean a user story has been satisfactorily implemented.

How do you know when you are done with a user story? This is negotiated with the client and added to the user story as acceptance criteria. Acceptance criteria say what must be true about the functionality specified by the user story in order for the user story to be considered done (i.e., establishing the **Definition of Done** for the user story).

Example acceptance criterion:

> **Given** the user is playing a video file **and** their operating system is Windows, **when** they do the Ctrl-T keyboard shortcut **then** they will see the "Go to Time" screen **and** the video will pause.

There are bolded words in that example because its using a common format (Alliance n.d.) for acceptance criteria:

> Given . . . when . . . then . . .

The "and"'s are optional parts of the format. Ideally, acceptance criteria testing can be automating.

Example pseudocode for testing acceptance criteria:

```
def test_go_to_time():
  # given
  assert os.isWindows(),"Not Windows!"
  player.open()
  player.play_video('test.mkv')
```

```

# when
user.send_keyboard_shortcut("Ctrl-T")

# then
assert player.screen.is_showing(GOTOTIME)
```

2.8.2 Use Cases

Use cases are a more formal method of specifying functional requirements. They are structured descriptions of what a system is required to do when a user interacts with it.

use case: "A contract for the behavior of the system under discussion" (Cockburn 2001)

Use cases are not specific to a particular software process model (e.g., Agile, Waterfall, Spiral) or environment. Instead, like much of what you will encounter in this book, they are a well-known approach software teams can choose to use (and many do), or not.

More examples of use cases: `https://web.archive.org/web/20201111211938/https://www.usability.gov/how-to-and-tools/methods/use-cases.html`

Example of a use case:

- **Name**: Generate list of recovered patients
- **Actor**: Clinician
- **Flow**:
  1. Clinican authenticates using smart card
  2. Software confirms credentials and access permissions for specific machine
  3. Software logs access
  4. Software displays patient search
  5. Clinician selects "Advanced Patient Search"
  6. Software confirms user access permissions for advanced search page
  7. Clinician selects ailment and patient status
  8. Clinician executes search using "Search" button
  9. Software returns results
  10. Software logs query

Required Parts of a Use Case

What every valid use case has:

- **Name**: A short title for the use case that often starts with a verb (e.g., Schedule weekly wellness check). Briefly states the user objective the use case will be describing.
- **Actors**: The user or users (human / non-human / computer) that are interacting with the software (e.g., Medical staff)
- **Flow of events** (a.k.a. "basic course of action" or "success scenario"): Sequence of actions describing the interaction between the actor and the software.

Sometimes, the actor is implied through the flow of events (e.g., Shopper selects the calendar icon). Other times, the actor is stated separately from the flow of events (e.g., Actor: Shopper).

The correct amount of detail to give a use case is the minimum amount to adequately describe what you're trying to communicate.

Additional Parts of a Use Case

Sometimes included in use cases:

- **Identifier**: A unique way of referring to the use case (e.g., UC-002)
- **Pre-conditions**: What must be true before the flow (e.g., The shopper has added at least one product to their shopping cart.)
- **Post-conditions**: What must be true after the flow (e.g., The shopper received an order confirmation email.)
- **Business relevance**: Justification for why the use case exists
- **Dependencies**: Other use cases the use case relies on. This unique identifier is handy for this part.
- **Extensions**: Contingencies, alternate routes, and branches to other use cases
- **Priorities**: The importance of the use case
- **Non-functional requirements**: How well the software must perform during the flow

2.9 Requirements Specification

requirements specification: Converting stakeholder requests into written requirements.

The process of writing down requirements is called **requirements specification**. Used as a noun, requirements specification refers to the document that contains the requirements. That document is also called an **SRS** (Software Requirements Specification). The best way to understand what an SRS looks like is to look at some.

Freely available SRS examples (including some for open source software):

- **SRS for apps and a data repository for distributing manufacturing data**: Thomas Hedberg Jr., Moneer Helu, and Marcus Newrock (Dec. 2017). *Software Requirements Specification to Distribute Manufacturing Data.* `https://doi.org/10.6028/NIST.AMS.300-2`

- **SRS for data system that assesses conservation practices**: Data System Team (n.d.). *System Requirements Specification for STEWARDS.* `https://web.archive.org/web/20200923200038/https://www.nrcs.usda.gov/Internet/FSE_DOCUMENTS/nrcs143_013173.pdf`

- **SRS for an app that splits and merges PDFs**: Ploutarchos Spyridonos (Feb. 2010). *Software Requirements Specification for PDF Split and Merge, Version 2.1.0.* `https://web.archive.org/web/20170225043950/http://selab.netlab.uky.edu/%7Eashlee/cs617/project2/PDFSam.pdf`

- **SRS for software that processes EEG data**: Inria Innovation Lab (n.d.). *Software Requirement Specification for CertiViBE, v1.0.* `https://web.archive.org/web/20190710221933/http://openvibe.inria.fr/openvibe/wp-content/uploads/2018/04/CERT-Software-Requirement-Specification.pdf`

- **SRS for library software**: Fred Eaker (Nov. 2006). *Software Requirements Specification for Vyasa.* `https://web.archive.org/web/20161127184329/http://vyasa.sourceforge.net/vyasa_software_requirements_specification.pdf`

Software Requirements Specification (SRS): A document that contains software requirements.

Another type of software document, which sometimes gets confused with an SRS, is a Software Design Document (SDD). If the SRS is what the software *should* do, the SDD is what the software *is*. However, there is often overlap between the two.

References

- Agile Alliance (n.d.). *What is "Given - When - Then?"* https://web.archive.org/web/20201124202211/https://www.agilealliance.org/glossary/gwt
- Roger Atkinson (1999). "Project management: cost, time and quality, two best guesses and a phenomenon, its time to accept other success criteria". eng. In: *International journal of project management* 17.6, pp. 337–342. ISSN: 0263-7863
- Carol M. Barnum (2020). *Usability Testing Essentials: Ready, Set...Test!* 2nd ed. Morgan Kaufmann. ISBN: 0128169427,9780128169421. URL: http://gen.lib.rus.ec/book/index.php?md5=85EE81383B3A0CDDD57DB26FE50F59AB
- Alistair Cockburn (2001). *Writing effective use cases.* Boston
- Mike Cohn (n.d.). *User Stories and User Story Examples.* https://web.archive.org/web/20201124004807/https://www.mountaingoatsoftware.com/agile/user-stories
- Martin Fowler (2004). *UML distilled : a brief guide to the standard object modeling language.* Boston
- Clifford Odimegwu (July 2000). "Methodological Issues in the Use of Focus Group Discussion as a Data Collection Tool". In: *Journal of Social Sciences* 4, pp. 207–212. DOI: 10.1080/09718923.2000.11892269
- Dan Olsen (2015). *The lean product playbook : how to innovate with minimum viable products and rapid customer feedback.* Hoboken: Wiley. ISBN: 9781118961025
- Rebecca Parsons (June 2003). "Components and the world of chaos". In: *Software, IEEE* 20, pp. 83–85. DOI: 10.1109/MS.2003.1196326
- Bill Wake (Aug. 2003). *INVEST in Good Stories, and SMART Tasks.* https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/. Accessed: 2020-12-31