Requirements

Objectives

Software is developed with specific requirements in mind, not for the sake of developing software. Discovering what users' requirements are is a key part of software engineering, since all other engineering steps are not useful if the software being built does not do what users want. This includes discovering requirements that are implicit to users, even if their context is different from yours. As a software engineer, you need to define requirements in cooperation with users and evaluate them to prioritize the ones users want (or need?) most. While you could check that your code matches the requirements only after having written it, using the requirements during development will save you time and improve the readability of your code by using context-specific vocabulary and concepts.

In this module, you will learn:

- How to define requirements
- How to evaluate requirements
- How to develop with requirements in mind
- What to consider when targeting a large and diverse audience

What is a requirement?

Requirements correspond to user needs, which can be concrete features but also criteria related to performance, security, accessibility, and other categories that are not usually considered as "correctness" but still needed in real-world software. For instance, one of the requirements for a mobile phone is the ability to make phone calls. Other requirements for the phone may include a level of quality for the calls, which imposes constraints on what kinds of microphones and loudspeakers can be used, as well as a level of privacy for the calls, which imposes constraints on how the call data is transmitted and logged.

Requirements are usually not objective; they are rarely formal specifications, because most humans do not typically think in formal terms. A user may ask for a "good quality" microphone on their phone even if they do not know exactly what quality they'd like or how microphones and voice transmission works. And "good sound quality" may mean one thing to a tennis player and another thing to a music composer. To the user, what matters is the interface to the external world, just like a module interface, not how the requirement is implemented. A key part of software engineering is the translation of customer needs into more formal specifications and code, with more objective criteria.

What customers need is not always what customers want. There are plenty of reasons why a customer might think they need something that they actually do not. For instance, customers may ask that a new piece of software should have a similar interface to an old piece of software, believing that this interface will allow them to best do their job, even if from an outside point of view there is clear room for improvement. This does not mean that the software engineer always knows best, but that what customers say is not always the ultimate truth.

Requirements may not be explicitly stated or even known. A classic example of this is a quote attributed to Henry Ford, the inventor of mass-produced automobiles: "if I had asked people what they wanted, they would have said faster horses". When the car was not a common and accessible means of transportation, customers could not know they would like to have one. However, once cars were mass-produced, they were a success. Modern cars have many improvements over old ones, in part derived from customer feedback. The same logic holds in software projects: customers will understand better what they want once they see a prototype of the software, and will provide better feedback once they have gained a deeper understanding of what they need by using the software. It is not realistic to imagine that the very first release will immediately satisfy all requirements.

A classic example of a software system that fulfilled requirements users did not know they had is the iPhone. Previous similar devices were either keyboard-based, such as Nokia or Blackberry phones, or stylus-based, such as the Palm PDAs. Even without third-party apps, the iPhone was already a revolution in phones, which led to the modern smartphone. Features such as advanced front-facing cameras are a direct reaction to how users actually ended up using smartphones, such as for making video calls.

A less successful software example is Windows 8, which released with an all-new user interface and a new model for applications, focused on consuming content rather than creating it. Windows users were used to the taskbar-and-window-based interface and found the new interface confusing and limited, which led to Microsoft mostly removing it from the next version of Windows. However, the interface may simply have been ahead of its time and on the wrong kind of hardware; Apple introduced multitasking to their iPad tablet with a very similar interface to Windows 8's.

How do we define requirements?

The first step towards defining requirements is to talk to users. The common stereotype of the "lone wolf" developer who does everything from their office, never discussing anything with anyone else, and who emerges once in a few months with a new release, is not a good way to develop software. It can work in the sense that the resulting software might be useful to some people, but it is unlikely to produce high-quality software that meets users' expectations.

Gathering requirements is often referred to as *requirements elicitation*, which makes it clear that requirements are not merely collected like fruits from a tree but obtained by discussing with users. There is no objective criteria that allows you to stop eliciting requirements; you have to decide when you think you have covered all of the important subjects, in a context in which you do not exactly know what is important!

Market research is a special kind of requirements elicitation, with different processes and results. Eliciting requirements is typically done with a known set of users, for instance by having engineers talk to the employees of the company that contracted the engineers' company. Market research is typically done with the goal of identifying something that will make people want to buy a product, or an improved version of an existing one.

Personas

To aggregate requirements from various users, one can create *personas*: fictional users that represent an "average" user of a certain kind. They do not correspond exactly to actual people, but average the characteristics of their category. For instance, in the context of a phone:

- Alice is a manager who uses her phone all the time between meetings, mainly for emails and calendar, and who also uses applications developed by her company such as for reporting her time.
- Bob is an office worker who rarely uses his phone for work, but frequently browses the Web and listens to podcasts on his commute to and from work.
- Carol is a student who uses her phone to chat with her friends, listen to music and watch videos
- Daniel is a retiree who only has a phone because he wanted to use the same chat apps as his family
- Eve is a huge technology fan and wants the latest phone with the best features; she uses her phone to control her smart home appliances and installs exotic apps such as network scanners.

The existence of personas does not imply the software can only do what the personas want, but that these are the main categories of users. For instance, given the above personas, an engineering team might attempt to develop apps that would make Daniel use his phone more. When doing market research, not all personas have to be served; a team might decide that a phone satisfying Eve is not a priority because she represents a small fraction of users that have complex needs.

User stories

A common structured way to write down individual requirements given by users is *user stories*: a single sentence that describes who the user is, what they want to do, and why they want to do it. For instance, "As a SwEng student, I want to learn about software requirements, so that I can engineer better software". This "As a [role], I want to [action], so that [reason]" is a common phrasing for user stories.

All three parts are important: the role indicates who the user is and what you can assume about their background, the action is a user-centric view of the feature, and the reason provides context. If you cannot find a role or reason for a user story, or if you have to use very generic ones such as "As a user", the feature may not be something users actually care about.

Structured definitions

To define requirements more formally, even if they cannot be fully formal, you can use structured definitions. For instance, in a framework like Cucumber, you write definitions in ordinary language, such as "Given [some context], when [something happens], then [some reaction should happen]". These definitions can be used nearly as-is in unit tests, which helps link code to user needs.

Structured definitions can then be grouped based on the *feature* they correspond to, into sets of *scenarios*. Consider the following:

```
Feature: Learn about requirements
      Scenario: Read notes
        Given the professor has posted the notes and the student has not read them yet
        When the student has free time
        Then the student reads the notes
      Scenario: Do exercises
        Given the student has read the notes
        When the student has free time
        Then the student does the exercises
```

We will discuss this more later, in the context of behavior-driven development.

How do we evaluate requirements?

When eliciting requirements, you will inevitably end up with a list that is far longer than anything you could produce in reasonable time for a reasonable budget. You will thus have to sort and filter requirements.

The step of verifying that what you are doing is actually useful to users is called *validation*. While users may not always know what they want, they can often tell you what they like and also what they do not want. For instance, if you discussed how to display medical records with some doctors who you are building an application for, then built a prototype user interface, the doctors might tell you that the way you are displaying X-rays is good, but that you should not give so much screen space to patients' personal details, because they rarely need them.

Validation can also be done "in the wild" with an already-deployed application. For instance, you could randomly show a new feature to only 10% of the existing users, and if the feature catches on with these users, then show it to another 50%, and finally to the entire 100%. If it turns out nobody is using the feature, perhaps you need to change the interface to make it more prominent, or perhaps it is simply not needed. You could also show the same feature with one interface for 50% of users and another interface for the other 50%, then see which group appears to use the feature more. This also works for paid products, such as testing what kind of "please buy the professional version of this software" message results in more buys. This kind of validation is called *A/B testing*.

Validation does not have to yield a binary yes/no answer: it can also help you prioritize which features to develop first. For instance, if only a small fraction of users agree that a feature is useful, but you believe those users really do need that feature, you can put it on your "backlog" of features to add, but place it below some more important features that affect more people or have a greater impact on their target audience.

Validation and prioritization do not only take users into account but also constraints such as technical feasibility and legality. For instance, a feature that is only mildly useful might be worth doing if it is trivial to implement, whereas a particularly useful feature that would require a full-time research team for several months might not be worth doing. Some features are required depending on where you are and what you are doing, such as providing specific privacy guarantees in the EU, or making government websites multi-lingual in Switzerland.

The list of features is not stable: as you release software and get feedback, you will add new features to the list of possible features to implement, and re-prioritize existing ones. This inevitably means that some features will always be at the bottom of the list, because there is always something else more important to do: this is a normal part of software engineering.

A common way of prioritizing features, bug fixes, etc. is to assign priority P0 (highest), P1 (medium), or P2 (low). There is an unwritten rule that P0s get done, P1s might get done if there is time, and P2s never get done. Another taxonomy is must-have, nice-to-have. Another way to approach prioritization is the so-called MoSCoW method, whose name stands for the four categories of requirements: must-have, should-have, could-have, and won't-have. A must-have is a non-negotiable product need (e.g., multilingualism for a Swiss government website), while a should-have adds significant value to the product, but is not vital. A could-have (often called also a nice-to-have) would be nice to have but would not affect the product much if left out. Since development teams are almost always pressed for time, the nice-to-haves typically turn into won't-haves after a few months.

In general, validating and prioritizing features is about finding how much *value* they have to users. Defining value is often tricky. A software company may look at requirements as the definition of a product that will sell to many customers for a high price, and quantify the value by how much the customers are willing to pay, e.g., a product that costs 5 CHF/user vs. 25 CHF/user. A paid-per-project type of IT consulting company looks at requirements more as a way to make the one customer of a project happy. Even a single contributor releasing an open-source project needs to think about who gains what from that software, since even free software needs happy users to justify its existence.

How to produce software that satisfies requirements?

There are many methodologies and tools that help software engineers align requirements and code. We present here three approaches: designing your software with requirements in mind, translating user requirements into tasks, and writing tests with requirements in mind.

Domain-driven design

Domain-driven design, or "DDD" for short, is a term invented by Eric Evans in the book of the same name. The goal of DDD is to base software on a model of the users' domain, and to reason about software in terms of that model. This allows users and developers to have a shared understanding of the software because they both use the same vocabulary, with users providing domain knowledge and developers providing information about what is technically feasible or not.

The key idea of DDD is to discuss code interfaces using a few concepts rather than the full expressive power of programming languages:

- An *entity* is an object whose identity depends on a specific key, such as a person being identified by a passport ID or social security number
- A *value object* is an object whose identity depends on its components, such as two passports being the same if the photo page and the visas are the same
- An *aggregate* is a set of objects that form a single logical item and have a logical "root" through which the item is accessed. For instance, a car contains many components.

Importantly, what is an entity, a value object, or an aggregate is entirely dependent on the context. In the context of a restaurant, customers might be value objects because the identity of individual customers is not relevant. In the context of a factory that produces passports, two passports with the same contents might be considered different because they are part of different orders. In the context of an auto repair shop, cars are not an aggregate because mechanics may directly operate on their components.

This small set of DDD terms can be taught to customers without them having to understand programming languages or advanced object-oriented concepts. Thus, you could ask a customer "are cars an aggregate" and they might answer "no, because we operate on their parts when we do <some task>". Compare this to asking a customer "should the Car class have a field for its ID, should its equality entirely depend on all of its fields instead, or should it be the root of an object graph?".

In general, even without customers, thinking about what your code should do using vocabulary from its context will help you design the code better than if you only use programming concepts, because a piece of software is a blend of programming and domain knowledge.

From user stories to tasks

After collecting user stories, validating them, and prioritizing them, a software engineering team must write code corresponding to these stories. User stories can correspond to one or more tasks depending on how complex they are.

The art of translating user stories to developer tasks requires time to master, so you will have to practice it. Let's take an example story: "as an administrator, I can log into the system, so that I can use the admin interface". This user story tells us nothing about how it will be implemented, or how much time it will take. You would need to decide how to implement this story. You can choose to use an authentication service (e.g., Google, Facebook, EPFL). Then you need to show something to the user: a login screen. And the last part is that you want to keep some settings associated with each user's account in your database. These steps of achieving the login functionality are not the only way to implement what is necessary for the user story to be considered done but they are one possible way. Whatever the steps for delivering the functionality described in a user story, you need to describe these steps as tasks, estimate how much time each task is going to take, assign a team member to work on each, and do it.

Behavior-driven development

Behavior-driven development, or "BDD" for short, is a software development process that specializes test-driven development by writing requirements in a domain-specific language that can be run as tests. Developers write requirements in the form "Given… When… Then…", using frameworks such as JBehave. Earlier in the lecture we mentioned Cucumber as another example.

The goal of BDD is to introduce an intermediate language between free-flowing user requirements and code, that can be understood by both users and developers. Developers then have to translate this language into code, and that code is more likely to match users' expectations because developers need to do less subjective interpretation.

Even without using a tool, you can practice BDD in your own code by writing down requirements in the given/when/then format, and then writing tests that correspond to these scenarios. These tests may be unit tests for low-level scenarios, but most of them will likely be end-to-end tests that correspond to a specific user expectation, such as what happens when an user clicks on a button in a given context.

"Given/When/Then" is similar to the widely-used Arrange/Act/Assert (AAA) pattern in unit tests; you can "evolve" tests using the AAA pattern by expanding on what context the code arranges, what happens in the action, and what is being asserted.

Requirements when targeting diverse audiences

One key requirement when your software will be used by people from different cultures is *internationalization*, sometimes abbreviated as "i18n" with the "18" standing for the number of letters it replaces. Internationalization is not only about localization (sometimes abbreviated as "l10n"), in which one translates software into different languages, but also about cultural differences.

For instance, a Swiss person will view the check mark ✔ as a symbol for "correct", but a Japanese person would expect an O-mark ⭕ instead. An American will recognize a "head nodding" sign to mean "yes", but in Bulgaria this can mean "no" instead. Software that hardcodes the position of user interface elements will not be properly displayed in languages that read in another direction, even if the translation is correct, such as English script going left-to-right vs. Arabic script going right-to-left. Translators may not be able to do a correct job if the text they are given is split in ways that make assumptions about languages; for instance, having one version ("singular") of a string for 1 item and one version ("plural") for more than 1 items is standard in languages like English but will not work in Russian, in which there can be different forms for 1, 2 to 4, and more than 5 items.

The examples above are all user interface issues, but internationalization goes deeper than that and affects data models, i.e., how the software represents objects. For instance, what parts are there in a name? Some countries have two, named "first" and "last"; some countries allow for multiple "first" names, potentially with a "preferred" one; some countries have a "middle name"; some countries have "patronymics" or "matronymics" derived from ancestor names either in addition to or as a replacement for "last" names; some countries do not have the concept of "first" or "last" names, with the entire name being just a name; sometimes names are just one single word. In general, programmers believe many falsehoods about names, time, geography, emails, and so on. This can lead to major problems for the users of the software written by these programmers, such as those with particularly short or long names, or with more or fewer components to their names than a piece of software was designed for.

Even "facts" within a single culture can be unexpected. For instance, how many days are there in a year? In 1752, there were only 355. How many seconds are there in a minute? There are usually 60, but there can be 61 with a leap second added, or even 59 with a negative leap second, though this has not happened yet. Is x + 1 > x always true? In mathematics yes, but in limited-precision integers such as C or Java `ints`, it can be false due to overflows.

The important take-away is not that you should try to anticipate everything[1] but rather that you should try to be explicit about your assumptions and then double-check them. Ask questions instead of assuming that the world works as you believe it does everywhere.

Another important requirement for software that has more than a few users is accessibility: how usable is the software by everyone, regardless of any impairment? For instance, can it be used by people with vision or hearing impairments? It is unfortunately common to think of accessibility as an annoyance that has to be done to comply with laws and other regulations, but this is not a good way to look at it. Accessibility is not only about people who strictly need it all of the time, but also about people who only partially need it, who need it situationally, or who just find it convenient. For instance, closed captions are particularly useful for the hard of hearing, but are also useful to anyone in a situation where there is too much ambient noise to hear what is being said (e.g., crowded public transportation), or to parents who want to teach their child to read.

There are standards to make software accessible and to define what accessibility means. For instance, ARIA helps developers make their dynamic web pages accessible. The EN 301 549 regulation was created by the European Union as a standard for accessibility that all public European content must meet.

One interesting application of accessibility is testing, which you can think of as accessibility to machines. If your user interface is machine-readable, testing it is far easier.

Google has good documentation on how to check the accessibility of Android apps.

[1] In general, there are far too many cultural differences to even think about, especially because many of them go unnoticed until a problem occurs. For example, did you know that, in 1936, Liechtenstein and Haiti met at the Olympic Games and realized that they had been using the same flag without knowing it?