Lecture Notes: Requirements

17-313: Foundations of Software Engineering
Claire Le Goues and Christian Kaestner

Requirements engineering is the process of capturing in symbolic representation a worldly computational problem so that the statements obtained by following rules of symbolic manipulation are useful statements once translated back into the language of the world [Scherlis 1989].

In plainer English: requirements engineering seeks to (1) figure out what system should be built, and then (2) express those ideas such that the correct thing can be verifiably built. The first is a question of knowledge *acquisition*, addressing how to capture all relevant detail about a proposed system and then determine whether that knowledge is complete and consistent. The second is a question of knowledge *representation*: Once captured, how do we express it most effectively, such that it is received consistently by all the different people who will interact with it?

(We also need to know how to check requirements for correctness, consistency, and implementability; how to check that they've been represented faithfully; and how to maintain requirements as they change and throughout the implementation process. These notes focus on preliminary definitions, and we focus in this course primarily on elicitation and representation, with some coverage of consistency/correctness checking.)

A major challenge in drilling down into further detail on this issue is that the word "requirements" is used inconsistently throughout the software industry. It can mean everything from the vague idea your cofounder has about an awesome new app to a detailed and lengthy Software Requirements Specification (SRS) used to guide bidding and contracting for a large IT acquisition. In this course, we will drill down into formal requirements engineering to some degree of detail, because in so doing we will better understand the challenges of requirements and the techniques and terminology that can be used to overcome some of those challenges. We will stop short of writing a full SRS document.

Requirements play a major role in sinking software projects. Building the wrong thing, failing to build the right thing, or building a thing that would be the right thing under different circumstances, are all common contributors to real-world project failures.

Requirement types and their expression

We divide requirements into three broad categories: (1) functional requirements, (2) quality requirements,[1] and (3) domain assumptions.[2]

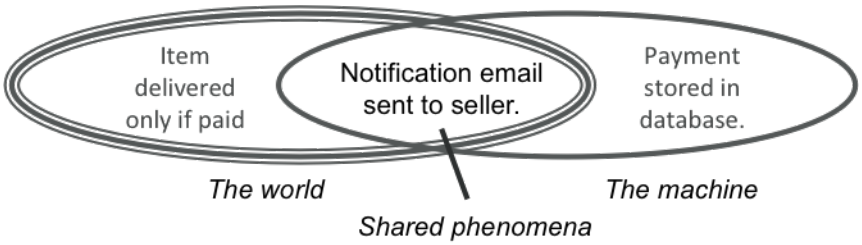

Figure 1: The problem world, the machine solution, and the interface between them (adapted from [van Lamsweerde 2009])

Functional requirements and implementation bias

The functional requirements describe what the machine should do. They cover the inputs and expected outputs of the system, its interface with the world, and how it will respond to events.

A common one-sentence summary of requirements is: "Requirements describe what the software should do, and not how it should do it." A key goal in requirements engineering is to avoid *implementation bias*: describing the desired system in terms of its implementation. This constrains the software engineers' choices and may prevent them from exploring alternative (but possibly better) solutions.

For these reasons, functional requirements should be expressed in terms of the relationship between the system to be built (the *machine*) and the *environment* in which it will operate. The goal of software engineering can thus be broadly reformulated as seeking to improve the world by putting some machine into it that solves the problem at hand. Requirements then describe what is observable at the environment-machine interface, and should be expressed as statements about the effect the machine has on the environment (Figure 1).

Taking this approach, requirements are largely expressed using two types of grammatical statements:

- Statements written in the *indicative mood* describes the environment (as-is, independent of the system). Example: "Every product has a unique product code." We will discuss these statements in the context of the domain knowledge and assumptions, below.
- Statements written in the *optative mood* describes the environment with the system to be built. Example: "The system shall email clients about their shipping status."

This approach to describing functional requirements avoids implementation bias because the requirements do not actually make statements about the system being built. Instead, they describe the effect the system has on the world.[3]

Beyond these types of statements, presented in an organized prose form, we will primarily focus on use cases to express functional requirements.

[1]The literature often uses the term "non-functional requirements", the phrase is misleading; using either term on exams or homework is fine.

[2]Some writers also contrast between *user* (human-ese) and *system* (computer-ese) requirements, but this distinction is less important for our purposes.

[3]Of course, even following these principles, requirements can still be flawed, e.g., if they make invalid statements about the world, if they are vague, etc, but they cannot overconstrain the implementation.

Quality requirements

The quality requirements specify the quality with which a system should deliver its functionality (examples include but are not limited to performance, throughput, reliability, availability, usability, etc). Rather than strictly specifying desired functionality, quality requirements serve as constraints on the implemented system and provide design criteria to select between alternative implementations.

Good requirements, of any kind, are verifiable. They serve as contracts, and thus should be testable or falsifiable. We therefore contrast *informal goals*, or general intentions (such as "ease of use", or even "the system should be easy to use by experienced controllers, and should be organized such that user errors are minimized.") from *verifiable non-functional requirements*, or statements that use measures that can be objectively tested (such as "Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day, on average.") Informal goals may still be informative for developers, as they convey the intentions of the system users, but verifiable quality requirements are key to effective requirements expression.

Domain knowledge

Assumptions or domain knowledge document existing properties of the environment. Such assumptions serve two general purposes: (1) constrain the *scope* of the system by describing the conditions under which it is intended to perform correctly, and (2) define the relationship/interface between the machine model of the world and the actual world.

The first goal is straightforward enough: by documenting assumptions about the environment as it is, we specify (and thus limit!) the conditions under which the system is expected to perform correctly. Additionally, recall that requirements are expressed in terms of the actions performed by the machine on the environment. Specifying a clear boundary between machine actions and environment actions clarifies which actions are actually the responsibility of the system-to-be. For example, consider a system like Netflix's that suggests movies or TV shows to users. The scope of that system would vary tremendously based on whether it is able to query an existing system (within its environment) that stores and manages user watch logs and information, or if, alternatively, it is responsible for storing and managing that information itself.

The second goal requires a bit more explanation. Domain assumptions ground the definitions used in requirements in reality (providing *designations*). This helps ensure that the functional requirements are correctly defined and implementable. Consider the following example, borrowed from Zave and Jackson [Zave and Jackson, 1997]:

> **Ash:** Two important basic types are *student* and *course*. There is also a binary relation enrolled. It can be expressed as: $\forall s \forall c(enrolled(s, c) \implies student(s) \wedge course(c))$
> **Brook:** Do only students enroll in courses? I don't think that's true.
> **Ash:** But that's what I mean by student!

If a student is simply a person enrolled in a course, Ash's assertion is vacuous, but easy to implement. If a student is a person who has matriculated at the university and not yet graduated or withdrawn, Ash's assertion strongly constrains who can take courses. The difference is important to the correct implementation of the system.

The problem here is that a student is just a concept that the machine will express and manipulate symbolically. A machine does not *know* what a student is. Without providing a designation

for the word "student", we cannot check that a specification derived from these requirements faithfully reflects the desires of the stakeholders.

More formally, the meaning of "being a student," is *unshared* knowledge: The machine cannot observe it. This is true of most real-world phenomena. Other actions are observable by both the world and the machine: This is *shared* knowledge. Figure 1 shows an example of the distinction: in a payment system, the "payment record created in database" is an action observable by the machine; both the machine and the world can see if the payment notification was sent to the seller; but the machine cannot possibly know whether the item is actually physically delivered, because this is a property of the physical world.

The machine cannot be put in charge of unshared actions or phenomena, and requiring them makes for requirements that are impossible to implement. Domain knowledge and assumptions therefore serve to provide definitions of unshared information in the requirements and relate them to shared information that the machine can actually observe and affect.

Eliciting requirements

Stakeholders and actors. We elicit requirements primarily by engaging *stakeholders*, consisting of any person or group who will be affected by the system, directly or indirectly. The first step in requirements elicitation is typically *stakeholder analysis*, which identifies relevant stakeholders. That is: who is the system for? Who will decide whether the system is acceptable? Missing or hidden stakeholders pose a key risk in requirements elicitation because the system must, by definition, satisfy stakeholder goals. If you miss someone, you risk not knowing of an important goal that the system must satisfy (and thus risk not implementing the appropriate functionality).

We distinguish between stakeholders and the *actors* or *agents* in a system. An actor is an entity that interacts with the system for the purpose of completing an event [Jacobson, 1992], and can be a user, organization, device, or even another system. The actors are not as broad as the stakeholders.

Interviews. Once stakeholders have been identified, they must be engaged to help draw out the requirements of the system. A common way to do this is via *interviews*. The goals of these interviews are to understand functional requirements, identify and learn domain-specific concepts, and prioritize quality attributes.

There are many potential challenges to effective interviews: Stakeholders may disagree; Unhandled or hidden conflicts are another key risk. Stakeholders do not always know what they really want nor how to articulate it, nor how much it will cost. They often have domain knowledge, use jargon, or leave out "obvious" requirements that aren?t obvious to a non-expert. Thus, effectively engaging stakeholders can be difficult regardless of their technical sophistication: A non-engineer may not know very much about the underlying technology, making it hard to draw out what they actually want. By contrast, a highly technical engineer may operate under hidden "obvious" assumptions.

Effective interviewing typically begins concretely, with specific questions, proposals, or by working through a mockup or prototype. Humans are typically much better at recognizing correct or incorrect solutions than they are at solving a problem from nothing. Stories, storyboards, scenarios, informal or hypothetical use cases, or examples can be helpful, or contrasting the proposed system with a current system (if applicable). Eliciting requirements at the boundaries of the system—corner, negative, or failure cases—can also help with scope and exceptional situations. We will see examples of these types of tools in class and in the slides.

Cited references

1. William L. Scherlis, responding to E W Dijkstra "On the Cruelty of Really Teaching Computing Science". Communications of the ACM 32:12, p407.
2. Pamela Zave and Michael Jackson. "Four dark corners of requirements engineering." ACM Trans. Softw. Eng. Methodol. 6, 1 (January 1997), 1-30.
3. Axel van Lamsweerde: *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley 2009.
4. Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar vergaard: *Object-oriented software engineering - a use case driven approach.* Addison-Wesley 1992, ISBN 978-0-201-54435-0.