Part A

Introduction

In this first part of the book we explore the foundations of Requirements Engineering, and introduce many of the key ideas. Chapter 1 explains what we mean by requirements engineering, and defines software-intensive systems as a suitable scope for application of RE. Chapter 2 explores the idea of a requirement, and introduces some key distinctions for understanding requirements. For example, a key distinction is between describing a problem to be solved and describing a particular design for solving it. We also distinguish between two very different worlds: the world of human activity in which problems that need solving can be found, and the world of computer technology in which programs are written and tested to run on particular hardware configurations. A requirements analyst must understand both of these worlds, and seek ways to bridge between them. This explains why requirements analysts concern themselves with people, and with the organizational and social contexts that surround them, as well as the technical aspects of designing and implementing computer systems.

Chapter 3 examines the context in which requirements engineering takes place – as part of a larger engineering process. We will define what we mean by engineering, and explore the role of RE in an engineering project lifecycle. Because of the huge variety of ways in which computers are used, the processes of requirements engineering vary tremendously from one domain to another. Developing software for an aircraft flight control system is a very different task from developing the software for a new web browser. We will explore the commonalities and the differences by considering the nature of different types of engineering project.

Finally, requirements engineering is concerned with *complex* problems, and so we need a way of dealing with complexity. Chapter 4 explores how to understand complex problems by identifying and describing *systems*: fragments of the world that make sense when treated as a coherent set of activities. As will become clear, many of the systems we need to study are *human activity systems.* To understand such systems we will need to draw on ideas from a variety of disciplines, ranging from social and cognitive sciences through to logic, mathematics and the engineering sciences.

- Chapter 1: What is Requirements Engineering?
- Chapter 2: What are Requirements?
- Chapter 3: What is Engineering?
- Chapter 4: What is a System?

CHAPTER 1

What is Requirements Engineering?

The field of Requirements Engineering (RE) is relatively new, so it seems appropriate to begin by asking some very basic questions: What is RE all about? When is it needed? What kinds of activities are involved in doing RE? Our answers to these questions provide both a motivation and a scope for the techniques introduced in the remainder of the book. We will begin with the idea of a *software-intensive system,* by which we mean an inter-related set of human activities, supported by computer technology. The requirements express the *purpose* of such a system. They allow us to say something meaningful about how good a particular system is, by exposing how well it *suits its purpose*. Or, more usefully, they allow us to *predict* how well it *will* suit its purpose, if we design it in a particular way. The idea of human-centered design is crucial – the real goal of an engineering process is to improve human activities in some way, rather than to build some technological artifact.

Requirements engineering applies to the development of *all software-intensive systems*, but not necessarily to the development of all *software,* as we shall see. There are a huge range of different kinds of software-intensive system, and the practice of RE varies across this range. Our aim throughout this book is to explore both what is common and what varies across these different types of system. We will set the scene in this chapter by offering some examples of different types of system development, and indicate the role of RE in each. Finally, we will end the chapter with a quick tour of the kinds of activity that comprise RE, as a preview of the rest of the book.

By the end of the chapter you should be able to:

- Define the term software-intensive system.
- Summarize the key techniques used in requirements engineering for dealing with complexity.
- Explain what is meant by a “wicked problem”, and give examples of wicked problems.
- Use the definition of quality as “fitness for purpose” to explain why software quality cannot be measured unless the requirements are properly understood.
- Give examples of different types of engineering project to which requirements engineering applies.
- Suggest some types of software for which requirements engineering is unnecessary.
- Explain the risks of an inadequate exploration of the requirements.
- Account for the reasons that requirements change over time.
- Distinguish between the hard and soft systems perspectives.
- Judge whether a human-centered perspective is appropriate for a particular project.
- Describe the typical activities of the requirements analyst.

1.1. Basic Concepts

Software-intensive systems have penetrated nearly all aspects of our lives, in a huge variety of ways. Information technology has become so powerful and so adaptable, that the opportunities for new uses seem boundless. However, our experience of actual computer systems, once they have

been developed, is often disappointing. They fail to work in the way we expect, they are unreliable, and sometimes dangerous, and they may create more problems than they solve. Why should this be?

Computer systems are *designed*, and anything that is designed has an intended *purpose*. If a computer system is unsatisfactory, it is because the system was designed without an adequate understanding of its purpose, or because we are using it for a purpose different from the intended one. Both problems can be mitigated by careful analysis of purpose throughout a system's life. Requirements Engineering provides a framework for understanding the purpose of a system and the contexts in which it will be used. Or, put another way, requirements engineering bridges the gap between an initial vague recognition that there is some *problem* to which we can apply computer technology, and the task of building a system to address the problem.

In seeking to describe the purpose of a computer system, we need to look beyond the system itself, and into the human activities that it will support. For example, the purpose of a banking system is not to be found in the technology used to build such systems, but in the business activities of banks and day-to-day needs of their customers. The purpose of an online flight reservation system is not to be found in the details of the web technology used to implement it, but in the desire by passengers for more convenient ways of booking their travel, and the desire of airlines to provide competitive and profitable services.

Such human activities may be *complex*, because they generally involve many different types of people, with conflicting interests. In such situations it can be hard to decide exactly which problem should be tackled, and it can be hard to reach agreement among the stakeholders. Requirements engineering techniques offer ways of dealing with complexity, by systematically breaking down complex problems into simpler ones, so that we can understand them better.

For some types of software, we may already have an excellent understanding of the intended purpose, even before we start the project. For other types of software, the problem to be tackled may be simple to describe, even if the solution is not. In both these cases, requirements engineering techniques may not be needed. However, some problems look simple until we start to analyze them, and some people develop software systems thinking that they understand the purpose, but find out (eventually!) that they were wrong. So we need to first consider what type of projects need requirements engineering. We will begin with the idea of a *software-intensive system*, consider the importance of *fitness-for-purpose,* and take a closer look at *complexity of purpose*. This will lead us to a definition of requirements engineering.

1.1.1. Software-Intensive Systems

To understand the scope of requirements engineering, we will consider the idea of a *software-intensive system*. By this we mean a lot more than just software – software on its own is useless. Software can only do things when it is run on a computer platform, using various devices to interact with the physical world. Sometimes this platform will be taken for granted. For example, some software is designed to run on a 'standard' PC platform, meaning a particular combination of workstation, operating system, keyboard, mouse, printer and network connection. For other types of software, we may need to design specialist hardware or special configurations of hardware devices along with the software. Either way, the hardware platform and the software together form a system, and we can refer to these software+hardware combinations as "computer systems".

But we don't mean just computer systems either – computer systems on their own are also useless. Computer systems can only do useful things when they are placed in some human context where they can support some human activity. This human context provides a purpose for the computer system. Sometimes this human context will be taken for granted – when computer systems are designed to be used in 'standard' contexts for 'standard' types of task. For example, web servers, email clients, word processors, and even aircraft autopilots are designed for relatively well-understood user groups, engaged in fairly routine activities. Nevertheless, any new computer

system will lead to some changes in the human activities that it supports. Effectively, this means the human activity system must also be (re-)designed along with the computer system. For example, if you design a new air traffic control system, you must take care to design an appropriate set of roles and coordination mechanisms for the people whose work it will support. If you design a new web browser, you should consider the ways in which it will change how people access the web. This is because the computer and the human activities together form a system. This is what we call a *software-intensive system*.

The term 'software-intensive system' describes systems that incorporate hardware, software *and* human activities. But we chose a term that emphasizes the word 'software' because it is *software* that marks such systems as radically different from any other type of engineered system. Some of the principles we describe in this book apply to any engineering activity, whether computers and software are involved or not. But we concern ourselves with software-intensive systems because software presents so many special challenges. We will examine these challenges in more detail in chapter 3, when we look at engineering processes. In brief, the challenges arise because software is both complex and adaptable. It can be rapidly changed on-the-fly without having to manufacture replacement components. Software now makes computer systems so compellingly useful that it is hard to find any aspects of human activity that have not been transformed in important ways by software technology. And the design of software is frequently inseparable from the task of designing the human activities supported by that software.

An important observation is that the introduction of new computer technology into any human activities inevitably changes those activities, and people inevitably find ways of adapting how they use and exploit such technology. Thus, a cycle of continuous change is inevitable. We will return to the theme of continuous change throughout the book.

1.1.2. Fitness for Purpose

We can say a system is badly designed if it is not well suited to the purpose for which it was intended. If the system is supposed to make some job more efficient, and it does not, then it is a poor system. If it is supposed to make a risky activity (like flying) safer, but does not, then it is a poor system. Turning this around, we define *quality* in terms of *fitness-for-purpose*. This means that we cannot assess quality as a measure of software by itself; we can only assess quality when we consider the software in the context of a set of human activities. In other words, software *on its own* cannot be said to be of 'high quality' or of 'low quality', because quality is an attribute of the *relationship* between software and the purposes for which it is used. This means that requirements engineering plays a critical role in our understanding and assessment of software quality.

The purpose for a particular system, and hence the key quality measures, may appear to be self-evident. For example, it is hard to imagine a purpose for an aircraft flight control system that does not include flying from one place to another as safely as possible. But human activities are complex, and so it is rare to find a system that has a single, unchanging purpose. Most systems have multiple purposes, and those purposes change over time. The different purposes of a system may be associated with different stakeholders, or with the different roles that the stakeholders play.

A systematic investigation of the purpose for any proposed software-intensive system is therefore essential. If we do not fully understand the purpose of a computer system, we cannot assess its quality. Further, if we do not understand the intended purpose of a system that we are trying to design, then we can only ever *achieve* good quality by accident.

1.1.3. Complexity of Purpose

Because software-intensive systems involve a high degree of interaction between people, software and hardware, they are intrinsically complex. To get a sense of this complexity, consider first the *physical* interaction between people and computers. Devices that contain little or no software tend to have relatively simple interfaces – a few buttons to push, handles to hold, warning lights to flash, etc. By contrast, a typical computer has close to a hundred buttons/keys to push, together with hundreds of menu items to select, commands to type, and icons to click. It has a screen with thousands of pixels that can change appearance completely from moment to moment.

Consider also the *duration* of interaction. There are very few devices that we use intensively for hours on end, and those few that we do tend to have simple, unchanging modes of interaction. We might drive a car for hours, but for most of that time we will use no more than a handful of actions (turn the wheel, press the brake, activate the turn signals, etc). If we use a computer for hours, we are likely to carry out hundreds of different actions during the interaction.

Now consider the question of *control* in this interaction. Human-computer interaction is *mixed-initiative* – sometimes we tell the computer to do things, but just as often it tells us to do things. When a person carries out some task with the help of a computer, the *responsibility* for various steps of the task frequently passes back and forth between the computer and the person. The structure of the task and the choice of what to do next are sometimes under the control of the person, and sometimes under the control of the computer. No other designed artifact comes anywhere close to this degree of interactivity with us.

Finally, consider the *situatedness* of this interaction. The activities that computers support are nearly always the activities of a group of people. Sometimes the computer system mediates the interaction and coordination of these people (for example, when we use email, shared files, wikis and chats, etc), while at other times it is on the periphery, assisting or recording the work of individual members of the group. Whatever roles the computer plays, we cannot consider the interaction of *one* person with *one* computer without also examining how that activity fits into, and helps to shape, the wider set of social activities of which it is a part.

For all these reasons, the interaction between people and software is more complex than our interactions with any other kind of designed artifact. The actions of the software are woven so closely with human activities that each shapes the other in ways that are hard to predict. In other words, the activities of software and people are *closely coupled*. This close coupling is a major source of complexity for software-intensive systems. Because of its complex interactions with the world, the *purpose* of the software is complex. And because computers are so useful, we demand ever more functionality from them, and so complexity of purpose increases as software technology develops.

Because of this complexity of purpose, the design of software-intensive systems belong to a class of problems known as *wicked* problems. The term was coined by Rittel and Webber for problems that have the following characteristics:

- There is no definitive formulation of the problem – for example because different stakeholders each have their own conception of what the problem is.
- There is no stopping rule – each solution is likely to lead to new insights into the problem, and the problem is never likely to be solved entirely.
- Solutions are not right or wrong, but merely better or worse.
- There is no objective test of how good a particular solution is – such a test involves the subjective judgment of various stakeholders.
- There is no pre-existing set of potential solutions, nor is there a well-described set of features of such solutions. These must be discovered during problem analysis.
- Every wicked problem is essentially unique – each problem is sufficiently complex that no other problem is exactly like it.

- Every wicked problem can be considered to be a symptom of another problem, which means it is hard to isolate the problem and hard to choose an appropriate level of abstraction to describe the problem.
- The designer has no 'right' to be wrong – because wicked problems often have strong political, ethical or professional dimensions, stakeholders tend to be intolerant of any perceived mistakes by the designer.

The over-riding feature of wicked problems is that arriving at an agreed statement of the problem *is* the problem. Wicked problems are not just ill-defined; rather they are the kinds of problems that tend to defy description.

1.1.4. Dealing with complexity

Requirements Engineering offers a number of techniques for dealing with complexity of purpose, which are built into the various techniques described in this book. Of these, three general principles are so useful that we will briefly introduce them here: *abstraction, decomposition* and *projection*:

- Abstraction involves ignoring the details so that we can see the big picture. When we take some set of human-computer activities and describe them as a system, we are using an abstraction. When we take two different actions and describe them as instances of the same general activity, we are using an abstraction.
- Decomposition involves breaking a set of phenomena into parts, so that we can study them independently. Such decompositions are never perfect, because of the coupling between the parts, but a good decomposition still offers us insights into how things work.
- Projection involves adopting a particular view or perspective, and describing only the aspects that are relevant to that perspective. Unlike decomposition, the perspectives are not intended to be independent in any way.

These ideas are so useful that we use them all the time, often without realizing it. Requirements analysts use them in a particular way to understand problem situations, and to identify parts of a problem that can be solved using software. Systematic use of decomposition, abstraction and projection allows us to deal with complexity by making problems simpler, and mapping them on to existing solution components. For example, we may look for decompositions in which some of the parts are familiar. In the ideal case, this leads to sub-problems that are sufficiently well-known that they have standard solutions. However, we may still have significant work to do in adapting these known solutions to the new problem context.

Not all *software* has the complexity of purpose that we have described. Some pieces of software do not have a close coupling with human activities. If the interaction between a piece of the software and the remainder of the system is sufficiently simple, we can standardize the *interface* between them. A good decomposition may result in a component having a sufficiently simple purpose that we can proceed to design the component without worrying about the larger system(s) in which it will be used. The software component itself might still be very complex. The systems in which the component is used may be very complex. But the *purpose* of the component can still be simple, resulting in a simple and stable interface between the component and the rest of the world.

Sometimes, such components are sufficiently useful that they are needed in many different systems, and eventually their purpose becomes standardized. The requirements engineering is done when the component's purpose is identified, and its interface is standardized. The interface describes the basic function of the component, usually with one or two basic properties that matter for such components, such as performance or accuracy. New variants of these components can then be designed without re-analyzing their purpose, as long as that purpose doesn't change. Examples include compilers, many of the individual components of modern user interfaces, operating systems, and networking software. Because the purposes of such components are so well

understood, we do not need to re-analyze the requirements each time we design a new version. In such cases, we are not designing software-intensive systems, but rather, well-defined *software devices*. We will return to the distinction between systems and devices in chapter 3.

The idea of re-usable components brings us to a fourth principle for dealing with complexity: *modularity*. While decomposition helps us to break a large problem into more manageable pieces, modularity is concerned with finding structures that are stable over time and across different contexts. Modularity is important in design, because it produces designs that are robust, especially when changes can be localized in particular modules. However, modularity is just as important in requirements, because it allows us to exploit existing solutions when considering any new problem, and it allows us to handle evolution of the requirements over time.

1.1.5. Defining Requirements Engineering

We are now ready to consider a definition of Requirements Engineering:

Requirements Engineering (RE) is a set of activities concerned with identifying and communicating the purpose of a software-intensive system, and the contexts in which it will be used. Hence, RE acts as the bridge between the real-world needs of users, customers, and other constituencies affected by a software system, and the capabilities and opportunities afforded by software-intensive technologies.

The name "requirements engineering" may seem a little awkward. Both words carry some unfortunate connotations:

- 'Requirements' suggests that there is someone out there doing the 'requiring' – a specific customer who knows what she wants. In some projects, requirements are understood to be the list of features (or functions, properties, constraints, etc.) demanded by the customer. In practice, there is rarely a single customer, but rather a diverse set of people who will be affected in one way or another by the system. These people may have varied and conflicting goals. Their goals may not be explicit, or may be hard to articulate. They may not know what they want or what is possible. Under these circumstances, asking them what they 'require' is not likely to be fruitful.
- 'Engineering' suggests that RE is an engineering discipline in its own right, whereas it is really a fragment of a larger process of engineering software-intensive systems. The term 'engineering' also suggests that the outputs of an RE process need to be carefully engineered, where those 'outputs' are usually understood to be detailed specifications. It is true that in some projects, a great deal of care is warranted when writing specifications, especially if misunderstandings could lead to safety or security problems. However, in other projects it may be reasonable not to write detailed specifications at all. In many RE processes, it is the understanding that is gained from applying systematic analysis techniques that is important, rather than the documented specifications.

Despite these observations, we will use the term 'requirements engineering' to describe the subject matter of this book, because the term is now well established, and because there are no better terms available. The term 'engineering' also has some useful connotations. Good engineering requires an understanding of the trade-offs involved in building a useful product, as perfection is rarely possible. These trade-offs also apply to the question of exactly which 'practical problems' one should attempt to solve – a decision that lies at the heart of requirements engineering.

One of the key themes of this book is that requirements engineering is an essential part of *any* attempt to develop *any* kind of *software-intensive system*. All such systems have a *purpose* – whether we call it 'the requirements' or 'users needs' or 'product features', or whatever. Furthermore, the purpose of a software-intensive system is *never* self-evident. This is so, whether or not the developers explicitly analyze 'the requirements' and write detailed specifications, or just