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 skip that part and get on and write program code. If the developers do not have an adequate understanding of the purpose they will find it hard to build a system that suits its purpose.

Bear in mind that we are talking about *software-intensive systems*, and not just *software*. There are some types of *software* whose purpose *is* self-evident, and for which requirements engineering may therefore be unnecessary. Such software either does not form part of a software-intensive system, or has become such a standard *component* that its purpose is completely understood *a priori*. This is the case if a complete description of the interface between the component and its environment is available in advance, and can be treated as fixed during development of the component. If such a description is not available, then a requirements engineering activity will be needed to create one.

1.2. Examples of Requirements Engineering in action

Requirements Engineering takes place in a huge variety of settings, for a huge variety of computer-based systems. The following examples should give some idea of this variety:

1. The Air Force wishes to engage a contractor to develop a control system for a new type of experimental aircraft. It conducts a requirements analysis, and draws up a detailed requirements specification. This specification is used as the basis of an invitation to tender. Potential contractors bid on the contract. The Air Force chooses a contractor, AlphaCo, on the basis that the price that AlphaCo bid was within the Air Force's expected budget, and the Air Force was impressed with AlphaCo's presentation, experience, and the expertise of its staff. A fixed price is agreed and the contract is signed. AlphaCo then implements the system according to the detailed specification, which also acts as a legal contract. The specification is regarded as fixed, although occasionally AlphaCo seeks to negotiate a change, usually when some part of the specification turns out to be infeasible to implement.
   Happy Ending: AlphaCo delivers the control system several weeks early; it is tested by the Air Force, used for several years in experimental aircraft, and eventually adopted as the basis for an entire fleet of fighter jets. AlphaCo makes a small loss on the contract (it cost more to develop than the agreed price), but makes a large profit on subsequent contracts with the Air Force over many years.
   Or perhaps: The specification turns out to be poorly written, and AlphaCo's engineers cannot understand it. They build what they think is requested and write test cases using the same assumptions. The software passes the tests, but all the early test flights crash, and in one incident, a pilot is killed. The Air Force cancels the project, and considers not paying AlphaCo, but is advised by its lawyers that it cannot legally abrogate the contract, because the specification appears to have been met. AlphaCo makes a profit anyway as their development costs were significantly lower than the agreed price.
2. A government agency responsible for registering drivers and motor vehicles wishes to centralize its record keeping process. It produces a rough statement of the scope of the system they envisage, and puts out a call for proposals. Contractors submit proposals that set out their experience in building such systems. The agency selects a company, BetaCo, which appears to have the most experience in developing similar systems, and agrees to pay BetaCo its costs, plus an agreed percentage. BetaCo then conducts a detailed requirements analysis in partnership with the agency. They meet regularly to ensure that they agree about the evolving set of requirements. Once the agency is happy with the detailed specification, BetaCo proceeds to develop the software. BetaCo developers continue to meet with the agency throughout the development to review any changes to the requirements specification, and to discuss how these would impact the cost and schedule.
   Happy Ending: The system delivered by BetaCo is a major success – it allows the agency to cut its record-keeping costs, dramatically reduces fraud, and cuts in half the time taken to issue new licences. Development costs are recouped within two years of delivery.
   Or perhaps: Throughout the development, the agency asks for additional functions to be added to the system. There is an election during this time, and the new government requests

that the system be linked with police records as part of a new policy for reducing crime. Six years into the project, BetaCo is still spending more time on handling requirements changes than on building the system. Two years later, another new government cancels the project and orders an inquiry. BetaCo has been paid $50 million over the life of the project, but has never delivered any working software. The inquiry concludes that inadequate requirements management and problems in the contracting process were major factors, but that BetaCo cannot be blamed for the fiasco.

3. A large manufacturing company, FabriCo, wishes to improve the efficiency of its head office. It invites an established IT consultancy company, GammaCo to come and investigate its work practices, to see what can be automated. Consultants from GammaCo spend several months on site at the head office, observing the existing work practices. They initiate a series of participatory design sessions with the office management and staff. Together, they develop some storyboards characterizing key procedures, and sketch out designs for various software tools that might help. Prototypes of these tools are developed and the staff try them out. This process iterates, as the consultants learn more about which suggestions work and which do not.
   Happy ending: The staff are delighted with the new system, productivity soars, and FabriCo uses this as the basis for a successful expansion of their operations.
   Or perhaps: The staff are initially enthusiastic about the early prototypes, and want to start using them immediately. However, GammaCo discovers a series of technical problems in scaling up the software from these prototypes. As a compromise, GammaCo agrees to deliver an initial version with just a few of the basic functions. When the system is delivered, the user interfaces are slightly different from the early prototypes, but in ways that make them awkward to use. There are problems with getting the new system to interact with existing databases, and none of the printers work. FabriCo spends several months struggling with this system, and GammaCo switches all its attention to fixing the problems. Productivity at Fabrico tails off, several important customers are lost, and several key staff members accept jobs elsewhere. GammaCo never delivers version 2, and eventually FabriCo reverts to using the old system.
4. A small software company, DeltaCo has an idea for a new PC-based application. In order to understand the potential market, it conducts some market research, sending out questionnaires to existing customers, and setting up focus groups to come and discuss how they might react to the new product. Using the feedback from this market analysis, DeltaCo develops a working version of the application, and sends it out to a few trusted customers for evaluation. Based on feedback from these evaluations, the company makes some modifications, and then proceeds to market it as a shrink-wrapped product.
   Happy Ending: The new software gets rave reviews in trade magazines, sells widely, and wins several prestigious awards.
   Or perhaps: The sample customers used by DeltaCo during the development turn out to be significantly different from the target users identified by the market research. The opinions of the marketing manager conflict with the comments and requested changes collected from customers in the early trials. Some compromises are made, but in general, the customers' comments are given more weight. When the software is released, the marketing campaign falls flat. A few of DeltaCo's existing customers buy the software and love it, but in the end, so few copies are sold that DeltaCo never recoups the development costs, and eventually goes out of business.
5. The health service in a large city decides it needs a new computer-aided dispatch system for its fleet of ambulances. It puts out an invitation to tender, which contains a one-paragraph summary of the need. A number of companies that have "off-the-shelf" dispatch systems make bids. The health service then draws up a detailed comparison of the bids, and eventually selects the cheapest. The selected company, EpsilCo meets with the health service to discuss whether any further customization is needed to their system. Once the customizations are agreed, EpsilCo installs the new system.
   Happy Ending: The new system dramatically improves the efficiency of the ambulance dispatch process, so that response rates to emergency calls are reduced, as is the stress on the ambulance crews.

Or perhaps: EpsilCo's existing system has never been used for emergency dispatch systems before, as it was originally developed for dispatching taxi cabs in small towns. The system makes a number of incorrect assumptions about the crews' ability to communicate their location and status to the dispatch center. It also assumes that the nearest crew should always be dispatched to any emergency, irrespective of which base station that crew is associated with. When the system is installed, the crews find it hard to use the new terminals in their vehicles, and get frustrated at being sent to emergencies further and further from the areas they are familiar with. Crews get lost, and the dispatchers lose track of where some ambulances are. The system turns out to be unable to handle the volume of calls. After 48 hours of operation, the system is abandoned. A second trial two weeks later is also abandoned. The Health Service receives a huge number of complaints from the public, and several deaths during the period of the trials are attributed to delays in getting ambulances to the scene quickly enough.

6. A small real estate agency wishes to select a new spreadsheet for its office staff. The office manager draws up a list of the main spreadsheets available, and compares the advertised features of each. She selects a product, ZetaCalc, that contains all the features the staff need, at the lowest price.
   Happy ending: ZetaCalc is a significant improvement over the applications the staff previously used.
   Or perhaps: the user interface of ZetaCalc is significantly different from the one the staff are familiar with. Some staff members are sent on training courses, but still find ZetaCalc hard to use. Six months later, the office manager discovers that some staff have quietly reverted to the old software, while others are now using handwritten notes to keep track of data that they cannot enter into ZetaCalc. She calls a meeting, and the staff all vote to revert to their old spreadsheet. Nobody can remember why they thought they needed a new spreadsheet in the first place.
7. A large retail company, ShopCo, has an existing computerized billing system that was procured a few years ago from a company that no longer exists. ShopCo has recently started a new direct marketing service over the internet, and needs to modify the billing system to handle the new service. They engage a consultancy company, EtaCo to come and study the existing system, along with the few specifications that they have. EtaCo reverse-engineers the design of the existing system, and studies how the existing design might be modified to support the new service. They make the suggested changes, and test the new version of the system.
   Happy ending: The new version successfully handles the internet-based sales, and ShopCo is delighted that they now have detailed specifications of the billing system to help in future maintenance of the software.
   Or perhaps: Although EtaCo manages to reverse-engineer a model of the design of the billing system, they misunderstand how some of the design features respond to ShopCo's needs, and cannot distinguish features that are no longer needed from those that must be preserved. Their modifications do support the new internet service, but ShopCo now experiences a series of errors in bills issued to their traditional customers. Further, because some of the intermediate steps in processing bills have been eliminated, they find it hard to trace the cause of the problems. EtaCo makes two more attempts at fixing their new version, but some of the problems persist. Eventually ShopCo concludes that they need an entirely new billing system…
8. A large charity, FirstAid, wishes to introduce a system to provide its employees with better access to personnel records and payroll calculations. It procures an entire business system from IotaCo, a company that specializes in enterprise solutions. FirstAid then proceeds to re-design its personnel record-keeping and payroll processes to match the model used in IotaCo's system.
   Happy ending: The new system works well, and the employees are happy that they can correct payroll problems more readily. The new business processes that FirstAid put in place are more efficient than the old ones, and put FirstAid in line with industry best practices, which pleases its major corporate donors.
   Or Perhaps: The system introduced by IotaCo proves to be rather inflexible, and cannot

handle some of the special cases to do with distinguishing volunteer work from paid activities of its employees, nor with the arrangements for sharing employee's time with sister charities. The changes to the business processes required to use the new system turn out to be major, and FirstAid's staff are extremely unhappy at having to devote time to implementing the changes rather than concentrating on their charity work. Donations dry up because donors notice that FirstAid is spending less of its income on its aid projects.

1.3. Importance of Requirements Engineering

The examples in the previous section each have two very different endings: a good outcome and a bad outcome. It may seem that in many cases the choice between the endings is due to (good or bad) luck. While luck often does play a part in whether particular business decisions turn out well, it is a relatively small factor here. The major determining factor in each case is how well the organizations involved understand and manage their requirements. How do we know this? Let's look at the evidence.

1.3.1. The cost of fixing errors

One of the best-known investigations of the economics of software development was a series of studies conducted in the 1970's by Barry Boehm. Boehm investigated the cost to fix errors in the development of large software systems. Most of the projects followed a fairly standard sequence of phases: requirements analysis, design, coding (programming), development testing, acceptance testing and operation. Errors made in a particular phase cost more to fix the longer they are left undetected, so that, for example, a programming error that is not discovered until acceptance testing will cost ten times as much to fix than if it is found during programming. A requirements error not found until after delivery will cost a hundred times more to fix. The explanation is fairly simple – the longer a problem goes unnoticed, the more subsequent design decisions are based on it, and hence the more work it will take to unpick them.

Boehm's data invites a number of conclusions, the main one of which is that the strict sequence of phases described above (known as the waterfall model) may not be the best way to manage a large project – an iterative approach may be more suitable. We will examine this question in more detail in chapter 3. However, the conclusions concerning the relative cost of fixing errors do not necessarily depend on the order in which things are done, because the cost driver is not the length of time that an error goes undetected, but the number of other decisions that are based on it. Errors made in understanding the requirements will always have the potential for greatest cost to fix, because so many other design decisions depend on them. Clearly, time invested in understanding the requirements early in a project can have a significant payoff.

1.3.2. The causes of project failure

Through the 1990's, the Standish Group conducted a series of surveys of software development projects in the US, examining failure rates, and identifying critical success factors. In the initial study, in 1994, they found that only 16% of projects were successful. The survey defined a successful project as one that is completed on time and within budget, with all features and functions as originally specified. Note that this is a very strict definition of success – many projects may deliver a useful product that is a little late or over budget, say, or for which the original requirements have evolved during development. In the study, 53% of projects fell into this "challenged" category. However, the cost and schedule overruns were not small – among the challenged projects, there was an average 189% cost overrun and 222% time overrun. Either project estimation is hopelessly unrealistic, or the ability to manage projects according to plan is missing. Or perhaps both. Finally, 31% of projects were cancelled altogether before completion. This last figure led the Standish Group to estimate that American companies and government agencies would spend around $81 billion on cancelled projects in 1995.

A follow-up study in 1998 found a small improvement: the proportion of successful projects had grown from 16% to 26%, although the proportion of cancelled projects had barely changed, at 28%. However, there was a dramatic reduction in the average size of cost and schedule overruns for projects in the challenged category. The Standish Group attributed a great deal of the improvement to a major reduction in the typical *size* of projects. For example, in large companies, the average project size had fallen from $2.3M in 1994 to $1.2M in 1998. Smaller projects are easier to manage, and hence it is easier to control costs and schedule.

The most interesting part of this study was the list of success factors. When analyzing the causes of success or failure, the respondents to the survey cited the top three success factors as:

- user involvement;
- executive management support; and
- a clear statement of requirements.

The top three factors leading to failure were:

- lack of user input;
- incomplete requirements and specifications; and
- changing requirements and specifications.

Bear in mind that these are based on the *perceptions* of survey participants, rather than on hard data, so we have to interpret them carefully. But every one of these factors is a requirements engineering issue. Both users and executive management are crucial stakeholders whose interest and involvement in the project will be determined largely by how the requirements engineering processes are conducted. The dominance of requirements issues in the factors listed by respondents clearly indicates a consensus about where the main problem lies. Also worth noting was that "unrealistic expectations" figured prominently in the list of factors too – setting and managing appropriate expectations is also a crucial part of requirements engineering.

1.4. Human-Centered Design

We have characterized software-intensive systems as being embedded in the context of human activity, and it is this activity that gives them their purpose. Therefore, a study of human activities is a crucial part of requirements engineering. Throughout this book, we will emphasize the importance of *human-centered design.* Rather than considering design to be geared towards creating artifacts (machines, programs, devices, or even the 'systems' of systems engineering), human-centered design makes human activities the focus of the design process. The ultimate goal of all design processes is to change human activities to make them more effective, efficient, safe, enjoyable, etc. We may create new computer systems along the way, but these are only useful insofar as they support the human activities that we care about. Thus, human-centered design is not just the idea of making software more user-friendly, but represents a fundamental shift in perspective for what design is really about.

1.4.1. Soft Systems

The idea of human-centered design reflects a fundamental tension in requirements engineering. Many of the commonly used methods for analysis and design of software-intensive systems are based on what has been termed a *hard systems* view. Essentially, the view is that software design can proceed by decomposing a problem in a systematic way, using various modeling techniques to express the way in which data is processed by an organization. This view is central to the structured analysis methods of the 1970's and 1980's, as well as the object-oriented methods of the 1990's. However, the view is contested by a different tradition of design, known variously as *socio-technical* or *soft systems* methods. This view contends that system design cannot be reduced to a rational process because it always takes place in a complex organizational context, replete with multiple stakeholders and organizational politics. Such design is never a 'one-off' activity, but is rather part of an ongoing learning process by the organization. Nor can the 'requirements' ever be fully captured in a technical specification, so participation of users and other stakeholders at every stage of development is essential.

The soft systems view is perhaps best summed up by Peter Checkland and colleagues, most notably with the 1981 book "Systems Thinking, Systems Practice", and the development of Soft System Methodology (SSM). The perspective of SSM is that human activities can be thought of as systems, and general systems ideas can be applied to help understand them. Systems have emergent properties that only make sense when the system as a whole is understood – these properties cannot be decomposed across the parts of the system. Further, the decision to treat a particular set of activities as a system only makes sense from a particular perspective, and there are many possible perspectives on any human activity. Indeed, the perception that there is *a problem* that needs solving is itself the expression on a particular *worldview* (Checkland prefers the German word *Weltanshauung*). For these reasons, Checkland talks of *problem situations* rather than *problems*. As soon as you identify a specific problem, you have adopted a worldview that makes that problem relevant. Finally, SSM sees any design activity as a process of inquiry, or problem solving, which can itself be treated as a system. In other words, in SSM, one makes sense of a set of activities by treating them as a subject system, and then one *creates* a system for making interventions to improve the subject system, where evaluation of something as an 'improvement' is judged with respect to a particular worldview.

In much of the literature, the distinction between hard and soft systems approaches would seem to be irreconcilable. Either you believe in the rational, reductionist approach of standard software engineering methods, or you believe that software development is embedded in a complex social web of interdependencies, and can only ever be approached with a holistic, participatory approach. However, there is a simple reconciliation, hinted at in Checkland's writings. Basically, the 'hard' tradition can be viewed as a special case of the 'soft' approach, in which there is local agreement about the nature of the problem, and on the nature of a system to be engineered to solve it. Reaching such agreement may be difficult in some cases, and may indeed be *the* problem! But, if we assume that some local agreement is necessary for any kind of design, this perspective allows us to combine ideas and techniques from both traditions, which we seek to do throughout this book.

We will use soft systems ideas to identify and characterize human activity systems, so that we can study how they operate. We will examine techniques that allow us to explore how a human activity system will change as a result of introducing a new computer system. Most importantly, we will ask how to design the human activities and the software to fit together. By viewing requirements engineering as a continuing negotiation process between multiple perspectives, we keep the process human-centered. We also assume that various modelling techniques, both hard and soft, can be used to develop and capture points of local agreement between stakeholders.

1.4.2. Which systems are soft?

Whether the human-centered perspective strikes you as radical or commonplace may depend on what types of design you have been involved in. This is because different types of software have different levels of interaction with the human world.

At one extreme, we have various types of core operating system functionality, networking services, and middleware, which may have no user interface at all: their interaction with human activities is indirect, conducted through other software applications that make use of them. For such systems, the perspective shift doesn't change much. The functionality of such systems tends to be relatively stable, and tends to grow only because occasionally a function that was previously provided by some applications software gets moved into the operating system or middleware if it turns out to be more generally useful. However the perspective shift does lead to some significant insights into how computer systems have reshaped human activities. For example, the idea that a computer operating system provides a hierarchical file system is now firmly part of our mental model of how computers work. We now expect to organize our activities into things that can be stored in files, and the idea of explicitly saving a file is deeply entrenched; sophisticated computer users even exploit this when they open a file and use it as a "scratch copy" merely by not saving any changes. Similarly, the way in which internet protocols locate information has also become part of our common mental picture of the world. The expression "dotcom" arises purely by accident from the way that IP addresses are managed, and we regularly pass around URLs to point to information on the web. URLs were never intended to be visible to the user, and the fact that we use them so widely, in all their cumbersome syntax, indicates a failure to understand how human activities would be changed by the invention of the network protocols that form the world wide web.

Somewhere in the middle of the spectrum lie things like aircraft flight control. Here, the interconnection with human activities is more complex, but still restricted to a narrowly defined class of user (pilots and air traffic controllers) working on very well-defined activities (flying planes). Much of the functionality of such systems is determined by the physical processes that they control (e.g. the dynamics of heavier-than-air flight), rather than the human activities that they support. For such systems, it is common to talk about "the operator", and to design an "operator's interface", and a set of operational procedures. However, analysis of accidents indicates that treating the humans merely as 'operators' of a machine can be very dangerous. Many accidents occur because the system provided misleading feedback to the human operator at a crucial moment. Such accidents often get blamed on "operator error", but usually, they are the result of the designers not understanding how to design for people. Specifically, they occur when the designer tries to get the operator to adapt to the machine, rather than designing a system that will properly support the human activities.

The vast majority of software-intensive systems fall somewhere near the other end of the spectrum. Office automation software, information systems, groupware, web services, and the software that underlies nearly all sectors of commerce fall into this category. Such systems have complex, tightly coupled interactions with the world of human activity, and development of such systems results in significant changes to the human activity system in which they are embedded.

For these types of system, the cycle of continuous change is particularly important. In the landmark early work on software evolution, Lehman characterized such systems as E-type software (E for "embedded in the world of human activities"), to distinguish them from software that can be completely and precisely specified (S-type) and software that is designed to solve some precisely stated problem (P-type). E-type software is subject to continuous evolution, or it will become steadily less useful over time.

1.4.3. Studying Human Activities

To understand human activities, and how they will evolve in response to changes in the software systems that support them, we need to draw on ideas from psychology and sociology because we need to address both cognitive and social processes that underlie work activities. For example, we can draw on the work on user psychology in *Human-Computer Interaction* (HCI) to understand the perceptual limitations on people as they use computers, and the mental models that people form of computer systems.

Techniques, Methods, Processes…

Notations

A *notation* is a representation scheme (or modeling language) for expressing any of the things we want to study in RE. Examples include: dataflow diagrams, the Unified Modelling Language UML (which is really a set of notations), predicate calculus, and even natural language.

Techniques

A *technique* describes how to perform a particular technical activity, and, if appropriate, how to use a particular notation as part of that activity. The techniques we describe in this book cover the basic skills of the requirements analyst. Examples include: interviewing, calculating return on investment, use case diagramming, and conducting an inspection. Note that techniques generally do not have proper names – they tend to be generic building blocks for more specific methods and processes.

Methods

A *method* is a prescription for how to perform a related collection of technical activities, especially where a set of techniques and notations are needed. A method offers guidance for how to use the notations and techniques together to achieve a particular goal. Methods tend to be specific in their applicability: for example they concentrate on a particular type of modeling, or a particular type of application. Examples include Systems Analysis and Design Technique (SADT), Object Modelling Technique (OMT), the goal modelling method KAOS, and the Rational Unified Process (RUP). Note that methods tend to have proper names, given to them by whoever invented them. Note also that their names may include words like 'technique' and 'process' – don't be fooled, they are all methods really!

Process Models

A *Process Model* is an abstract description of how a particular organisation normally conducts a collection of activities, focusing on resource usage and dependencies between activities. The difference between methods and process models is that methods focus on technical activities (i.e. the content of activities), whereas process models concentrate on management of activities (i.e. how activities can be measured and improved). Hence, technical staff tend to be more interested in methods, while managers tend to be more interested in process models. Process models are usually developed within particular development organizations to capture successful processes so that they can be reused (and improved) from one project to another.

Processes

A *Process* is an enactment of a process model, describing the behaviour of one or more agents and their management of resources. The process model describes what should happen; a process is what actually happens on a particular project.

One of the trends that emerged in the 1990s in both HCI and RE is the use of *ethnographic techniques* to study how machines affect work activities. These techniques treat the workplace as a setting in which a variety of behaviors can be observed and understood, much as anthropologists study other cultures. In RE, ethnographic techniques have become a powerful means of uncovering subtle features of a current system, such as how teams manage to coordinate their activities. For example, in a study of a train control room the ethnographers noticed that the person making customer announcements knows what announcements to make because she sits next to the controller and is able follow his gaze, to see all the displays that the controller can see. Any new system that separates the role of announcer from that of the controller may break this shared context, and result in much greater overhead for the announcer to figure out which announcements to make when.

Another human-centered tradition is that of *participatory design*. Drawing its inspiration from the soft systems approaches, participatory design assumes that the development of any new computer system must involve and empower the workforce. By giving workers a high level of autonomy in the development of a new system, the result should be a better system for everyone, because the design draws on the detailed domain knowledge of those who actually carry out the work. In part, participatory design arose in response to the tendency for computer systems to dehumanize and deskill people. It also underpins methods developed in Norway and Sweden (collectively referred to as the "Scandinavian approach"), where is it tied into a political philosophy of workplace democracy. It is clear that active participation in the project by the stakeholders is an important factor in the success of any system development project. Hence, we will take the principles of participatory design as an ideal to strive towards, even when such full participation is hard to achieve in practice.

1.5. What do Requirements Analysts Do?

Because of the diversity of contexts in which Requirements Engineering is conducted, and the variety of systems to which it is applied, there is no 'one way' to do requirements engineering. The process for developing a specification for a control system will be very different from the process for exploring the requirements for a new information system. As we shall see in chapter 3, there are many different possible processes for building software intensive-systems, and RE has a role in each of them. However, there are a number of activities essential to RE that need to be addressed in one way or another in any RE process.

We can characterize some of these activities through a series of questions that a requirements analyst must address. Imagine a journalist, given a new story to cover. She must investigate the story, and then write it up in a concise way, perhaps to fit a certain number of column inches in the newspaper. Journalists often start by formulating a range of questions to ask, to get the basics of the story, using the seven question types: Who?, What?, Where?, When?, Why?, Which?, and How? We can apply this technique to requirements engineering by exploring the following questions:

- **Which problem needs to be solved?** Usually, some idea of a problem or opportunity has already been identified before a requirements-gathering exercise begins. But some choice is involved: choosing from a set of possible problems, or choosing the part of a large complex problem that can be feasibly solved. Equally important is the decision about which problems (or aspects of the problem) will not be solved. The requirements analyst is responsible for setting an appropriate scope for the project by identifying *Problem Boundaries*.
- **Where is the problem?** The discussion so far in this chapter has underlined the importance of understanding the *context* for any proposed software-intensive system. This includes an investigation of both the physical and organizational context: locations and organizational units that are affected by the problem, or which will need to be involved in implementing a solution. The *organizational context* is especially important for understanding whether the right problem is being tackled. The *physical context* is especially important for systems that are to be used in harsh or demanding environments (e.g. remote terrain, or busy control rooms). The requirements analyst is responsible for ensuring that all participants of the development process understand the *Problem Domain*.
- **Whose problem is it?** The development of any new system may have an impact upon a huge range of different people. Often, they will have competing needs, and different perceptions of the problem. The requirements analyst is responsible for identifying and characterizing all the relevant *Stakeholders*.
- **Why does it need solving?** In order to make good design decisions, it is necessary to understand the motivations and rationale that the stakeholders have for wanting the problem

solved. The requirements analyst is responsible for identifying and analyzing the stakeholders' *Goals*.

- **How might a software system help?** As we shall see in chapter 2, it is impossible to completely separate requirements from design. Hence, it is entirely appropriate to understand whether the stakeholders have any preconceptions of what the solution might look like, so that their expectations can be taken into account. The requirements analyst is responsible for identifying a set of *Scenarios* that capture the stakeholders' expectations for how the problem might be solved.
- **When does it need solving?** Providing a perfect solution to the problem a year after it was needed is unlikely to be acceptable. In the software industry, the delivery date is often set before anything else is agreed. Any constraints on schedule demanded by the application domain are valid requirements, as are other resource constraints such as cost, available staffing, and so on. The requirements analyst is responsible for identifying all the relevant *Development Constraints*.
- **What might prevent us from solving it?** Requirements engineering is crucial for risk management – requirements engineers must balance the selection and scoping of the problem with the feasibility of implementing a solution within the given constraints. The requirements analyst is responsible for analyzing *Feasibility* and *Risk*.

The above questions focus on information gathering activities, which is an appropriate focus for most of requirements engineering. In the process of making sense of the answers, requirements engineers may build models, and test them in various ways. They will communicate and negotiate with a variety of stakeholders, to reach agreement on the nature of the problem. They will help to bridge the gap between the problem and the solution, and manage the evolution that takes place as the problem (or peoples' understanding of it) changes. We have organized the remainder of the book around these different types of activity. All of these activities are important, and a requirements analyst needs to know how to select appropriate methods to support the different activities, as well as how to maintain a balance between them.

1.6. Chapter Summary

TBD

1.7. Further Reading

Complexity: The idea of a wicked problem was first described by Rittel, H. J., and M. M. Webber (1984). "Planning problems are wicked problems", In N. Cross (Ed.), Developments in Design Methodology, Wiley, pp. 135-144.

Abstraction, Decomposition and Projection: These are so basic to nearly all of computer science that they often get overlooked, even in introductory texts. One of the best introductions to their use in programming is Liskov and Guttag's "Program Development in Java: Abstraction, Specification and Object Oriented Design". For an clear, readable account of how they are used for problem analysis in requirements engineering, take a look at Michael Jackson's "Problem Frames: Analyzing and Structuring Software Development Problems". We will examine some of Jackson's ideas in more detail in chapter 2.

Modularity: Modularity is perhaps the most important idea in software engineering. One of the first papers to discuss it clearly was David Parnas' 1992 paper "On the criteria to be used in decomposing systems into modules". For an up-to-date description of the theory of modularity read Carliss Baldwin's book "Design Rules: the power of modularity".

Quality as fitness for purpose: Our definition of quality as fitness for purpose is due to Joseph Juran and the quality movement. Juran's "Quality Handbook" is a classic, but it is designed to be used as a desk reference, rather than a readable text, so it perhaps not the best place to start. Deming's "Out of the Crisis" makes a better introduction to idea of quality, and how statistical quality control applies in manufacturing industries. For a completely different take on the idea of "quality", read Robert Pirsig's "Zen and the Art of Motorcycle Maintenance".

Examples of RE in action: Some of the examples in section 1.2 are loosely adapted from real accounts of system failures. For example, the drivers license example is based on the Californian Division of Motor Vehicles, which commissioned a new computer system in 1987 and scrapped the project six years later. The ambulance service example is based on the London Ambulance Service's computerized dispatch system, which was commissioned in 1990 and abandoned in 1992. For more examples of project failure and their causes see Bruce Rocheleau's paper "Governmental Information System Problems and Failures: A Preliminary Review". Public Administration and Management: An Interactive Journal. 1997, Volume 2(3), and Nancy Leveson's book "Safeware". The Comp.Risks forum also collects stories from the media relating to system failures.

Empirical Data on importance of RE: Boehm's early studies on the cost of fixing errors is described in the first few chapters of his book "Software Engineering Economics", and the Standish reports can be found on their website at http://www.standishgroup.com/

Human-Centered Design: There are several good books on designing for the user, of which Don Norman's "The Design of Everyday Things" is perhaps the classic. However, these concentrate very much on user interfaces. For a more holistic account of human-centered design, read Kim Vicente's "The Human Factor: Revolutionizing the Way People Live With Technology", and Enid Mumford's "Redesigning Human Systems"

Soft Systems and Ethnography: The classic text on soft systems is Checkland's, "Soft Systems Methodology in Action". The use of ethnography in systems design was explored thoroughly in RE throughout the 1990's. Two early papers bear re-reading: Sommerville, Rodden, Sawyer, Bentley and Twidale's "Integrating ethnography into the requirements engineering process", and Goguen and Linde's "Techniques for Requirements Elicitation", both of which appeared in the first Requirements Engineering Symposium, RE'93. For a more up to date account, read Andy Crabtree's book "Designing Collaborative Systems: A Practical Guide to Ethnography". The control room example is from Heath and Luff's classic 1992 study, which was published in vol 1, issue 1 of the CSCW journal.

1.8. Exercises

TBD