Fundamentals of Requirements Engineering

CHAPTER 2

What are requirements?

The simple question "what are requirements?" turns out not to have a simple answer. In this chapter we will explore many of the key ideas that underlie requirements engineering. We will spend some time looking at two fundamental principles in requirements engineering: (1) that if we plan to build a new system, it is a good idea to describe the problem to be solved separately from particular solutions to the problem, and (2) that for most systems, this separation is impossible to achieve in practice. The tension between these two principles explains some wildly different perceptions of RE. We will break down the idea of a problem description into three components: the requirements (which are things in the world we would like to achieve), the domain properties (which are things that are true of the world anyway), and specifications (which are descriptions of what the system we are designing should do if it is to meet the requirements), and show how these are inter-related. Throughout the chapter we will emphasize that we are primarily concerned with systems of human activities, rather than 'software' or 'computers'.

Our aim in this chapter is to introduce a number of key ideas and distinctions that will help you understand what requirements engineering is all about. By the end of the chapter you should be able to:

- Distinguish between requirements and specifications, and describe the relationship between them.
- Explain how a system can meet its specification but still fail.
- Give examples of how misunderstanding of the domain properties can cause a system not to meet its requirements.
- Explain why any specification is likely to be imperfect.
- Distinguish between verification and validation, and give criteria for performing each.
- Explain the different perspectives of the systems engineer and the software engineer.
- Give examples of how the systems engineer can move the boundary between the application domain and the machine, to change the design problem.
- List the principal parts of a design pattern, and explain why a clear understanding of the requirements is needed to select appropriate design patterns.
- Use problem frames to describe the differences between major problem types such as control systems, information systems, and desktop application software.

2.1. Requirements Describe Problems

In chapter 1 we introduced the idea of capturing the purpose of a software-intensive system. To identify the purpose, we need to study the human activities that the system supports because it is these activities that give a system its purpose. Suppose we are setting out to design a new system, or perhaps to modify an existing system. Presumably, we have perceived an opportunity to use software technology to make some activities more efficient, or more effective, or perhaps to enable some new activities that are not currently feasible. But what, precisely, is the problem we are trying to solve? If we want to understand the purpose of the new system, then we need to be clear about what problem it is intended to solve.

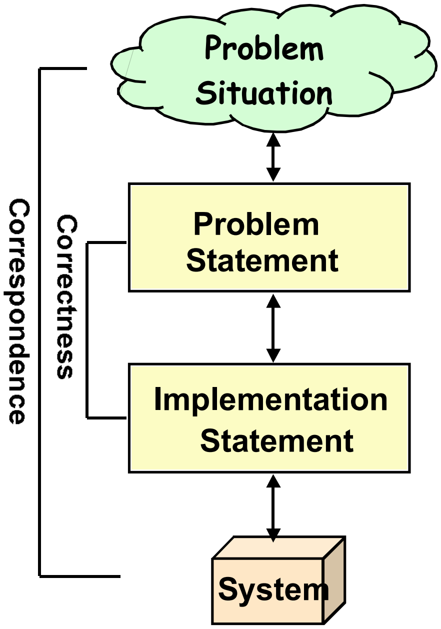

Figure 1: Separating the problem statement from the solution statement (adapted from Blum 1992)

2.1.1. Separating the Problem from the Solution

The first key insight of requirements engineering is that it is worthwhile to separate the *description of a problem* from the *description of a solution* to that problem. For a software-intensive system, the solution description includes anything that expresses the design: the program code, design drawings, the system architecture, user manuals, etc. The problem description is usually less well-documented – for some projects it may be captured in a ‘concept of operations’ document, or a ‘requirements specification’. For other projects, it may exist only in notes taken from discussions with customers or users, or in a collection of ‘user stories’ or ‘scenarios’. And for some projects there is no explicit statement of what the problem is, just a vague understanding of the problem in the minds of the developers. A basic principle of Requirements Engineering is that *problem statements* should be made explicit.

Separating the problem from potential solutions, and writing an explicit problem statement is useful for a number of reasons. To create a problem statement, we need to study the messy “real world”, ask questions about the activities that the new system should support, decide a suitable scope for the new system, and then write a precise description of the problem. This allows a designer to properly understand the nature of the problem, before considering how to solve it. The exercise of analyzing the real world problem situation will reveal many subtleties that might be missed if the developer launches straight into designing a solution:

- It might reveal that the most obvious problem is not really the right one to solve.
- The problem statement can be shared with various stakeholders, to initiate a discussion about whether their needs have been adequately captured.
- The problem statement can be used when comparing different potential designs, and when comparing design trade-offs.

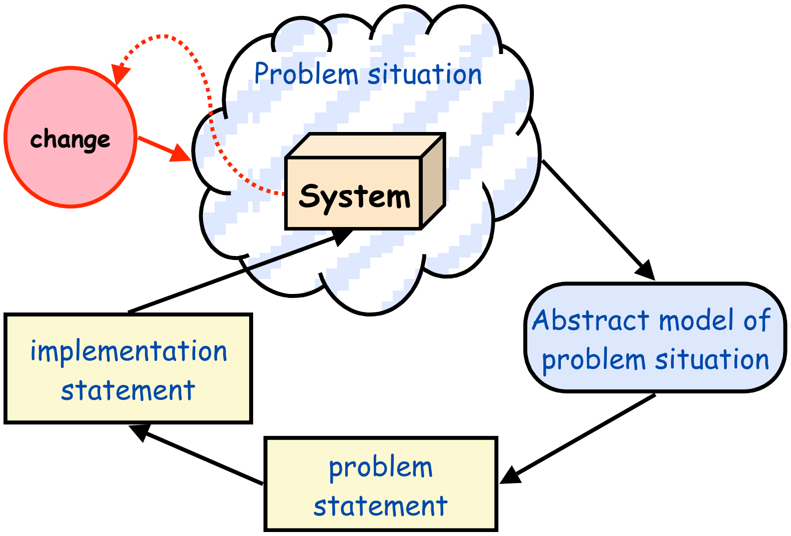

Figure 2: Delivering the completed system inevitably changes the problem.

- Making the problem statement explicit makes it much easier to test the system – a candidate solution is only correct if it solves the problem as stated.

Of course, the solution might still be unsatisfactory, because we might have made a poor job of writing down the problem statement, or we might have focused on the wrong problem. In other words we need to check both that the solution is *correct* according to the problem statement, and that the solution statement *corresponds* to the real-world needs of the stakeholders (see figure 1). But we have gained something by breaking this down into two separate steps – we can ask the question of whether the problem statement is adequate, independently from the question of testing whether a proposed design solves the problem as stated.

2.1.2. Intertwining of Problems and Solutions

The second key insight of requirements engineering is that this separation of problem statement from solution statement cannot really be done in practice. The real world in which human activities take place is complex, and any attempt to model and understand some piece of it will inevitably be imperfect. The world (and the people in it) change continually, so that the problem statement we write down at the beginning of a project may be wrong by the end of it. And because software technology opens up all kinds of new possibilities for how work is organized, the process of design itself will change the nature of the problem: show a user an early prototype of the system, and she will usually think of all sorts of new things she would like it to do. In some cases, it is only by attempting to design a new system that we start to understand whether there really is a problem that needs solving. For example, nearly all of the most practical uses of the world wide web weren't discovered until after the web was created, and most people never realized they needed cellphones with built-in cameras until they saw all the cool things you can do with them.

These observations lead to a fundamental tension at the heart of requirements engineering. While it is worthwhile separating the problem statement from the solution statement, in practice, this separation can rarely be fully achieved. People react to this tension in various ways. At one extreme, some developers attempt to 'freeze' the requirements early in a project, allowing

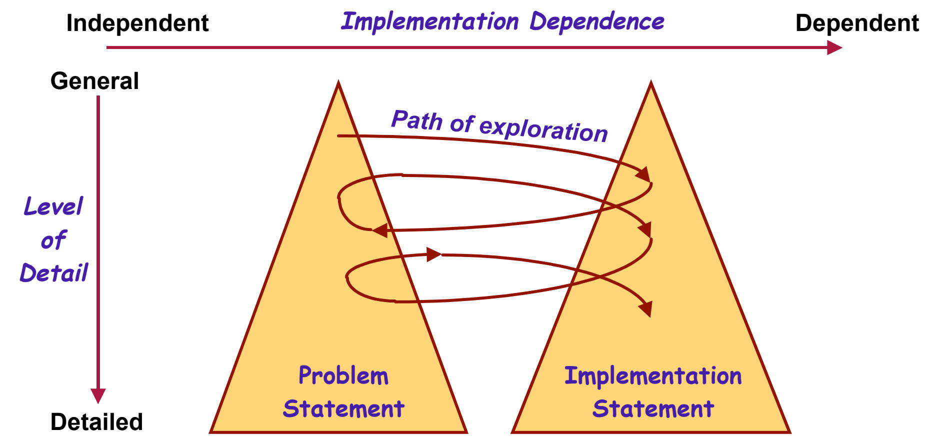

Figure 3: The Twin Peaks model. Exploration of the problem and its possible solutions are closely intertwined. (Adapted from Moffett 1999)

development to proceed in an isolated bubble, without having to worry about what is happening in the real world. This allows for a great degree of control of the engineering process, but runs the risk of developing a product that will be useless by the time it is delivered. At the other extreme, some developers discard all attempts to document the requirements explicitly, arguing that such effort will be wasted anyway. This allows for a high degree of 'agility' in responding to new ideas, but runs the risk of a chaotic development process that never converges on a useful product, or in which developers fail to understand what the system is really for.

Rather than adopting either of these extreme positions, we can use an understanding of this tension to explore some of the basic principles of RE, and to predict which techniques are likely to be useful:

- Firstly, the idea of separating the problem statement from the solution statement does not imply that these steps should be done in a particular order. Writing a problem statement is a way of capturing the current understanding of the purpose of a system, and can be useful at any stage of development. Rather than being the first phase of a project, requirements engineering is a set of activities that continues throughout the development process.
- Any version of the problem statement will be imperfect. The models produced as part of requirements engineering are only ever approximations of the world that the requirements analyst is trying to understand, and so will contain inaccuracies and inconsistencies, and will omit some information. Specifications are always imperfect, and there will usually be missed requirements. The requirements analyst needs to perform enough analysis to reduce the risk that such imperfections and missed requirements will cause serious problems, but that risk can never be reduced to zero. Requirements Engineering is therefore crucial for *risk management*.
- Perfecting a specification may not be cost-effective. Although we have characterized a number of benefits of writing an explicit problem statement, those benefits must be weighed against the cost of performing a requirements analysis. For different projects, the *cost-benefit balance* will be different. In large safety critical systems, the cost of producing and validating a detailed, formal requirements specification may be easy to justify, but for many other types of system, this cost may outweigh the benefits, and a much less rigorous problem description may be more appropriate.

- The problem statement should never be treated as fixed. Change is inevitable, and therefore must be planned for. Whatever process is used for producing a problem statement, there should be a way of incorporating changes periodically, or updating the problem statement as more is learned about the requirements. Even the process of building a new system changes the problem, so perhaps we should redraw figure 1 more like figure 2.

We characterized requirements engineering as being concerned with explicit statements of the problem to be solved. We could equally characterize it as a process of improving the developers' understanding of the problem to be solved. Sometimes that understanding is improved by trying to write a problem statement, and other times it is improved by attempting to design a solution to what you *think* the problem is. The diagram in figure 3, expresses this: an iteration of requirements and design activities, in which the understanding of both is increased at each iteration. It is illustrative to compare figures 1 and 3. Figure 1 represents an ideal, while figure 3 represents what often happens in reality.

2.2. Distinguishing the Problem

So requirements engineering is about describing problems separately from describing solutions to those problems, even though maintaining this separation is hard in practice. To make things harder, most stakeholders make no such distinction themselves. People are natural problem solvers – it is very hard to resist the temptation to solve a problem rather than merely describe it. Stakeholders often answer questions about what they need by describing their ideas about how they think the new system should work. To help keep the distinction clear, we need a way to decide which kinds of statements refer to problems, and which refer to solutions.

2.2.1. 'What' vs. 'How'

Early papers and textbooks on requirements engineering used to distinguish between requirements and designs by talking about 'what' versus 'how'. The distinction was introduced to illustrate the idea of *implementation bias*. A problem statement has implementation bias if it unnecessarily suggests or precludes particular design solutions. So, ideally, a requirements specification should describe *what* the problem is, without describing *how* it should be solved.

Unfortunately, this distinction is confusing, because the 'what' and 'how' will vary depending on the level of analysis. For example, the requirements for an overall system capture *what* that system is required to do, and the architectural design gives an indication of *how* the system will do it, in terms of a set of interconnected components. But then we still have to specify *what* each component should do (and we can only do so with reference to the overall design, or the '*how'*). So thinking in terms of 'what' versus 'how' does not help when trying to decide if a particular statement is a requirement or part of a design.

Another criticism of the 'what' versus 'how' distinction is that it leaves out other equally important questions, such as 'why' (why is this system needed? why should it behave like that?) and 'who' (whose problem is it anyway? who will use it?), and so on. Also, the problem of implementation bias is not fully addressed by separating the 'what' from the 'how', because there may well be good reasons why a customer *requires* certain design choices to be made. A classic example is the choice of programming language. This is clearly a 'how' issue, and for most projects should be a free design choice. But if a customer needs to maintain the software after delivery, and only has Java programmers available to do this (and expects this not to change for the life of the software), then insisting the software be written in Java is a perfectly valid requirement, even though it is a 'how' statement.

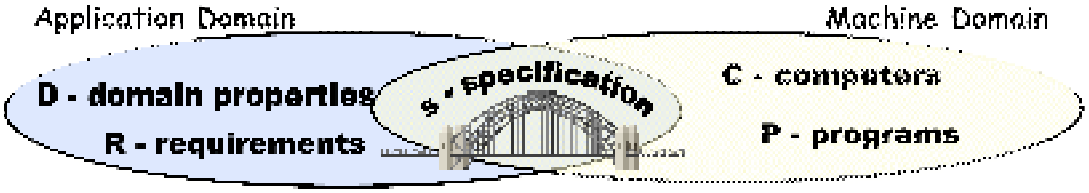

Figure 4: Application domain versus machine domain (Adapted from Jackson 1997)

Clearly, the distinction between 'what' and 'how' does not get us very far. The simplistic distinction we made between a 'problem statement' and 'solution statement' in the previous section also suffers from some of these criticisms. We clearly need a better way of understanding the distinction.

2.2.2. Application Domains vs. Machine Domains

A more appealing distinction is introduced by Michael Jackson, and concerns the difference between two different worlds that we might wish to describe – the *machine domain* and the *application domain*. Jackson uses the term 'machine' to describe the thing that is to be built. The term captures the notion of writing some software to turn a general-purpose hardware platform into a useful machine for a particular purpose[1]. The machine domain is the set of phenomena that the machine has access to: data structures it can manipulate, algorithms it can run, devices it can control, inputs it can get from the world, and so on. In contrast, the application domain is the world into which the machine will be introduced, and in particular, is that part of the world in which the machine's actions will be observed and evaluated. Given our characterization of requirements engineering as concerned with the purpose of a system, it should be clear by now that requirements are part of the application domain, rather than the machine domain. It is the application domain that provides a purpose for the machine, and so it is the application domain that determines the requirements.

The application domain and the machine domain must be connected somehow, because the machine must interact with the world in order to be useful. The connection is via shared phenomena – things that are observable both to the machine and to the application domain. Shared phenomena include events in the real world that the machine can directly sense (e.g. buttons being pushed, movements that sensors can detect) and actions in the real world that the machine can directly cause (e.g. images appearing on a screen, devices being turned on or off).

There are of course, many things in the world that the machine cannot directly sense, which we can think of as 'private phenomena' of the application domain. For example, the machine cannot know whether a person typing a password really is the person authorized to use that password. Requirements are, in general, about private phenomena, because a machine does not usually have access to the phenomena that define its purpose. For example, the requirement to allow only authorized personnel access to a building involves events and states that are private phenomena of the application domain, such as the identity of people, possession of authority, and people entering buildings. The machine senses most of these things indirectly, through devices that ask for passwords, mechanisms that lock or unlock doors, and so on.

[1] We should note that designing 'a machine' (in Jackson's terms) is very different from designing a 'software-intensive system' as we described in chapter 1. We will explain this difference shortly.

Finally, a *specification* for the machine can only be written in terms of the shared phenomena between the machine domain and the application domain. We cannot refer to private phenomena of the application domain in a specification, because we cannot (reasonably) specify what the machine should do in response to phenomena to which it has no access. We should not refer to private machine domain phenomena in the specification, because these have no role in giving the machine its purpose, and should be left to the designer to decide. The specification really only refers to things that cross the boundary between the application domain and the machine domain: primarily the inputs and outputs of the software, but also any ways in which the application domain constrains the design or operation of the machine.

Using these terms, we can recast the distinction between a problem statement and solution statement in the following way (see figure 4). The requirements analyst must be familiar with two aspects of the application domain: *the requirements*, which are things that the machine is required to make true (e.g. "prevent access to unauthorized personnel") and the *domain properties*, which are things that are true about the application domain irrespective of whether we build the machine or not (e.g. "only a manager can assign access authority"). Using these, the requirements analyst writes a specification for the machine, in terms of phenomena that are observable at its interface with the world (e.g. "when the user enters a valid password, the computer will unlock the door"). The programmer is then responsible for designing a program to run on a particular computer to meet this specification. Hence, a full description of the problem statement now has three components: the requirements, the domain properties and the specification.

A description of the problem statement can be said to suffer from implementation bias if it contains things that have no justification in the application domain. This gives us a much better way of determining whether something should be a requirement than the 'what' vs. 'how' distinction. For example, if in the application domain, there is an army of Java programmers available to maintain the software, then specifying that the software be written in Java is a valid requirement, even though it constrains *how* the software should be developed.

The role of the domain properties is crucial in this process. Domain properties help to link the specification and the requirements. Recall that the machine will probably not have direct access to the phenomena described in the requirements. For example, if a requirement is "to only allow access to authorized personnel", it is unlikely that the machine can directly sense who is authorized and who is not. However, we can make use of domain properties such as the fact that authorized personnel can be issued with passwords, that authorized personnel can be trusted to keep these passwords secure, and will be able to remember them when needed. This allows us to write a specification that refers to "entering a password", which is an input that the machine *can* sense. Of course, if we have misunderstood the domain properties, we may end up with a program that satisfies its specification, but does not meet the original requirements – in this example, the password software may work correctly according to its specification, but the overall system may not meet its *purpose* because security may be breached when passwords are shared with unauthorized personnel.

Whether certain domain properties will hold or not depends on the context in which we use the system once it is developed. A system that meets its requirements when used in one context might not do so when used in a different context. For example, our security system might work fine in an office environment where people are familiar with the need to remember their passwords and keep them confidential, but fail entirely in a care home for the elderly, where the residents share their passwords with each other because they can never remember them themselves. Sometimes we can make the design simpler by assuming that certain domain properties will hold, even if we know they can be violated. In this case we are deliberately restricting the kinds of context in which a system should be used. One of the reasons for explicitly capturing the domain properties is so that we have a record of such assumptions.

2.2.3. Verification and Validation

Using these distinctions, we can now return to the question of quality – does the system meet its intended purpose? Figure 1 showed two separate aspects of this question:

- Verifying correctness (or just *verification*), by which we mean checking that a design solution correctly solves the stated problem. Using Jackson's terms, we can break this into two separate criteria:
  1. The program, running on a particular computer, satisfies its specification.
  2. The specification satisfies the stated application domain requirements, assuming the stated domain properties hold.
- Validating the problem statement (or just *validation*), by which we mean checking the correspondence between the problem we have stated and the demands of the real world. Again we can break this into two parts:
  1. Did we discover and understand all the relevant requirements?
  2. Did we discover and understand all the relevant domain properties?

If we know all the properties of the program and the computer on which it is run, and we express the specification sufficiently precisely, the first verification criterion is entirely objective, and could conceivably be automated. By 'objective', we mean the outcome should not depend on the opinions of the person performing the test, nor on how she interprets the specification. Increasingly, this verification criterion *is* checked automatically, through automated software testing, and/or formal proof techniques. Similarly, if we have a precise specification, and write down the requirements and domain properties sufficiently precisely, the second verification criterion should also be entirely objective, and could conceivably be automated. However, it is hard (and perhaps expensive) to write down the requirements and domain properties so precisely, and therefore this step is usually performed manually, if it is checked at all.

In contrast, validation steps are always subjective by their very nature. Our understanding of both the domain properties and the requirements involve an assessment of what is true of the real world. Two different people may disagree on whether the problem statement is valid, because they may disagree on whether certain domain properties really are true, or they may have different understandings of the real requirements.

We can illustrate the difference between verification and validation with another example, also due to Jackson. For an aircraft, it is important to prevent accidental engagement of reverse thrust while the aircraft is flying. This is an important safety requirement – it is especially dangerous to engage reverse thrust while in the air. We can express this as the following requirement:

> R1: "Reverse thrust should be enabled only when the aircraft is moving on the runway, and disabled at all other times".

However, because the control software cannot directly sense the state "moving on the runway", we need to find a way of connecting this to phenomena that the machine can detect. A standard solution is to use the sensors on the wheels that pulse when the wheels are turning. We can then address the requirement using this specification:

> S1: "reverse thrust should be enabled if and only if wheel pulses are on".

Note that we are using some assumptions about the domain properties to connect the specification to the requirement:

> D1: "wheel pulses are on if and only if wheels are turning".
> D2: "wheels are turning if and only if aircraft is moving on the runway".

To *verify* the software, we could write test cases that check whether it meets its specification, i.e. that reverse thrust is enabled when wheel pulses are on, and disabled if they are not. For complex software, this may require a very large set of test cases, because we may need to test whether these properties hold in every possible mode. In some cases, there will be so many combinations that we cannot possibly test them all. Even if we manage to complete this testing, we have still only covered the first verification criterion. We could attempt to check the second verification criterion during system testing, by building the aircraft, and checking that reverse thrust is indeed enabled only when the aircraft is moving on the runway, perhaps with the help of a wind tunnel. However, such testing is very crude – it is impossible to perform this test under all possible flight conditions. More likely we will have to rely on careful reasoning, using a model of the domain properties, and a model of the requirements.

To *validate* the problem statement, we need to check that the requirement and the domain properties adequately capture what happens in reality. If it is ever possible for the wheels to turn while the aircraft is in the air, or fail to turn when it is moving on the runway, we have a problem. In fact, D2 is not always true. In one accident, an aircraft touched down very lightly on a wet runway, so that reverse thrust would not engage[2]. Even if we fix this error in our understanding of the domain properties, we can never be absolutely sure that there are not other circumstances under which our understanding of the world is still wrong. We can probably always think up some circumstances that do break the assumptions about the domain (what if mice stow away in the wheel compartments, and use the wheels as exercise wheels during the flight?), so an important part of requirements analysis is to assess the risk of each such circumstance, and decide whether the specification should be altered to handle them.

2.3. Software Problems or System Problems?

In chapter 1, we presented requirements engineering as an essential part of any development of software intensive systems, and argued that a key distinguishing feature of the design of such systems is that it inevitably involves the design of some of the human activities that the software is to support. Yet, in the previous sections, we defined requirements only in relation to specifying and building a *machine*, and we had to assume that the people using the machine would behave in a particular way – for example, we had to assume they would do the right thing with passwords. We assumed that our job was only to design the machine, and that the application domain was fixed, along with the boundary between the two domains.

In practice, the requirements analyst must also decide where this boundary lies, whether it can be moved to help redefine the problem, and hence whether parts of the application domain itself should be changed. For example, if we take seriously our argument about designing the human activities, then we should consider the whole question of whether passwords are the right way to detect authorization, and address the design of a process for issuing and protecting passwords, or whatever tokens of authority we eventually decide on.

The issue here is whether we are doing software engineering or systems engineering. Software engineering tends to assume that the hardware platform, and the devices with which it can interact with the world, are fixed, and the job is to write some software to make it all work. In contrast, systems engineering concerns itself with the development of an entire system, which may comprise a variety of components, including software, hardware, mechanical devices, and human operators. The systems engineer must examine how these various subsystems will interact, and how various functions should be allocated to different components. For example, should the selection and allocation of passwords be carried out by a human, by a mechanical device, or by a software algorithm? Should there be redundancy (more than one person, device or computer)? Essentially, the systems engineer must decide where to draw the boundaries.

[2] The incident was Lufthansa flight DLH 2904 from Frankfurt to Warsaw on 14 September 1993. There were several factors involved, but the fact that the reverse thrust and spoilers failed to deploy when the aircraft first touched down was a major factor. The flight computer failed to detect the touchdown, the pilot was unable to engage reverse thrust to slow the aircraft, and the plane overshot the end of the runway, crashed and caught fire with the loss of two lives.

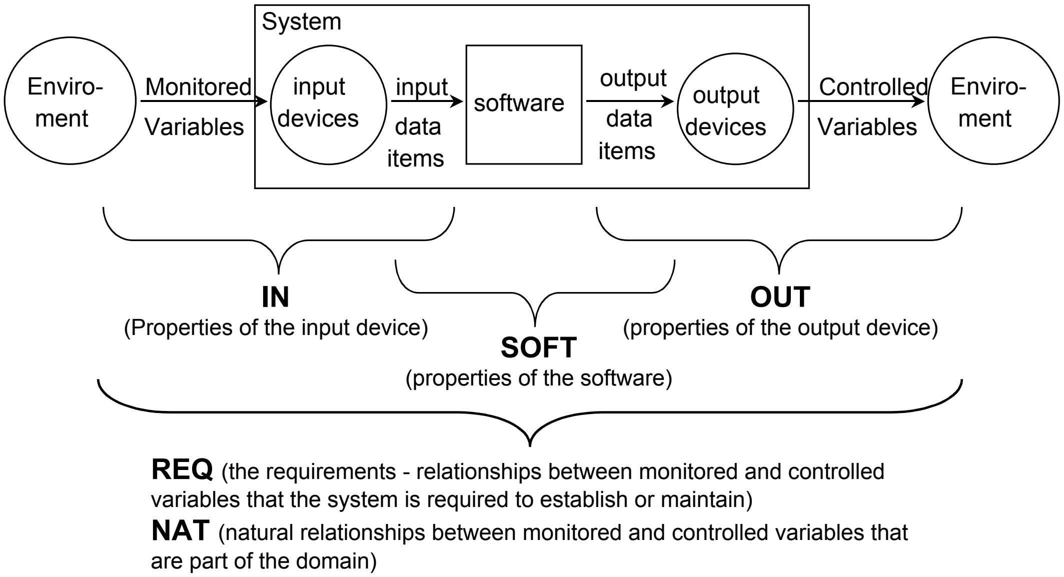
System
Enviro-ment
Monitored Variables
input devices
input data items
software
output data items
output devices
Controlled Variables
Enviro-ment
IN
(Properties of the input device)
SOFT
(properties of the software)
OUT
(properties of the output device)
REQ (the requirements - relationships between monitored and controlled variables that the system is required to establish or maintain)
NAT (natural relationships between monitored and controlled variables that are part of the domain)

Figure 5: The Four Variable Model (adapted from Parnas)

Another way of viewing the distinction between systems and software engineering is offered by considering the 4-variable model suggested by David Parnas (see figure 5). This model was originally conceived as a way of understanding real-time control systems, but the key ideas generalize nicely. A control system continually monitors some environmental variables (e.g. for flight control: altitude, windspeed, groundspeed, etc). In response to changes in the monitored variables, it manipulates some controlled variables (e.g. the angle of wing flaps, thrust from the jets, etc) in order to satisfy some control policy. However, the monitored and controlled variables are not directly accessible to the software, so we rely on input and output devices that map these environmental variables onto shared phenomena for the machine. For control systems, the input and output devices are known as sensors and actuators.

The *system* requirements are expressed in terms of a *desired* relationship between monitored and controlled variables. The system is *required* to maintain this relationship. There may be additional relationships between monitored and controlled variables that are natural properties of the domain (precisely Jackson's application domain properties). For example, if the thrusters are fired, the aircraft will accelerate, and so groundspeed will change.

The *software* requirements are expressed in terms of a desired relationship between the input and output variables. Clearly, the *software* requirements, together with the properties of the input and output devices should guarantee that the *system* requirements are met. But a *systems engineer* must also choose what types of sensors and actuators to use, and different choices may make the software easier or harder to design.

For example, in most elevator systems, the control software has no way of detecting when people actually want to use the elevator – it has to rely on buttons being pressed. It is possible to add additional sensors to detect when there is anyone in the elevator, and whether anyone is actually standing waiting at a floor (whether or not a button has been pressed). These could make the elevator more efficient, by cutting down wasted journeys when people press wrong buttons either accidentally or maliciously. In effect, such sensors take things that were previously private phenomena of the application domain, and make them shared phenomena with the machine. But the cost (and poor accuracy!) of these extra sensors may well outweigh the added benefit. A systems engineer must weigh up these trade-offs.

Both software engineering and systems engineering are relatively young disciplines, compared to traditional branches of engineering such as electrical engineering. One of the discernable trends in both fields as they have developed over the past decade is a move towards the use of standardized solutions to common types of problem. An example is the trend towards component-based systems, which attempt to use standard, "off-the-shelf" components wherever possible.

Unfortunately, there is often a tendency to assume that any deficiencies in the existing components can be fixed by adding more *software* during the design process; because software is so flexible, it can make up for problems elsewhere. Unfortunately, software is often the least well-understood part of any system. The assumption that such problems can be left to fix later using software patches demonstrates an inadequate risk analysis. Instead of fully considering whether a different system architecture would be a better choice, the risk is pushed towards the least understood components, to be addressed late in the design process, once the rest of the system has been created.

Requirements engineering seeks to overcome this problem by providing more detailed system-level analysis early in the design process, so that such risks can be more accurately assessed. Software is always embedded in a larger system, and in some cases this system too must be specified and designed. It is the requirements analyst's job to decide where the boundaries should be drawn, and which functions should be allocated to which types of component. This includes deciding which activities will be done by the software, and which will be done by people.

2.4. Requirements Patterns and Problem Types

Although we have described some general principles about requirements, the actual requirements for different types of system may look very different. This is because different methods for discovering and expressing requirements have been developed for different application domains. It remains an open question how much the experience gained in analyzing the requirements on one project can be re-used on other projects. One possibility is to look for patterns that seem to recur over different projects. Such patterns might offer a way of re-using our experience, as well as insights into similarities and differences between different problem types. Here we will consider the way patterns have been discovered in both designs and in requirements.

2.4.1. Design Patterns

Much of the work on design patterns for software engineering takes its inspiration from the work of the architect Christopher Alexander, who first catalogued patterns in architectural design and invented a pattern language for expressing them. In 1994 the "gang of four" (Gamma, Helm, Johnson and Vlissides) published their book "Design Patterns", in which they applied this idea to patterns found in object-oriented programming. The idea was immediately appealing because it captured patterns that were familiar to most programmers, but which had not previously been documented in any systematic way. The design patterns were closely tied to programming problems. Around the same time, the first book on software architecture appeared, in which Shaw and Garlan identified a number of common software architecture patterns, and carefully described the qualities of each.

These books beautifully illustrated the benefits of a pattern language. Each pattern captures both a design problem, and a well-known solution. Or, in the words of Alexander, "each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves".

Name: Contract
Context: any kind of financial deal
Problem: how to represent the transaction of buying and selling
Forces: need to distinguish two parties; the buyer's and seller's views look different; a deal really involves 2 instruments, but one is usually money, and so is implicit in many deals; …
Solution:

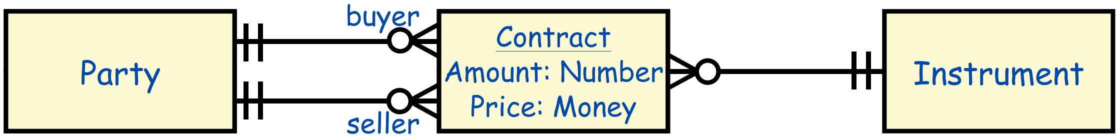

Figure 6: An analysis pattern (Adapted from Fowler)

Hence, patterns capture a little of both a problem statement and a design solution, including the context in which the pattern is applicable. A typical pattern description includes:

- a name for the pattern,
- a problem statement,
- the context in which it occurs,
- a description of the forces, a design solution,
- and cross-references to related patterns.

The emphasis on forces is particularly important – these are the inter-related set of requirements and constraints that surround the design problem. Understanding these forces means understanding the design trade-offs, and hence the rationale for using the pattern.

In a design process, the designer might be aware of a large number of useful patterns, and will apply one whenever she recognizes that some aspect of the current design problem matches the interplay of forces described in one of the patterns. But note that the use of design patterns is not a substitute for requirements analysis. On the contrary, it is only once the requirements are understood that it becomes possible to find good matches between the current problem and any of the patterns in the catalogue. The alternative is to apply patterns blindly in the hope that because they were useful before, they may be useful in the current context. This is a sure recipe for solving the wrong problem!

2.4.2. *Requirements Patterns*

So, design patterns do not help with analyzing requirements, but they do help in mapping between requirements and designs. However, there is another potential use of patterns in requirements engineering: we can look for re-usable patterns in the requirements themselves. Just as with design, the idea of capturing past experience in the form of typical patterns is appealing. Requirements patterns do not capture design solutions, but capture the "spatial configuration" of particular types of problem, and present a way of understanding and describing that problem. In essence, they describe problematic human activities, or particular arrangements of the world where we can see an opportunity for change. A catalogue of requirements patterns would help us to recognize different types of problem.

The idea of requirements patterns is relatively new, so there is as yet little consensus on what requirements patterns should look like. Here, we will briefly consider two quite different approaches: the analysis patterns identified by Martin Fowler, and the problem frames defined by Michael Jackson.

Fowler's analysis patterns are intended to be used to help build conceptual models of relevant parts of the application domain. The solution part of each of his patterns is represented as a fragment of the Unified Modelling Language (UML). Consider the example in figure 6. Here, the problem is how to model a *transaction* in the context of any kind of financial trading. A deal involves a buyer and a seller, but their views of the transaction are different. Technically speaking, a deal involves an exchange of one thing for another, so there should be two instruments involved, but as one of them is usually money, it is not clear whether this should be modeled separately. Fowler's solution in this pattern is to model the deal as a contract, with the price as an attribute. The annotations on the relationships indicate that a contract must have exactly one party who is the buyer, and exactly one party who is the seller. However, each party can participate in any number of deals as a buyer and any number of deals as a seller. Similarly, each deal involves exactly one instrument (although the quantity can be specified as a attribute of the contract), but each instrument can be involved in any number of deals. The pattern provides a standard solution to the problem of modeling transactions, and also offers suggestions for when this standard solution might not be appropriate. We will meet both UML and Fowler's patterns again in chapter 9, when we consider the role of modelling in requirements engineering.

Jackson's problem frames are quite different in intent: they are designed to help us to sort out different kinds of problem and problem decompositions as a prelude to more detailed requirements analysis. There are so many different types of problem to which software-intensive systems are solutions that it helps to have a high-level classification scheme for different problem classes. A problem frame provides an initial decomposition of a problem into principle parts and a solution task. Although, at first sight, problem frames may look overly simplistic, they provide a useful starting point for understanding what the problem description might be for a given problem.

Figure 7 shows some simple examples of problem frames. In each case, the machine to be built is drawn as a box with a double border, and the requirement it is expected to satisfy is shown as an oval. The remaining boxes describe other relevant parts of the application domain that need to be understood. For example, in the simple information display frame, the problem is to build a machine that will maintain a representation of some part of the real world (e.g. bank accounts), will accept information requests (e.g. account requests) and in response provide information outputs (e.g. account statements). The relationship between the current state of the real world, and the information output expected for each request is determined by the information function (e.g. banking rules), which the machine is required to apply.

As well as providing a starting point for problem analysis, Jackson's problem frames offer an initial typology of problems to which software-intensive systems might be applied. They demonstrate that information systems have a very different shape than, say, control systems. Here are Jackson's five basic frames – each represents an entire class of software system, and for each class, a different set of requirements engineering methods is likely to be appropriate:

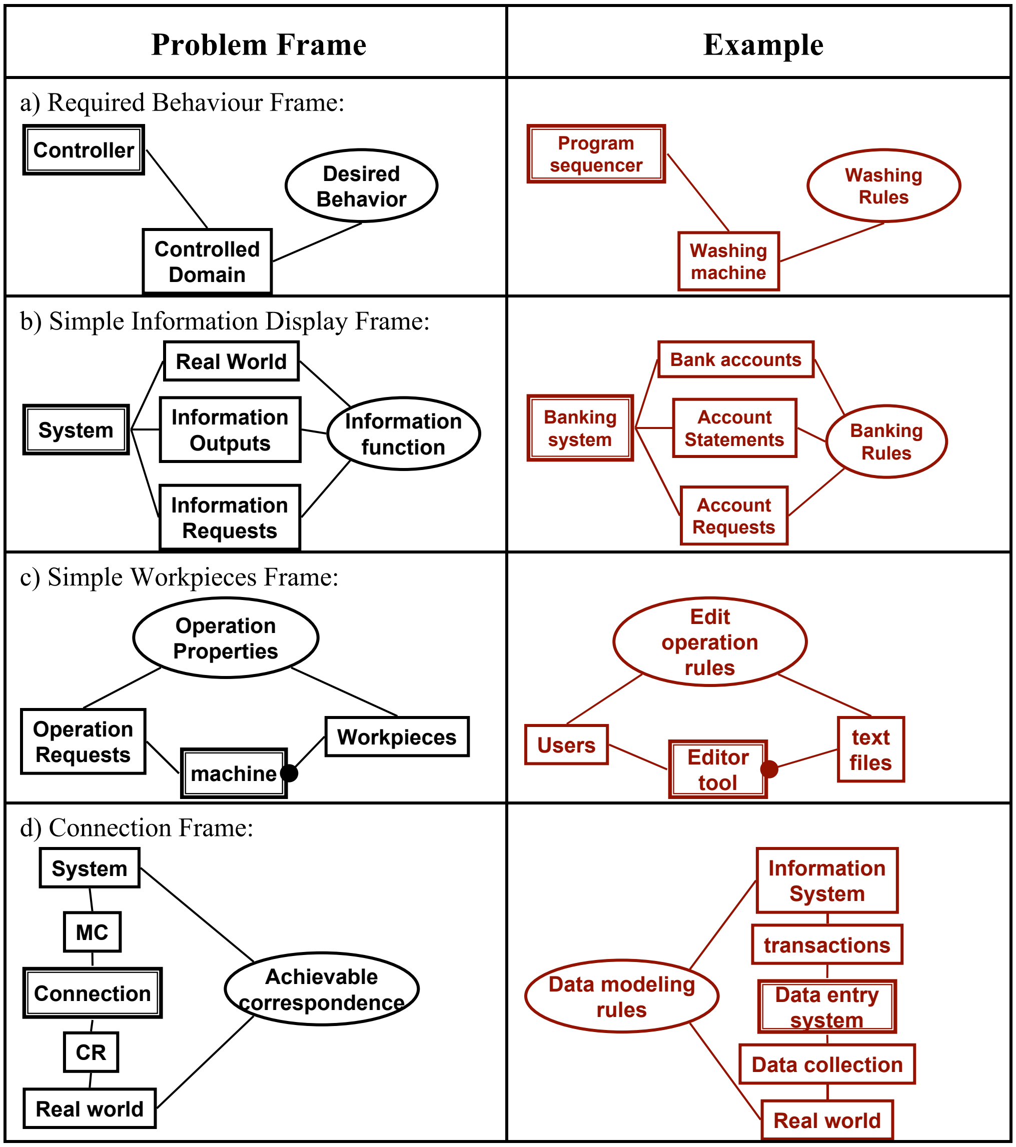

Figure 7: Some example problem frames (adapted from Jackson 1997)

- *Required behaviour* (figure 7a). The problem is to build a machine to control some part of the real world in accordance with a fixed set of control rules. The solution will be an automated control system.
- *Commanded Behaviour*. The problem is to build a machine that allows some part of the real world to be controlled by an operator by issuing commands. The solution will be a “human-in-the-loop” control system.
- *Information Display* (figure 7b). The problem is to provide information about the current state of some part of the real world in an appropriate form and in an appropriate place, in response to information requests. The solution will be an information system.
- *Simple workpieces frame* (figure 7c). The problem is to keep track of the edits that are performed to some workpiece, for example a text file or a graphical object. The solution is likely to be some kind of application software, such as a word processor.

- *Transformation*. The problem is to take some input data represented in a certain format, and provide a transformation of that data according to a certain set of rules. Solutions to this type of problem include traditional data processing applications, as well as tools such as compilers.

2.5. Chapter Summary

TBD

2.6. Further Reading

Requirements vs. Specifications: One of the best all-round introductions to requirements engineering is Michael Jackson's "Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices". It covers many of the same ideas described in this chapter, and is nicely divided up in to bite-size essays for easy browsing. We've followed several of Jackson's key ideas in this chapter, most notably on the distinction between requirements and specifications. Note that this distinction is not universally accepted in RE – many other authors suggest that there is no distinction, and that the requirements *are* what is written in the specification. For example, Suzanne and James Robertson, in their book "Mastering the Requirements Process" provide a template for documenting requirements as part of a requirements specification. The IEEE standard on requirements (IEEE-Std-830-1998) also makes no distinction. We will revisit this difference of opinion in chapter 14, when we consider how to document the requirements.

Intertwining problems and solutions: The idea of separating a description of the problem from a description of the solution appears throughout the requirements literature. The fact that this cannot really be done, however, was quite a radical idea once, perhaps summed best in Swartout and Balzer's paper "On the inevitable intertwining of specification and implementation", Communications of the ACM, vol 25, no 7, 1982. For a more recent account, including the twin peaks model of figure 2, see Nuseibeh's paper "Weaving the Software Development Process Between Requirements and Architectures" IEEE Computer, Vol 34, No 2, 2001.

Application Domains: Jackson's distinctions between requirements, domain properties and specifications is best described in his paper "The Meaning of Requirements", which appeared in the Annals of Software Engineering, vol 3, 1997. This volume was a special issue on Requirements Engineering, and many of the other papers are also worth reading. The ideas are also elaborated in the paper "The Four Dark Corners of Requirements Engineering", which appeared in ACM Transactions on Software Engineering and Methodology also in 1997.

The 4-variable model was introduced by Parnas and colleagues. It is described in detail in the paper by Parnas and Madey "Functional Documents for Computer Systems", which appeared in the Science of Computer Programming, vol 25, no 1, 1995.

Patterns and Problem Frames: The original work on patterns was due to the architect Christopher Alexander, in his book "Notes on the Synthesis of Form". The ideas were popularized in the software engineering community by the book "Design Patterns" by Gamma, Helm, Johnson and Vlissides. The original book on software architecture, Garlan and Shaw's "Software Architecture: Perspectives on an Emerging Discipline" is also a patterns book in all but name. Martin Fowler took patterns to a higher level with his book "Analysis Patterns", and now of course you can find books on patterns of just about anything in software engineering. For problem frames, read Michael Jackson's "Problem Frames: Analyzing and Structuring Software Development Problems".

Key distinctions in RE

Problem Description vs. Solution Description

Requirements Engineering assumes that it is useful to separate a description of the problem being solved from a description of a particular solution. This distinction is useful for communicating with customers, and for weighing up different design solutions. However, the problem and the solution interact, so that it is impossible to make this distinction entirely cleanly.

What vs. How

Traditionally, a specification states 'what' a system should do, without saying 'how' it should do it. The reason for this distinction is to prevent 'solution bias' in the statement of the problem – we should not write a problem statement in such a way as to suggest certain solutions, or preclude others. However, there may be good reasons to prefer some solutions over others, and we do need to capture these reasons, so 'what' versus 'how' is too simplistic for most purposes.

Application Domain vs. Machine Domain

It is useful to distinguish between the world in which the problem exists (the 'application domain') and the world in which software solutions are developed (the 'machine domain'). These worlds overlap in a limited way, and it is this overlap that allows us to take the application domain requirements that we care about and translate them into relationships between inputs and outputs that the software can control.

Functional vs. Non-functional Requirements

Functional requirements capture the functions that a system must perform, while non-functional requirements capture general properties about the system, such as its speed, usability, safety, reliability, and so on. Non-functional requirements are often also called 'system qualities', or 'ilities'.

Systems Engineering vs. Software Engineering

Requirements engineering applies to both systems engineering and software engineering. For software engineering, it is normally assumed that the way in which the software interacts with the world is fixed, using standardized types of input and output device. For systems engineering, no such assumptions are made – the task is to design an entire system, of which the software is just one component.

Customers vs. Users

In requirements engineering, we normally use the term 'stakeholders' to indicate in the broadest terms all the different groups of people who may be affected by a new system, and therefore who might have requirements that need to be considered. Two important subgroups are customers and users. However, these are distinct roles: the customers are those who are responsible for commissioning a new system, while the users (or 'end-users') are the people who will interact with the system once it is installed. Only in special cases are these the same person or people.

Indicative vs. Optative Descriptions

In describing a problem, it is often necessary to talk about both the current situation, and the future envisaged situation once we have designed a solution. An indicative statement describes the world as it is now, while an optative statement describes a state of affairs that we would like to bring about. The distinction is important because we need to understand which things can be assumed about the world, and which things the software is required to bring about. Application domain properties are indicative, whereas requirements are optative.

Verification vs. Validation

Verification is the process of determining that a program meets its specification, whereas validation is the task of making sure that the system will address the right problem in the real world. Many people remember the distinction as: "verification means are we solving the problem right; validation means are we solving the right problem". This quote captures the essence of the distinction, but remember that the distinction only makes sense when you consider the role of a specification.

Capturing vs. Synthesizing Requirements

Stakeholders often do not know what they want or what is possible. We may not know exactly who the customers and users will be for some systems. Under these circumstances, it is wrong to think of requirements as being 'out there' ready to be captured. Rather, they need to be negotiated or even invented. But this does not mean we just make them up. Instead, we synthesize the requirements based on our best understanding of the problem we are trying to address, along with reasonable estimates for the unknowns.