# Fundamentals of Requirements Engineering

CHAPTER 3

# What is Engineering?

In this chapter we explore the context in which requirements engineering takes place. We start with an analysis of engineering itself, and compare the engineering of software-intensive systems with other types of engineering. An engineering discipline captures and codifies the knowledge and practices needed by engineers to design complex devices. It also seeks to use experience from past projects so that engineering practice can be continually improved. To understand how an engineering project forms the context for RE, we need to understand how engineering know-how is applied in a systematic way as a project unfolds. We will examine two types of model: lifecycle models and process models. Lifecycle models are ecological descriptions of the stages that an engineering project goes through, much as we may observe and describe the stages in the lifecycle of a butterfly. As there are a large variety of species of engineering project, so there are a large variety of lifecycle models. Lifecycle models are useful for comparing project types, but are too simplistic for the management tasks of measuring and controlling a particular project, and for assessing where new techniques and tools may be applied. Detailed process models address this need, and so we will briefly examine the idea of process modeling and improvement.

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

- Define the term engineering, and distinguish engineering from science.
- Distinguish between normal design and radical design.
- Explain why systems engineering is fundamentally about radical design.
- Identify the attributes of software that make software engineering significantly different from engineering of physical devices.
- Summarize the basic tenets of an engineering code of ethics.
- Evaluate the ethical issues in any given requirements engineering activity.
- Describe the ways in which a manager can control an engineering project.
- Differentiate between types of engineering project based on the existence (or otherwise) of a customer before and during the project.
- Discuss how the idea of a product family affects the way in which requirements are identified and prioritized.
- Critique the use of design considerations arising from reuse of existing components to constrain the requirements.
- Summarize the strengths and weaknesses of the waterfall model with respect to requirements engineering issues.
- Describe the differences between a phased and an iterative development model.
- Summarize the key stages in the lifecycle of a requirement.
- Contrast process modeling and improvement with agile development and give examples of the types of project for which each is appropriate.

## 3.1. Engineering

Requirements Engineering is a collection of activities that only make sense as part of a larger engineering project. The requirements activities will play a significant role in scoping and guiding the overall project, but the overall project will also constrain and guide the RE activities. As we saw in the first chapter, there are a huge variety of different kinds of engineering project in which RE will play a role. Later in this chapter, we will examine this variety, so that we can compare different projects both in terms of how they are initiated, and how they are organized. Before we get there, we will examine engineering itself.

### 3.1.1. What is Engineering?

A typical definition of engineering runs as follows: "Engineering is the development of cost-effective solutions to practical problems, through the application of scientific knowledge". Several parts of this definition are important:

- *Cost-effective* – engineering involves a consideration of design trade-offs, especially those to do with resource usage. Cost is often used as a common evaluation criterion in judging whether a particular engineering solution to a problem is a good one, but good engineering also demands a wider definition of cost-effectiveness, involving wise use of all resources, and minimizing any negative impacts of a particular solution.
- *Solutions* – engineering emphasizes the design of solutions, usually tangible artifacts (we will refer to them as *devices* for convenience, and we'll explain this term shortly)
- *Practical problems* – the problems that engineers tackle are those that matter to people; engineering has a wider concern with improvements to human life in general, through technological advance. Of course, some engineering solutions to practical problems may turn out to have detrimental effects (on the environment, for example), however, the overall concern of the engineering profession is to make the world a better (or at least more convenient) place for humankind.
- *Applying scientific knowledge* – a key feature that distinguishes engineering from other forms of design (for example, clothing design, furniture design, etc) is the systematic application of analytical techniques grounded in science and mathematics, both to analyze the problem, and to guide design choices in creating a solution.

Many people believe fail to understand the relationship between science and engineering. The key difference lies in the idea of intervention. Scientists seek to understand the world through observation and experimentation, but do not seek to change the world. Engineers, on the other hand, seek enough understanding in order to make a change to the world, but do not regard any scientific knowledge that they generate or use in the process as an end in itself. Engineering degree programmes (and the professional bodies that accredit them) enshrine this idea by trying to achieve a balance between the teaching of two complementary strands, 'engineering science' and 'engineering design'.

It is also common amongst scientists to regard engineers as 'users' of the knowledge that scientists create. However, the history of engineering clearly indicates that this is not an accurate view of the relationship between science and engineering. The important ideas used in each of the engineering disciplines were typically discovered and codified as design principles well ahead of the corresponding scientific advances. For example, engineers were able to build reliable and cost effective bridges long before science was ever able to provide a complete analysis of the materials and forces involved in bridge building.

Engineers develop and validate theories in much the same way that scientists do, and indeed each design that an engineer creates is, in itself, a theory about some relevant aspect of the world. However, engineering science may involve a level of approximation that is unacceptable in the 'pure' sciences. For scientists, the existence of such approximations is often the impetus for further investigation, while the engineer will quite happily use approximations if they are good enough for the problem in hand. So, scientific advances may help to improve engineering practice, but such advances aren't necessarily the drivers of engineering advances. The *science* in engineering science is quite distinct from the science taught in science degree programmes.

## Engineering Design vs. Engineering Science

- Engineering Design "…is the process of devising a system, component, or process to meet desired needs. It is a decision-making process (often iterative), in which the basic science and mathematics and engineering sciences are applied to convert resources optimally to meet a stated objective. Among the fundamental elements of the design process are the establishment of objectives and criteria, synthesis, analysis, construction, testing, and evaluation. The engineering design component of a curriculum must include most of the following features: development of student creativity, use of open-ended problems, development and use of modern design theory and methodology, formulation of design problem statements and specifications, consideration of alternative solutions, feasibility considerations, production processes, concurrent engineering design, and detailed system descriptions. Further, it is essential to include a variety of realistic constraints, such as economic factors, safety, reliability, aesthetics, ethics, and social impact."
- Engineering Science: "…has its roots in mathematics and basic sciences but carries knowledge further toward creative application. These studies provide a bridge between mathematics and basic sciences on the one hand and engineering practice on the other. Such subjects include mechanics, thermodynamics, electrical and electronic circuits, materials science, transport phenomena, and computer science (other than computer programming skills), along with other subjects depending upon the discipline."

(Adapted from the ABET criteria)

The two engineering disciplines most relevant to this book are software engineering and systems engineering. The term *software engineering* can be traced to around 1968, when it was used as a provocative title for a NATO conference in Garmisch, Germany, intended to examine why software was proving so expensive to produce. The term *systems engineering* has been in use in the aerospace and defense industries since the 1970's, but has recently been popularized by INCOSE, the international council on systems engineering, which was founded (originally as a national society in the US) in 1990. The term *systems engineer* was coined in reaction to the fact that large systems have become so complex that there is a need for a new set of skills to do with understanding and controlling the complex interactions in the design of such systems (such as aircraft, spacecraft, weapons systems, medical devices, etc). These systems do not necessarily contain software, although they invariably do, and by and large it is software that has *allowed* such systems to become so sophisticated and complex.

### 3.1.2. From Devices to Systems

A key distinction in engineering is between *normal design* and *radical design.* In normal design, the engineer knows how the device she is designing works, understands the usual features of such devices, and also understands the design principles that apply to such devices. Engineering practice is, in general, concerned almost exclusively with normal design – it codifies the knowledge and principles involved in the normal design of well-understood devices. Radical design, on the other hand, involves the creation of solutions to problems that have never been solved before, or the creation of novel solutions by combining devices in new ways. The development of the first internal combustion engine was radical design. The development of variants of these engines in the modern automobile industry is normal design. In most engineering disciplines, normal design is very common, while radical design is relatively rare. The design principles that engineers are taught are the principles of normal design.

For normal design, requirements engineering is relatively straightforward, although certainly not trivial. The device to be designed has well-known known properties, such that a specification for the device to be built will look very similar to specifications for other, similar devices. The set of design choices that still have to be made is constrained, and the principles on which these choices must be based are well-understood. Therefore, the requirements process focuses on two main activities – an initial matching of the problem type to a particular class of known devices that solve such problems, and a focused analysis of just those aspects of the problem that will determine how to resolve the remaining design decisions. This is not to say that such requirements analysis is easy; it is however much more tightly constrained than requirements analysis for radical design.

Software engineering, and especially software engineering education, has been criticized for focusing too much on radical design. Students of software engineering are taught to develop systems from scratch, solving each problem as though that problem had never been solved before. This criticism is usually followed by a call for software engineering to become more like other engineering disciplines, by codifying well-known solutions to routine design problems, so that software development becomes *normal design*. Whether you believe such criticism is valid depends on whether or not you think software engineering (and, as we shall see, systems engineering) *should be* like other engineering disciplines.

One of the reasons that we have used the term ‘device’ to describe the things that engineers design is so that we can contrast ‘devices’ with ‘systems’. By *device*, we mean an entity whose properties and design principles are well-known. By *system*, we mean an assembly of devices brought together to solve a complex (and perhaps poorly understood) design problem. The behaviour of individual devices may be well understood, but the system itself is not. In particular, systems tend to have emergent properties that are not at all obvious from a consideration of the component devices. If we build enough similar systems, we may come to understand the principles involved, such that they can be codified into a normal design. To keep the terminology consistent, we can say that in this case we can eventually start to treat a well-understood system as a device.

Recent advances in software engineering indicate that some parts of software engineering are moving towards normal design. The characterization of architectural patterns for certain kinds of system, and the analysis of the design principles needed to apply such patterns, indicate that software engineering (as a discipline) is beginning to turn some of its systems into devices. For example, the architecture and design principles of compilers are now sufficiently well understood that we can consider them to be devices. On the other hand, many of the software-intensive systems that we described in the last chapter continue to defy such codification into normal design, and perhaps always will, because of the complex nature of the human activity systems in which they are enmeshed. Just as we think we might understand how to engineer some business information system, advances in technology and changing user perceptions will change the problem so much that we can’t just re-use a standard design from last year’s solution.

With respect to this discussion, systems engineering would appear to be an anomaly as an engineering discipline, because its concern *is* with complex systems, rather than devices. By definition, systems engineering can never be codified into the kinds of normal design that dominate other engineering disciplines. Indeed, as soon as such codification occurs, the resulting devices will no longer be the concern of systems engineers – they become the concern of one of the other engineering disciplines that systems engineers rely on. However, this does not mean that the set of problems that systems engineers tackle will shrink over time. On the contrary, the trend is to attempt to exploit software-intensive technologies to solve ever more complex problems, and to combine software components (‘devices’) into new systems in ever more inventive ways.

It therefore seems likely that systems engineering, and perhaps many areas of software engineering, will always be very different from other engineering disciplines, because they are concerned with the radical design of complex systems. Because of this, analysis of requirements for both systems and software engineering will remain a major challenge. In this book we will address requirements engineering both for radical design and for normal design, although the former will occupy us for longer, as it offers a much greater challenge.

### *3.1.3. Is Software Engineering Different?*

Development of software-intensive systems presents a number of challenges beyond those found in other engineering disciplines. The first of these is the concern for complex systems (rather than relatively well-behaved devices) as we described above.

Other challenges come from the nature of software itself. Because software is not a physical artifact, we cannot appeal to the laws of physics to understand how software will behave. This poses challenges in understanding how software failures occur, and how to make software more robust. To illustrate the point, consider the difference between a physical component, such as a steel beam, and a software component such as a procedure (or a method):

- Small changes to the forces to which the steel beam is exposed will cause the beam to deform in small, predictable ways. In general, the behaviour of physical components can be described using equations in continuous mathematics. As we increase the load on the beam, it will continue to deflect in the same direction. Discontinuities in behaviour are rare, but do occur occasionally (for example, if our beam snaps). A robust design avoids such discontinuities, by ensuring components operate within a carefully chosen range of operating conditions. For example, to ensure our steel beam is strong enough for a particular task, we could choose a beam design that can withstand double (or treble) the expected maximum load.
- Small changes to the inputs to a method call may cause wildly different behaviour, and the changes are hard to predict without a careful analysis of every line of code. The behaviour of software cannot be described using continuous mathematics, but rather, requires discrete mathematics. If we wish to make a program robust, we cannot use the same trick as was used for the steel beam, because the concept of doubling the load makes no sense in a discrete system. Even if we can express some aspect of the function of a software system in continuous terms (for example, speed of an operation), the resulting design changes are usually not continuous when we need to vary these quantities. If we want a procedure to run twice as fast, we may have to redesign it completely, rather than just varying some design parameter in a continuous way.

Furthermore, when a software component fails, there are no physical limits on the propagation of such failure. In physical systems, when a component fails, the components that are physically close to it are most likely to be affected, and propagation of the failure is constrained by physics. Of course, in some cases, such as an explosion, the physical propagation can be extremely fast. However, software failures can propagate in ways that have nothing to do with physical proximity. An error in the output of a procedure can cause failures in any other part of the system, no matter where they are physically located. An extreme case of this occurred on NASA's Mars Climate Orbiter, when a small error in a data file in the ground support system on Earth led to the loss of the spacecraft hundreds of millions of miles away as it approached Mars.

Another key difference is that software is not manufactured. In the engineering of physical devices, due consideration must be given to how the device is to be manufactured. At some point, the design must be frozen, so that manufacture can begin. Software does not need to be manufactured – it can be replicated at virtually no cost at any moment. This has two important consequences. The first is that there is no manufacturing variability – every copy is a perfect replica, and *all* defects in the software are *design* errors. The second is that we can keep changing the design, even after delivery to the customer. Many software developers exploit this fact by shipping software to the customer before it is fully designed, with the expectation that any weaknesses in the design can be corrected in a future release.

Software never wears out, and so, in theory should never need replacing. In practice, we replace software regularly, both to correct design defects, and to add new functionality. Software that keeps doing the same job year after year is likely to become less and less useful. So although it doesn't wear out, it does 'age'. As the original problem that it was designed to solve changes over time, so the software becomes less and less well suited to its purpose. We will examine this continuous evolution of both software systems, and the requirements on which they are based, in chapter 18.

### 3.1.4. Engineering as a Profession

Our definition of engineering stressed the importance of careful use of resources, and on solving problems that matter to people. In many engineering disciplines, the consequences of poor engineering practice can be catastrophic – if engineered devices fail, they can lead to serious injury, death, destruction of property, or financial loss. Because of the scope for harm, in many countries there are professional engineering bodies that license (or charter) engineers. These bodies set standards for engineering practice, and work to protect the public interest. Typically they grant a license to an engineer only after she has taken an accredited engineering educational programme, and has passed various professional examinations.

An important foundation for a profession is a code of ethics, which spell out the responsibilities of engineers to use their knowledge and skills in an honest and ethical manner. As an example, the following box summarizes a code of ethics for Software Engineers developed by the IEEE and ACM.

## ACM/IEEE code of ethics

*PUBLIC*

Software engineers shall act consistently with the public interest.

*CLIENT AND EMPLOYER*

Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.

*PRODUCT*

Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

*JUDGEMENT*

Software engineers shall maintain integrity and independence in their professional judgment.

*MANAGEMENT*

Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

*PROFESSION*

Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

*COLLEAGUES*

Software engineers shall be fair to and supportive of their colleagues.

*SELF*

Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

For full version, see http://www.acm.org/serving/se/code.htm

There are several key issues to do with professional conduct that are particularly relevant to requirements engineering:

- Competence – A central part of any code of ethics concerns the appropriate use of good engineering practice, and the competence of engineers to do so. For example, professional engineers should never misrepresent their level of competence, nor should they knowingly accept work that demands skills beyond their competence.
- Confidentiality – Engineers are bound to respect the confidentiality of employers and clients, irrespective of whether a formal confidentiality agreement has been signed. Requirements analysts often need to interview and collect data from a wide variety of stakeholders, and appropriate steps need to be taken to protect confidentiality of all such stakeholders.
- Intellectual property rights – Engineers need to be aware of local laws governing use of intellectual property, such as patents, and copyright protection. Information collected during requirements gathering may be protected in various ways.
- Data protection – Many countries now have data protection laws that restrict how data held in computer systems can be used, and create rights for people to access (and correct mistakes in) data that refers to them. Compliance with such laws is an important requirement for any information system, and requirements analysts need to be familiar with the appropriate laws and the scope of the jurisdictions in which they apply.

Although a code of ethics describes a set of basic principles, application to any given situation can be difficult, as ethical questions often occur where two or more principles come into conflict with each another. For example, having interviewed a number of employees in a client's organization, and agreed to keep their responses confidential, a requirements analyst may discover that the data collected reveal that employees (individually or collectively) are acting in ways that management is not aware of. If they are ignoring safety standards or fiddling the accounts, say, the right course of action might be obvious.

More commonly, there is no obvious course of action. The analyst may be working in a situation where there is an existing conflict between the interests of different groups (for example, employees versus management, one customer versus another, etc), or between different engineering principles. Two examples should illustrate the point:

- A requirements analyst is conducting an observational study of new accounting software at a large consultancy company. Staff have consented for their interaction with the software to be recorded while they carry out their work. They have been told that it is not them, but the software that is being evaluated, and that their anonymity will be protected. During the study, it becomes clear that many of the junior consultants are making a series of systematic data entry errors, thus causing the company to lose profit. Company policy clearly states that in such cases, employees should have their salaries docked to cover the shortfall. Should the requirements analyst report the errors (thus potentially breaking confidentiality promised in the study), or should she cover up the problem (thus potentially falsifying the data)?
- A requirements analyst is analyzing the requirements for the personnel office of a small manufacturing plant. Her initial specification includes fairly stringent security requirements, reflecting the sensitivity of the data to be stored, as it includes employee performance evaluations, medical histories used in insurance claims, and so on. Her client deletes these requirements from the specification, arguing that it will be too expensive to meet them. However, without these requirements, it may be relatively easy for employees to figure out how to access one another's records, and the system may be easy to hack into from outside the company. She tries to explain these risks to the managers, but the head of personnel and the director of computing regard the security requirements as "gold plating", and hence unnecessary. What should she do?

### 3.1.5. Managing an Engineering Project

If the engineering of complex systems is difficult, then the *management* of such engineering projects is even harder. Management involves different skills from those needed for the technical engineering work. Managers need to understand how the project is progressing, and adjust the resources available in response. Although project management is not a major concern of this book, we give a brief account of it here as part of the context for understanding how management decisions interact with requirements engineering activities.

In some sense, a good manager is often invisible. When the project is going well, the engineering staff may not notice that the manager is doing anything at all, even when the success of the project is directly due to her decisions. The job of managing a successful project will often look very simple in hindsight, while the technical staff will get the credit for a job well done. On the other hand, failure is usually very obvious, and managers will tend to get the blame no matter what role they actually played.

The project manager is responsible for ensuring at the outset of the project that the necessary resources (funding, people, time, etc) are in place, and that the task to be carried out is correctly understood. From these, she can develop a project plan. During the project, she must measure progress, and assess how such progress (or lack thereof) affects the plan, adjusting the plan and/or the resources as necessary. At the end of the project, she is responsible for ensuring that the right lessons are captured, for use in future projects.

In essence, a manager can control four things:

- Resources – including funding, personnel, facilities and technical infrastructure. If the project is not going well, the manager may need to find additional resources (e.g. more personnel, better tools, etc), or re-assess how effectively the current resources are being used.
- Time – the project schedule is a key variable that can be adjusted as necessary. Often the simplest solution to an unforeseen problem is to slip the schedule by delaying key milestones or deliverables. Unfortunately, for many projects, this is the area where the manager has least flexibility, because "time-to-market" has become a dominating factor in the software industry.
- Product – the manager can change the product to be built, for example by reducing the planned functionality if the project is behind schedule, or even adding functional enhancements if the project is ahead of schedule. Reducing the scope of the product to be built during a project is often referred to as a 'requirements scrub', and is greatly facilitated if the manager has a clear idea of the relative importance of, and dependencies between, the various requirements.
- Risk – if a project is going badly, and the manager does not adjust any of the previous three factors, the result will be an increase in *risk*: for example, the risk of missing the delivery date or the risk of shipping a product that is not ready and still contains defects. Risk in itself is neither good nor bad, but necessary – if we don't take risks we are unlikely to achieve anything worthwhile. Risk can be treated as a resource to be traded against the other three variables; sometimes it may be appropriate to take bigger risks, if the consequences of failure are not likely to be catastrophic, and sometimes it may be sensible to take steps to reduce a risk that has become unacceptable, by adjusting one of the other variables appropriately.

Central to all of these issues is the problem of measurement. A manager cannot control a project if she cannot measure it. Software measurement is a topic for an entire textbook, but we can give at least a few pointers here. As a starting point, any sensible measurement program must start with a clear idea of what the measures will be used for. If we wish to understand how to adjust the four project management variables described above, then we need some basic information on how we are doing with respect to each variable. Hence, a core set of metrics might include measures of:

- Effort – how much effort will be needed? How much have we expended so far?
- Time – what is the expected schedule? How much are we currently deviating from it?
- Size – how much functionality are we planning to build? How much have we built already?

- Defects – how many errors are we making? What proportion of those errors are we detecting? How does this compare to our quality goals?

It shouldn't be hard to see that each of these four measures connects directly with a corresponding project variable, while focusing on things that are relatively easy to measure. Too often, projects are managed on the basis of only two of these metrics: Effort and Time. Lawrence Putnam likens this to flying an aircraft using only a fuel gauge and a stopwatch. Without data on the size and quality of the system being built, a manager has no sound basis for making adjustments.

Another common mistake is to confuse one of these measures with another. For example, when asked how big the system they are developing is, some managers answer in terms of 'man-months' (i.e. effort) instead of some measure of functionality. This is because we do not yet have good metrics for the size of complex software-intensive systems. This is an important question for requirements engineering, because a statement of the requirements is often the only benchmark available for assessing the size of a proposed system.

At the beginning of a project, the project manager needs good *estimates* of effort, time, size and risk, in order to plan good use of project resources. Good estimates of risk are especially important for assessing how much flexibility there is when adjusting the plan. As the project proceeds, measures of progress can be compared with the estimates on a regular basis, to find out whether the project is proceeding according to the plan. If they are diverging, either the plan or the project will need adjusting.

Good estimation depends on two things: accurate data from previous projects, and a clear understanding of the requirements. Initial estimates of size of the system to be built can be obtained from a consideration of the number of requirements, and the anticipated difficulty in meeting them. Time and effort can then be derived from size using data from similar projects in the past. Risk can be assessed by considering the importance attached to each requirement and the consequences of not meeting it.

The relationship between requirements engineering and good project management should now be clear. The requirements act as the basis for estimation and planning; without them a project may simply be unmanageable.

## 3.2. Engineering Projects

Before we look at engineering project lifecycles, we will consider the context in which such projects are initiated:

- *Projects get started* for a huge variety of reasons, ranging from an explicit contract with a customer, to a speculative development of some new idea or new technology.
- Projects vary according to how easy it is to identify a *customer*, who can act as the ultimate authority for requirements decisions.
- In most cases, there is nearly always *some existing system* to be enhanced or replaced, because whatever problem we are seeking to solve, people must have some way of dealing with it currently, even if that just means coping with the absence of a good solution.
- There are nearly always *existing products or components* that we can use to build some part of the solution, and which therefore may constrain how we scope the problem.
- Engineering projects do not exist in isolation, but are usually part of some larger business strategy, which may involve an interrelated *family of products*. For example, a particular system may be part of a coherent product line for an organization, or may be one step in a wider marketing strategy.

We will discuss each of these ideas in turn.

### 3.2.1. Project Initiation

A project to develop a software-intensive system may be initiated for any of the following reasons:

- *Problem-driven*: A problem is encountered in some human activities that requires a response. For example, a company may find that their competitors are beating them in the market place; a series of accidents indicate that an existing system is unsafe; or a crisis occurs that an organization finds it cannot handle adequately. The solution to the problem may take many forms: an entirely new system where none existed previously; the replacement of a system that is not performing well; or an integration of several existing systems to improve communication and coordination between them.
- *Change-driven*: Changes to a business or its environment mean that changes are needed in the systems on which it depends. This may be because of growth, expansion into new markets, changes in technology, changes in legal constraints, improvements to business practices, or changes in the cost or availability of resources. Often, the response will be to enhance or adapt an existing system. For example, new functions may be added to a system, or an existing system may be converted to a new platform. Alternatively, it may be that the existing systems cannot be adapted, and must be replaced entirely.
- *Opportunity-driven*: Development of new technology opens up new possibilities, or a new market opportunity is identified. Such projects tend to be exploratory in nature and perhaps riskier. The opportunity could result in new products or services, or new ways of carrying out some existing activity. For example an existing function might be computerized to exploit a low-cost new technology.
- *Legacy-driven*: Occasionally, a project arises because some previous plan called for it, or because an earlier project left some work unfinished. This is perhaps the trickiest type of project for the requirements analyst, because the rationale for the project might be unclear, or no longer valid. Unless the rationale is carefully re-examined, such a project is unlikely to go smoothly.

In any of these cases, the value of the potential outcome must be weighed against the expected development costs, to determine whether the project is worth doing. In chapter 6 we will examine some specific techniques for calculating these factors.

Purely economic factors may not entirely explain why an organization undertakes a particular project – organizational politics also plays a role. For example, initiation of a particular project may owe more to the pet interests of a particular executive, or an attempt to gain control over resources by some unit within the organization than it does to any explicit economic benefit. Such circumstances make the requirements analyst's job much harder. Even though they play an important role in shaping the requirements, making them explicit may not be possible or desirable.

### 3.2.2. The role of the customer

Projects also vary according to how easy it is to identify *the customer(s)* – those stakeholders who will derive the primary benefit of the proposed system, and who will therefore be willing to pay for (at least part of) the development costs. For example a project may be:

- *Customer-specific* – there is a specific customer is who needs a system to solve a specific problem, and who is available as the main authority in determining the requirements. The customer may be internal to the organization developing the system (e.g. another unit within the organization) or external, in which case there will most likely be a customer-supplier contract in place. Such contracts vary greatly in how much requirements detail they include. Customer-specific projects are also sometimes known as 'bespoke' systems.
- *Market-based* – rather than a specific customer, the system is developed to be sold to a broad market. In some cases an established market may already exist (because similar products are

already available); in others the product must create its own market. For market-driven projects, a marketing team may act as a substitute for a real customer, as they investigate the market opportunities, and determine which features will improve marketability.

- *Community-based* – some projects are targeted neither at specific customers, nor at particular markets. Rather, such projects are intended as a general benefit to a broad community, whether through creation of free services or infrastructure, or the creation of new knowledge. Many open source and free software projects fall into this category – they are funded through the donation of the developers' time, and provide a benefit to a large (but ill-defined) community. Systems created for scientific exploration also fall into this category – for example spacecraft for exploring the solar system, or systems for collecting data about climate change. Such projects may be *funded* by governments or research agencies, but these act as *enablers* rather than *customers* if they do not have a direct stake in shaping the requirements.
- *Hybrid* – various combinations of customer-specific, market-based and community-based projects are possible. For example, a system might be developed for a specific customer, but with an intention to market the system more widely once it is developed. Or a government agency may act as a customer for a community-based project for which it is also the major funder. In these cases, there may be a tension between the competing goals – making a system fit well to a specific customer's needs may make it too specific for the more general marketplace, or make it less useful to a target community. Relative priorities of the competing goals need to be understood when determining the importance of different requirements.

Problem-driven, customer-specific projects are perhaps the best understood, and were the target of much of the research in requirements engineering in the 1970's and 80's. Here, the key challenge is how to analyze the problem to discover the customer's real needs, and how to negotiate a suitable scope so that a satisfactory system can be delivered within the agreed constraints. Expectation management is also important – the requirements processes help to set the customer's expectations for what will be delivered, and hence have an impact on customer satisfaction. The challenges involved in balancing these concerns become particularly difficult as projects increase in scale and complexity.

Through the 1990's, as software technology became cheaper and more widely available, other types of project increased in prominence, especially market-based, opportunity-driven, and community-based projects. In such projects, the absence of a specific customer with a well-defined problem adds significant new challenges for the requirements analyst. Such projects tend to be smaller in scope than customer-specific projects, but the problems are complex because they involve more stakeholders, with greater diversity, and greater potential for conflicting needs.

### 3.2.3. The existing system

Requirements engineering rarely starts with a blank slate – there is always an existing system that can be studied for insights into how the new system should work. An analysis of the existing system can help to provide a better understanding of the human activities that the new system should support, because those activities can be observed as they are currently carried out. Some people argue that too much concentration on the existing system may stifle innovation; however a much bigger danger is that an innovative new system may be not be acceptable to the users of the old system, especially if it is too different, fails to address their most pressing problems, or discards whatever they liked about the old system.

In some cases, the need to study the existing system is obvious. For example if the project is to upgrade or enhance a system, then the project will be constrained by the existing design; some functions will be easy to add, others may be hard, and this may determine which new features are selected for implementation. If the project is to integrate several existing systems, then the requirements will be derived almost entirely from an analysis of the systems that are to be integrated.

If the project is intended to replace a system that is now considered obsolete, then analysis of the existing system is still important. Such analysis can help to avoid the weaknesses of the current system, and can identify opportunities for improvements. Although a project may have been initiated with a clear idea of what was wrong with the old system, the requirements analyst must still investigate the old system. Different users may disagree about what is good or bad about the old system, and there may be features of the old system that nobody realizes are important until they are removed.

Finally, some projects are intended to introduce software-intensive technology where none has been used before. In such cases it may seem like there is no existing system to study. However, using the human-centered perspective we described in chapter 1, it should be clear that there is still an existing human activity system to study, as people carry out some existing functions manually, or develop coping strategies to deal with the limitations of an entirely manual operation. A systematic analysis of these activities may reveal places where automation would be a mistake, and other places where a software system could make a dramatic improvement.

### *3.2.4. Pre-existing components*

As well as being constrained by the existing system, engineering projects are also affected by the availability of suitable components from which to construct a solution. In many cases, using commonly available components can dramatically reduce the price of a system, because they do not need to be custom-built. However, there may be a trade-off between the functionality of the available components and the needs of the current project.

A consideration of components during requirements engineering is important because it can help in finding a good problem decomposition. Decomposing complex problems into simpler ones is an important activity in RE. Such decomposition is hard to do in a purely top-down manner because, as Jackson points out: "it enforces the riskiest possible ordering of decisions. The largest decision is the dubdivision of whole problem: it is the largest in scale and the largest in its consequences. Yet this decision is taken first, when nothing is yet known and everything remains to be discovered"[1]. If a potential decomposition yields some subproblems that already have readily available solutions in the form of off-the-shelf components, this may reduce the risk. Hence, knowledge of available components can play an important role in guiding problem decomposition, thus permitting a mix of top-down and bottom-up strategies. Naturally, availability of existing components cannot be the only guide for this, because of the danger of ignoring aspects of the problem for which there are no existing components.

One issue that demands particular attention in RE is the mismatch between what an existing component offers and the actual requirements of the current problem. The requirements analyst needs to examine the trade-offs, and consider whether any compromises will be made. For example, there is a danger that some requirements will be ignored if they cannot be met with existing components. The benefits of using a pre-existing component may be offset by the difficulty of integrating it into the broader system. And the stakeholders' views of the problem may be distorted by knowledge of available components. Such distortion can lead to a system that is easier to build, but will not address the real problem. A requirements analyst must perform enough analysis to ensure that the stakeholders can make informed decisions about the use of pre-existing components.

[1] From Jackson's lexicon of practice, principles and prejudices book.

### 3.2.5. Product families and shared architectures

If the problem to be solved is similar to problems that other customers might experience, there is an opportunity to exploit this commonality to reduce costs. Software re-use has often been proposed a way of realizing such cost reductions: program code from an existing system is used as the basis for components of the new system. However, software re-use suffers from the problems we described in the previous section: in essence, the cost of adapting re-used software to the new problem may outweigh the benefits. An alternative is to design a coordinated *product family* (also known as *a product line)*.

In a product family, a set of software-intensive systems are designed to share the same overall architecture, with certain kinds of flexibility built into the design from the outset. Each member of the family has features that satisfy the specific needs of particular types of customer, while sharing the same overall design.

Example product families include vertical lines, in which customers can pay more for additional features, such as desktop applications available in 'basic', 'deluxe' and 'pro' versions. They also include horizontal families, where similar software systems are needed in different domains, such as control systems for similar engines in cars, trucks, boats, etc.

The design of a product family involves a mix of technical decisions and business decisions. The key technical decisions surround the choice of an architecture that is robust enough to be used across the family, while permitting the flexibility for particular specialized needs. The key business decisions involve deciding the scope of the family, and the variations within it, based on the likely market for each family member. The broader the family, the harder it is to find a suitable architecture that will support the expected variants. Hence, an important aspect of product families is the decision of what *not* to include. A company considering developing a product line needs to make decisions about its core business, and which of its (potential) customers' needs can be included in a coherent family of software-intensive systems.

Many of these decisions are requirements engineering issues. Understanding both what is common and what is different about the different needs across the product family is a requirements engineering problem, as is dealing with the trade-off between business goals and technical feasibility of the proposed family. Requirements engineering for a product family proceeds a little differently from other types of project, because of this need to balance the similarities and differences between the members of the family.

## 3.3. Engineering lifecycle models

We now turn to the shape of the project itself. In this section we will consider a range of lifecycle models, which provide a general overview of the key stages in the life of a software-intensive system development project. As we suggested in the introduction to this chapter, lifecycle models are useful for comparing different species of project, but are insufficient for managing a project. The lifecycle models do not describe the conditions under which one phase of a project stops and another starts. Nor do they specify what types of activity should (or should not) occur in each phase. This is appropriate: every organization and every project is different, and must be managed according to local practices. Some organizations capture detailed guidance for project managers by describing *process models* for their projects. We will address these in the next section. In this section, we merely compare some general species of project, and the role of requirements engineering in each species.

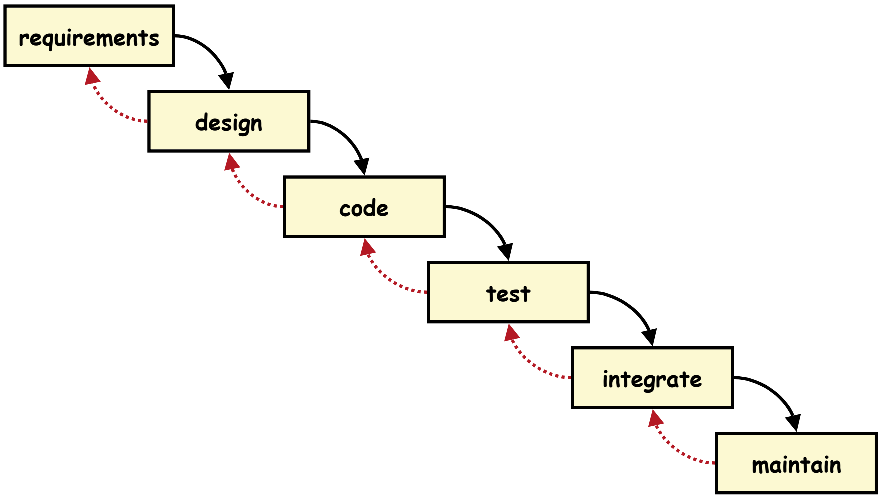

Figure 1: The Waterfall Lifecycle Model

### 3.3.1. Sequential Lifecycle Models

The waterfall model (see figure 1) is the simplest and best known lifecycle model. It divides a project into a sequence of phases based on the idea of stepwise refinement – requirements are first specified, then refined to produce a high-level design, which is then further refined to produce the detailed design (the program code). Once code is produced it is tested, and when it has passed all the tests, the system is integrated and finally delivered to the customer. At each stage, it may be necessary to revisit earlier stages, because of unexpected problems – for example, when missing or incorrect requirements are only discovered during the design phase, etc.

Early software engineering textbooks presented the waterfall model as an overview of how software development *should* proceed. Later textbooks acknowledged the weaknesses of the waterfall model, and presented some alternatives. However, the waterfall model is so deeply entrenched that many textbooks and courses still use it as a way of organizing the material to be presented.

The waterfall model is a very simplistic ideal, and only makes sense under certain very restrictive assumptions. The key assumption is that the requirements *can* be described at the beginning of the project, and then frozen for the remainder of the project. As we saw in chapter 1, this is unrealistic for most types of software-intensive system – the requirements evolve continuously, and uncertainties about the technical feasibility of meeting customers' needs means that requirements discovery is intertwined with design choices. This problem is compounded by the lack of analysis tools for software designs, so that major requirements and design errors may not be discovered until late in the test and integration phases, by which time the cost to correct them is significantly higher.

A variant of the waterfall model, known as the V model, is shown in figure 2. This model shows the conceptual relationship between the analysis/design phases (down the left-hand side of the V) and the corresponding verification steps (up the right-hand side). The verification steps ensure that each level of design is carefully checked in a sensible order – check each unit thoroughly before integrating it with other units, etc.

One interesting feature of the V model is that it separates time and level of abstraction as different dimensions. Because these were conflated on the waterfall model, progress on the project is confused with adding increasing detail. If we have to revisit the requirements during the design

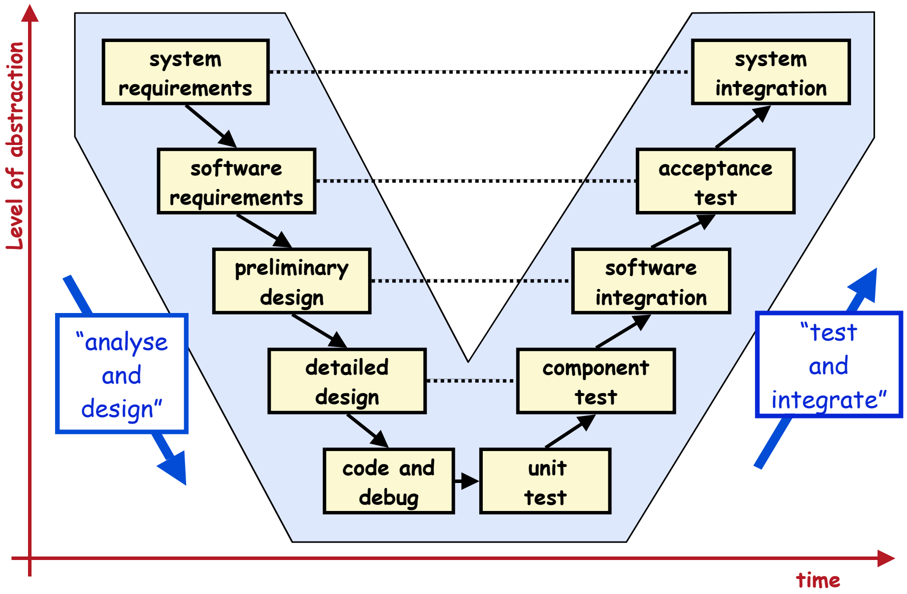

Figure 2: The V model.

phases, this looks like a backwards step on the waterfall model. In contrast, on the V model, vertical jumps make complete sense – for example during the requirements phase, it might be necessary to jump down to the detailed design phase to conduct a feasibility study on a particular technology; likewise, during detailed design, it might be necessary to jump up to the requirements level to revise part of the requirements. Such steps are moves only in the vertical dimension – they need not be seen as impeding the progress of the project.

### *3.3.2. Rapid Prototyping models*

Both the waterfall and V model assume the requirements can be pinned down almost entirely before any code is written. An obvious way to weaken this assumption is to allow for prototyping. Prototyping can be used to explore the requirements with the stakeholders, to evaluate designs of how the system will interact with users, to examine feasibility of a particular design approach, or merely to improve the communication and understanding between developer and client.

Figure 3 shows the basic waterfall model modified to allow for one or more prototyping phases before the main development process begins. Note that a prototyping phase could be added to any of the lifecycle models in this section in a similar way. We will discuss the use of prototypes for

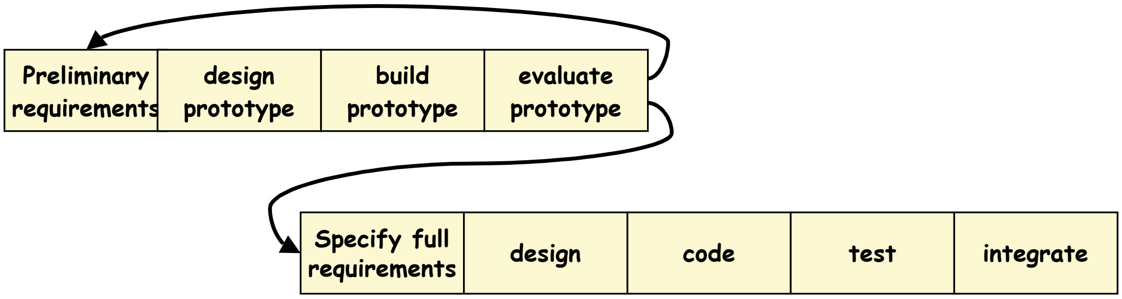

Figure 3: The prototyping cycle

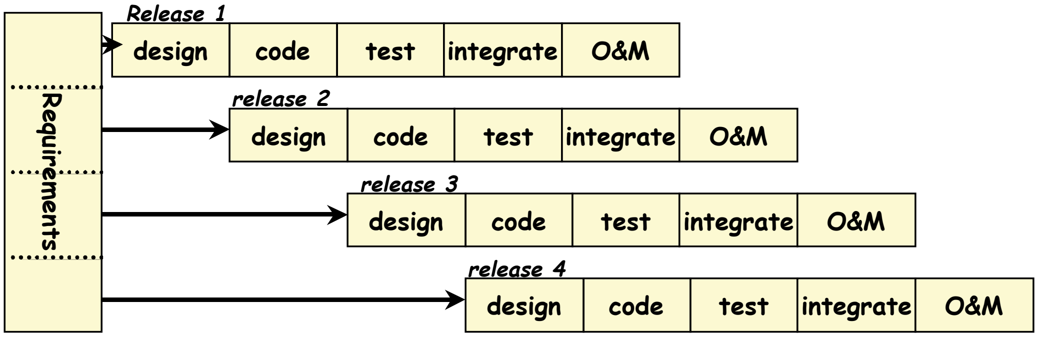

Figure 4: Incremental development. Initial requirements are allocated to a planned series of releases.

requirements engineering in more detail in chapter 16. For now, we will just note two problems often associated with prototyping:

- Customers (and even developers!) may fail to understand the difference between a prototype and a production quality system. Unlike prototypes for physical systems (a prototype car for example), these differences are invisible. The result is that a prototype that was intended as a rough mock up of a particular conception of the system often evolves into the delivered system, without proper attention to good design. The lack of principled design results in a system that is hard to understand, hard to maintain, and inflexible in the face of changing requirements.
- A prototype typically only covers some aspects of the requirements, most typically those related to how the user interface will work. This means that undue attention on the prototyping process can mean other requirements are ignored or forgotten.

In chapter 16 we examine some of the ways in which prototyping can be used as part of a requirements process.

### 3.3.3. Phased models

Another problem with the waterfall model is that it ignores what happens after the system is delivered. In practice, most software systems undergo continuing defect correction and re-design after initial delivery, resulting in a series of releases. Here we present two variants of the waterfall model that account for these subsequent versions of a system.

Figure 4 shows an incremental development model. As with the waterfall model, an initial requirements engineering phase aims to discover all the requirements. These are then prioritized so that they can be allocated to one of a planned series of releases, each of which meets more of the requirements than the previous one. This model avoids the one-shot effect of the waterfall model,

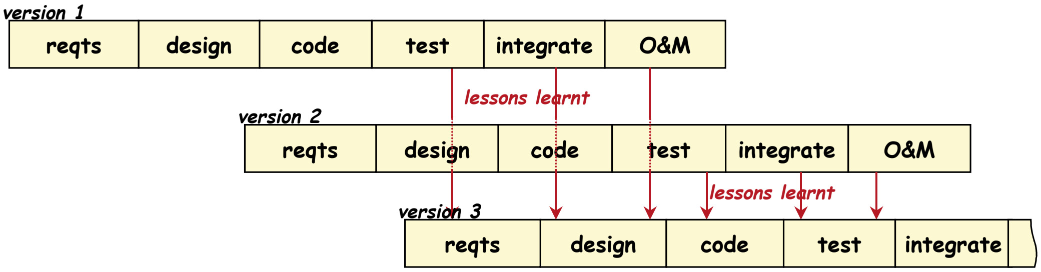

Figure 5: Evolutionary development. Each version incorporates lessons learned from earlier versions.

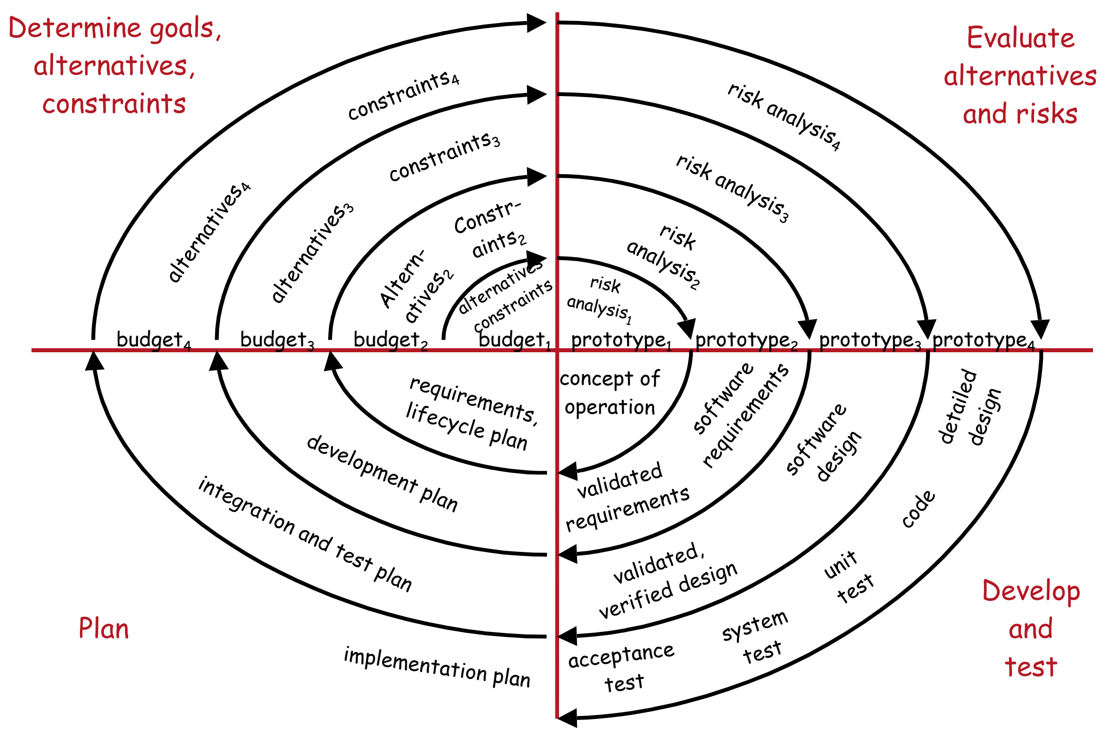

Figure 6: The spiral model.

but still suffers from the assumption that all the requirements can be known at the beginning of the project, and will not change subsequently.

Figure 5 shows an evolutionary development lifecycle. Here, instead of attempting to identify the requirements for all the releases initially, just enough requirements analysis is performed to permit development of a first version of the system. Each subsequent version then begins with another requirements phase, in which experience with earlier versions and changing needs can be taken into account. The difficulty with this model is that it is harder to plan the versions, and hence correspondingly harder to decide how to scope the requirements for each version. Also, if the development phases for each phase overlap, the lessons from the one version may be learnt too late to be incorporated into the next version.

### *3.3.4. Iterative Models*

The phased models recognize that most software-intensive systems are developed via a series of releases. This allows the developer to deliver some functionality to the customer early, and to adapt future releases to changes in the requirements. However, these models only allow for a very limited evolution of requirements from one release to another. For some projects, the problem complexity is so high that the only way to understand the requirements is through an iterative, exploratory process.

The spiral model, shown in figure 6, attempts to capture this iterative approach to development. In this model, each iteration is carefully planned, to include a sequence of planning, determining objectives and constraints, evaluating alternatives, resolving risks (including an explicit prototyping step), and development. The size of the spiral at each iteration is intended to convey an accumulation of knowledge and experience.

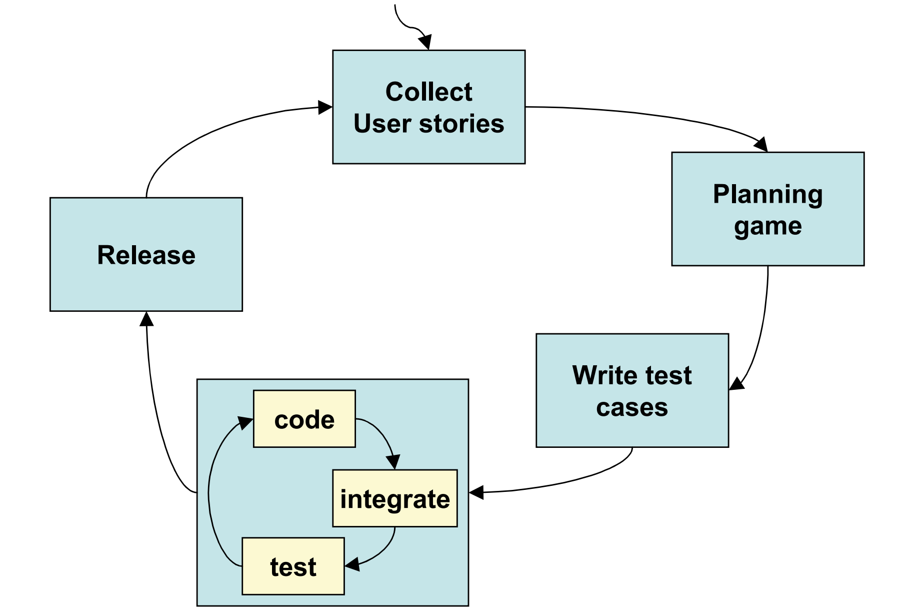

Figure 7: The XP lifecycle

There are several variants of the spiral model available in the literature. The version shown in figure 6 still includes some remnants of the waterfall model, because each iteration is associated with a major phase in the original waterfall model. The difference, of course, is that now each waterfall phase is embedded in an explicit risk reduction process, allowing re-planning and re-focusing between phases. However, the explicit requirements phase shown in the second loop of the spiral now represents only a small fragment of requirements activities, concerned with writing and validating specifications. Many aspects of the planning, evaluation, risk analysis and prototyping on each iteration are also requirements engineering activities.

### *3.3.5. Agile Models*

The lifecycle models in the previous sections are based on an ability to distinguish identifiable phases in the development of a software system. The phases are typically distinguished according to the documentation produced. Indeed, the waterfall lifecycle really only applies to a species of project found in large government agencies, especially defense, in which the end of each phase is marked by the production and approval of a large document (e.g. a requirements specification at the end of the requirements phase, a design spec at the end of the high level design phase, etc).

Agile development is a contrast to this document-heavy approach. Agile methods are based on the premise that too much reliance on documentation is a bad thing. The documents are expensive to produce, are of limited use, and lead to a very bureaucratic management style. In agile development, this documentation is replaced with direct interaction with customers. A key argument is that this allows the development team to respond much better to customer need, rather than relying on written contracts.

Probably the best-known agile method is XP – eXtreme Programming. XP includes a number of practices that are suitable for small project teams (typically no more than a dozen programmers). The XP lifecycle is shown in figure 7. An XP project is based around frequent releases, perhaps as often as every three weeks. Each release cycle begins by collecting ‘user stories’, each expressed on a notecard, and then using a ‘planning game’ in which a set of user stories are selected for

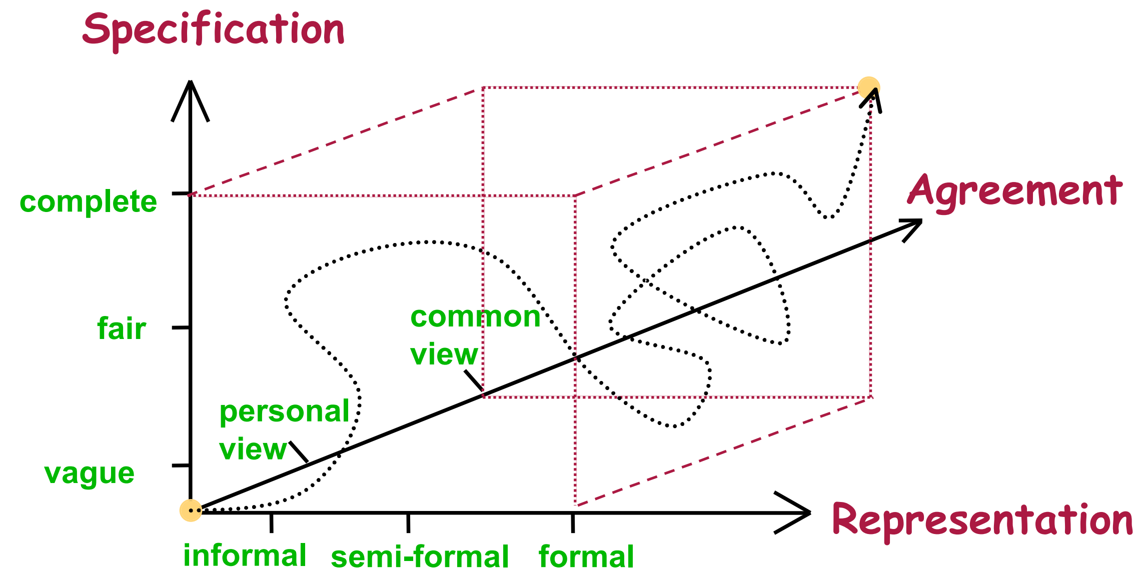

Figure 8: Pohl's 3 dimensions of requirements engineering

implementation. Test cases are written before the program code, and programming itself is conducted with a daily integration and test schedule. Programming is carried out in pairs, with the assumption that paired programmers produce higher quality programs than single programmers.

In agile development, the requirements engineering activities are not limited to a particular phase, but are carried out continuously. Mistakes are likely, but each mistake is treated as an opportunity for learning. An on-site customer representative provides continuous feedback, to guide this learning process. Above all, agile development relies on people rather than documents to convey an understanding of the requirements. Clearly, this type of project is appropriate for some types of system (e.g. small projects, where there is a great deal of uncertainly about the requirements), and is inappropriate for others (e.g. large safety-critical systems).

### *3.3.6. Is there a Requirements Lifecycle?*

We have surveyed a number of lifecycle models, and indicated the role of requirements engineering in each. In the past, RE methods often assumed that requirements engineering was performed for a specific customer, in a particular phase of the lifecycle, and that the customer could sign off a requirements specification. In fact, requirements engineering is performed in a variety of contexts, including projects for which no customer is initially identifiable. Because of this huge variety of projects, there is no single correct requirements engineering process. Many books on requirements engineering would have you believe otherwise: they describe one particular way to do requirements engineering, and assume it will work for all projects. In this book we describe the fundamental activities involved in requirements engineering. The decision about which of these activities will be needed, when they will be applied, and how they will be managed depends entirely on the type of project.

Because of this variety, there are a wide variety of terms used to refer to requirements, often with differing connotations. Some approaches, most notably the agile methods, refuse to use the term requirements at all, preferring instead to talk of scenarios, vision, metaphors, user stories, etc. However, a basic set of requirements activities is common to all projects. Every project needs to

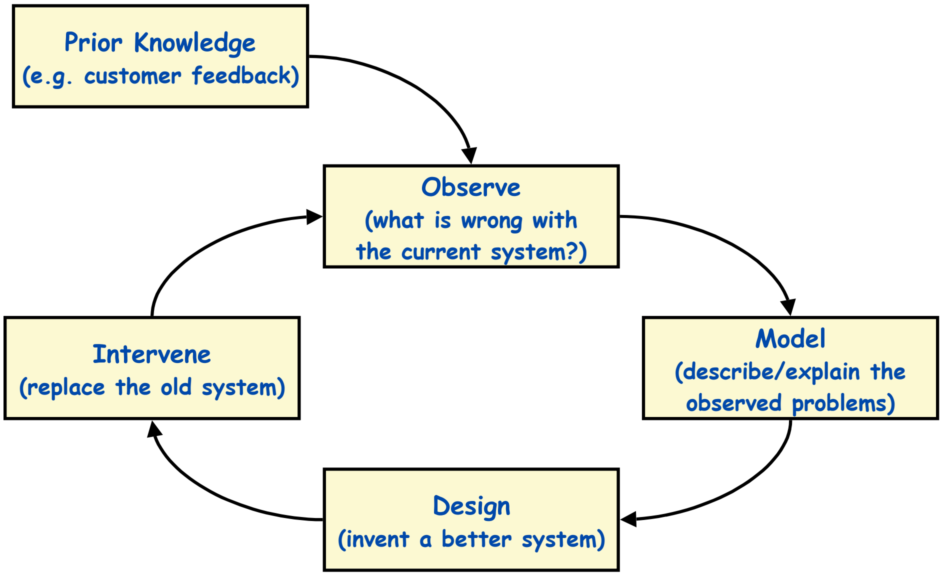

Figure 9: The inquiry cycle

identify and scope the problem, decompose it into manageable pieces, and manage the changes as the problem (and our understanding of it) evolves.

So, even if we cannot describe a single process model for requirements engineering, can we still identify lifecycle models for the *requirements themselves* (as opposed to the project in which they are embedded)? One of the simplest characterizations of a requirements lifecycle is the distinction between *early* and *late* requirements, first introduced by Mylopoulos and colleagues. Early requirements are concerned with understanding the problem context: modelling the organization in which the problem exists, and the goals of and dependencies between the various stakeholders. Late requirements are concerned with pinning down the desired functions of the system-to-be-built.

Another view of the requirements lifecycle is the model described by Pohl, as shown in figure 8. Pohl identifies three dimensions over which a requirement ranges during its life:

- Specification – initially each requirement is likely to be vague, but the eventual goal is to attain a complete specification of it.
- Agreement – initially each requirement is likely to be just a personal view of a single stakeholder, but eventually the aim is to reach agreement among all stakeholders
- Formality – initially each requirement is likely to be stated informally, perhaps in sketches and words, but the eventual goal is a precise formal statement of the requirement.

The trajectory of each requirement along these three dimensions will vary; figure 8 shows one possible path. The destination in the upper right hand corner is just an ideal. It might not be reached by all requirements, because attaining complete specification, agreement and formality may be too expensive.

By contrast, the inquiry cycle model shown in figure 9 equates the requirements with a *theory* about the nature of the problem to be solved. This model is based on (an idealized version of) scientific investigation: scientists develop theories to explain observed events, and then design and conduct experiments to test their theories. In this model, requirements engineering covers the first three boxes. Using both prior knowledge and observation of the current system, the requirements analyst builds models that represent the best current theory of what the problem is that needs solving. Based on this theory, a software-intensive system can be designed and installed, to test the theory. Observations of this new system in use then become the first step in the next iteration of the cycle: the requirements models are adjusted to capture what has been learnt about the problem, and the next version of the system can be designed and installed.

## 3.4. Improving the Engineering Process

There are two key outputs from any engineering project: the engineered artifact itself, and the knowledge and experience gained in producing it. The latter is often neglected, but for many projects, it may be the most important output – this is certainly true of projects that fail! The problem is that much of the knowledge and experience gets dissipated when a team disperses at the end of a project. Some project team members may remember and apply the lessons. However, individuals may not be *able* to apply these lessons, especially if they require major changes to the organizational culture. Some organizations produce a "lessons learnt" document as part of a post mortem at the end of each project. Other organizations have found that it is better to institutionalize the capture and application of lessons learnt through the use of *process improvement* techniques.

### *3.4.1. Quality Control*

The ideas of process improvement can be traced back to the early attempts to improve the quality of manufacturing processes in the latter half of the twentieth century, particularly in the car industry in the US and Japan.

Early approaches to quality control on production lines were based on inspection of intermediate and final products, so that defective products could be discarded. This approach led to various experiments with process control, whereby various control parameters on the production line were adjusted in response to observed defects in the products. For example, if some car body panels are found not to have a good coverage of paint, one could increase the length of the spraying.

Unfortunately, this approach tends to make things worse. This is because in general, on most production lines only a very small proportion of products are defective, typically less than 5%. If you adjust various control parameters to attempt to eliminate these defective products, you have to adjust them for *all* the products, even the non-defective ones. Hence, there is a huge risk that other defects will be introduced for the 95% of products that were previously okay. For example, increasing the length of paint spraying for *all* the body panels may introduce problems of paint runs and poor drying for panels that would otherwise have been fine.

Eventually, these observations led to the use of statistical techniques to analyze the production processes and identify the *causes* of defects. This analysis was then used to *redesign* the production process to eliminate the causes of defects, rather than just adjusting control parameters. To continue the body panels example, the defect data could be used to analyse the paint spraying process, with the aim of redesigning how paint is applied, to eliminate defects. The use of statistical analysis of defects is crucial, as it serves to pinpoint changes that will offer the greatest improvement on overall quality. The approach was pioneered by Deming, and applied with great success first in the Japanese car industry in the 1970's, and eventually in the US car industry in the 1980's.

### *3.4.2. Software Process Maturity*

The success of quality management ideas for improving quality in the car industry led several people to question whether the same ideas could be applied to software production. Because software is not manufactured, there is no *manufacturing* process to improve. However, the idea can also be applied to the software *development* process. Unfortunately, software development processes are poorly understood, at least in comparison with industrial manufacturing processes.

The challenge then, was to get organizations to document their development processes carefully, use statistical measurement techniques to identify places in these processes where the most design defects occur, and to use this information to re-design the process itself. The payoff is potentially large, because instead of just fixing defects in individual products, it may be possible to prevent the defect for all subsequent products, by redesigning the development step that causes the defect in the first place.

These ideas form the centerpiece of the Capability Maturity Model (CMM), developed by Watts Humphrey and colleagues at the Software Engineering Institute in Pittsburgh. The CMM is an assessment tool, used to determine how mature a particular company's software development processes are, by assessing how well that company measures and improves them on a routine basis. The CMM places software development companies at one of five levels:

- Initial – in which development is entirely ad hoc;
- Repeatable – in which the same development process is used in different projects, but such repetition is dependent on individuals;
- Defined – in which the development process is documented, and institutionalized;
- Managed – in which measurement techniques are used to quantify the process so that improvements can be made; and
- Optimizing – in which improvements are continually fed back into the process.

The CMM has led to significant quality improvements in some parts of the software industry, particularly those involved with large, safety-critical systems. Versions of the CMM have also been adapted for use in systems engineering, and a number of other engineering disciplines. The same ideas are also captured in the ISO9000 series of international standards, which are applicable across a wide range of industries.

One of the key ideas underlying the CMM is that each organization must understand its own processes, and a process that works for one organization may not work for another. Hence, neither the CMM nor the ISO9000 standards prescribe any particular development process. Instead, they insist that each organization should be aware of what process it is using, and should work to continually improve it. Most importantly, assessment and comparison between organizations is with respect to their relative *maturity* in understanding and managing their processes, and does not involve any comparison of their actual process models. In fact, comparison of the process models themselves would be counter-productive, as it may work against the goal of these models being an honest appraisal of actual development processes.

Finally, we should note that although there is empirical evidence that initiatives such as the CMM have had a positive impact on some of the organizations that have applied them, there has also been a backlash within some parts of the software industry, in response to the demand for a process-heavy approach. The value of documenting the development process, and then managing to this defined process is only of use if the various projects conducted by a company are sufficiently similar to one another for the same process to apply repeatedly. For some companies, this is clearly not true. In particular, small companies involved in very innovative projects may find that very few aspects of their development process carry over from one project to the next. For example, Agile development methods projects do not document development processes, and do not use process models as a way of managing the project. Instead, they rely on highly skilled individuals, and good quality direct communication between the customer and the development team to keep the project on track. However, such methods only really work well for small dynamic project teams.

### 3.4.3. Lifecycles vs. Process models

Earlier in this chapter, we described a number of *lifecycle models* for development of software. Lifecycle models are quite different in purpose and scope from the *process models* we have described above. Process models are detailed descriptions of the step-by-step development of a system, and include information about scheduling and resources for each step, along with dependencies between steps. They are intended as a tool for managing projects, and as a structure for statistical analysis of defects, used for process improvement. Lifecycle models, on the other hand, are very generalized descriptions of the phases that a project goes through, and are intended for understanding and comparing different species of project. The analogy with the lifecycle models used in biology is helpful – a description of the lifecycle of a butterfly is useful for getting the overall understanding, and for comparison with lifecycles of other types of insect. However, it doesn't tell you how exactly a butterfly manages to mutate from one stage to the next, nor does it contain any data on how various factors (temperature, food supply, predator density, etc) will impact the development stages of a particular butterfly.

Lifecycle models are so abstract that they are almost useless for management purposes. However, they are a useful pedagogical tool – in this book, we use them to help us understand the role of requirements engineering in different types of project.

## 3.5. Chapter Summary

TBD

## 3.6. Further Reading

**Nature of Engineering:** For interesting thoughts on the differences between engineering and science, read Walter Vincenti's book "What Engineers Know and How they Know it". The distinction between normal and radical design is also covered in this book.

**Nature of Software Engineering**: For thoughts on software engineering as a discipline, read Mary Shaw's paper "Prospects for An Engineering Discipline of Software", which appeared in IEEE Computer in Nov 1990. There is also an excellent set of papers in the Annals of Software Engineering, vol 6, no1-2, 1998, ranging from what it means to call software engineering an engineering discipline, through to specific ideas about how it should be taught.

**On whether software is different:** The classic paper is Fred Brooks' "No Silver Bullet" (IEEE Computer, April 1987), in which he discusses why software engineering is so hard. Tom Maibaum provides a more recent analysis in his paper "Mathematical Foundations of Software Engineering: A roadmap", which appeared in the IEEE Press 2000 volume "The Future of Software Engineering".

**Code of Ethics:** The IEEE/ACM code of ethics can be found online at http://www.computer.org/tab/seprof/code.htm The IEEE has also produced a "Software Engineering Body of Knowledge" as a first step towards identifying what software engineering professionals should know: see http://www.swebok.org/ Finally. IEEE Software of Nov/Dec 1999 has an interesting collection of paper on accreditation and licensing of software engineers.

**Ethical Dilemmas in Engineering:** The example ethical dilemmas in this chapter are adapted from examples at http://onlineethics.org/ and at from Kevin McBride's IS214 course at Berkeley: http://www.sims.berkeley.edu/courses/is214/ both sites have more examples.

**Project Management:** There are many books on project management. A good place to start for software engineering project management is Thayer and Dorfman's edited volume "Software Engineering Project Management (2nd Edition)", IEEE press, 1997. The suggestions for key measurements are adapted from Mah and Putnam's paper "Software By The Numbers: An Aerial View Of The Software Metrics Landscape" in American Programmer, Nov 1998.

**Product Families:** A good starting point for work on product families and how they related to requirements is Stuart Faulk's paper "Product-Line Requirements Specification: An Approach and Case Study", which appeared in RE'01.

**Project Initiation:** An interesting paper exploring how projects get started is Bergman and Mark's paper on project selection at NASA, "In Situ Requirements Analysis: A Deeper Examination of the Relationship between Requirements Determination and Project Selection", which appeared in RE'03.

**Lifecycle models:** Any good textbook on Software Engineering has a summary of lifecycle models. For a discussion of the difference between lifecyle models and process models, see Walt Scacchi's entry "Process Models in Software Engineering" in the Encyclopedia of Software Engineering (2nd edition), 2001.

**Agile Models:** Recent books on Agile Methods and Extreme Programming are too numerous to mention. Kent Beck's "Extreme Programming Explained" is probably as good a place as any to start. For thoughts on Requirements Engineering and Agile Methods, read Ben Kovitz's paper "Hidden skills that support phased and agile requirements engineering" in the Requirements Engineering Journal, volume 8, 2003.

**Requirements Lifecycle:** The distinction between Early and Late Requirements was first introduced by John Mylopoulos – see for example the paper "Towards Requirements-Driven Software Development Methodology: The Tropos Project," by Castro, Kolp and Mylopoulos, in Information Systems, June 2002, The Klaus Pohl's three dimension are described in his paper "The three dimensions of Requirements Engineering: a framework and its applications" in Information Systems vol 19, 1994. The inquiry cycle was first introduced by Colin Potts and colleagues in their paper "Inquiry-based Requirements Analysis", IEEE Software, November 1994.

**Capability Maturity Model:** Information about the capability maturity model can be found on the SEI website at http://www.sei.cmu.edu/cmm/ An empirical study of the benefits of the CMM was conducted by Jim Herbsleb et al in 1994, and is available as report CMU/SEI-94-SR-013 on the SEI website.

## 3.7. Exercises

TBD