1 Introduction

Requirements are a ubiquitous part of our lives, so it may seem strange that they have been singled out for study in computer science. Requirements and communication are inextricably intertwined. We start making our requirements clear soon after we are born by crying, usually for food. Parents soon become expert requirements engineers in inferring what their children want; but even in the cradle the dilemma of requirements is exposed. A baby's cry is ambiguous. Does he or she want food, warmth or a cuddle? How do we translate our interpretation into the right food, degree of warmth, or appropriate rocking motion?

The requirements–communication problem stays with us throughout our lives as we struggle to make our needs known to others. In professional life, requirements are more closely bound to design. Much human endeavour is directed towards creating things, be they goods or services. To create, you might be lucky and have clairvoyant inspiration; for most of us, it is better to start with a more prosaic definition of what the customer wants. All branches of engineering and manufacture have experienced the requirements problem since the industrial revolution. Creation, design and requirements have become competitive siblings. In some cases, design and creativity led requirements to the discovery of new possibilities, such as when Abraham Darby built the first iron bridge across the River Severn in 1796. Later requirements for cheaper bridges drove the designer of the Tay railway bridge to design an unsafe structure that led to the disastrous collapse of the bridge as a train crossed it during a fierce storm. So if requirements have always been with us, why has computer science discovered the need to make a special study of this phenomenon? The answer lies in complexity and the nature of software. Computer systems apply to nearly every walk of life. They appear in domains ranging from engineering to business, education and leisure. That gives computer scientists a particular problem. They have to become knowledgeable in many different domains that are not their immediate areas of expertise; hence, requirements is a special problem because of the ubiquitous nature of software. Requirements engineering is the area of computer science that addresses this concern, although it cannot claim sole ownership. Systems engineering, a product of other design disciplines, also encounters the requirements problem. I shall return to the wider perspective in the final chapter.

1.1 Motivation for Requirements Engineering

There are many definitions of requirements engineering (RE); however, they all share the idea that *requirements* involves finding out what people want from a com-

puter system, and understanding what their needs mean in terms of design. RE is closely related to software engineering, which focuses more on the process of designing the system that users want. Perhaps the most concise summary comes from Barry Boehm: requirements are "designing the right thing" as opposed to software engineering, which is "designing the thing right" (Boehm, 1981).

Requirements have always been with us in any human act of design; for instance, the *Titanic* disaster can be construed as poor requirements analysis, in that the shipbuilder and ship owner failed to set out the precise specification of what "unsinkability" meant. The *Titanic* was correctly built to the requirements that specified that the hull be subdivided into watertight compartments, with electrically operated doors between the bulkheads that could be closed from the bridge. The owners and shipbuilders thought this equated to unsinkability, because up to three contiguous compartments could be flooded and the ship would not sink. Unfortunately they never thought that an accident would flood four contiguous compartments, which is what happened as a consequence of the glancing blow from the iceberg collision. The rest is history. The tragedy also uncovered further mistaken requirements such as the false assumption that lifeboats sufficient for all the passengers and crew would not be necessary. This assumption saved money during construction, but with terrible consequences.

We have not improved our practice much since 1910, and mistakes in RE still cause problems. Modern aircraft are controlled by computers. An A320 airbus had just taken off from Gatwick airport when the pilot discovered the aircraft making an unexpected turn to the left. The flight crew struggled to control the aircraft and thought that the aerilons (flight controls on the wings, which alter the angle of banking as an aircraft turns) were not working correctly, over-compensating for slight changes in direction. They eventually managed to land the aircraft correctly having struggled with the emergency flight management procedures. These proved to be impossible to find via the menu hierarchy in their cockpit VDU displays but the pilot, luckily, had an old paper manual. The safe return of the aircraft was due to the skill of the flight crew rather than the system design, which did not diagnose the problem in the first place and which then hindered their attempts to correct the failure.

The cause of the near accident was that a maintenance crew had been servicing the hydraulics that control the wing slats. The wing slats automatically open and close to equalize pressure on each wing when the aircraft turns. The hydraulics can be disabled by placing them into service mode, so that damage by manual manipulation of the slats is avoided. When the hydraulics are disabled the slats open and close with small changes in air pressure leading to instability. Unfortunately, the maintenance crew left the hydraulics in maintenance mode so the slats did not work on take-off, and nobody noticed beforehand. The maintenance crew were new, poorly trained and under time pressure to complete a service on schedule. The management culture of pressure and inadequate resources led to the window of opportunity for disaster. The designer had never thought that a requirement for a warning light showing that the hydraulics were in maintenance mode would ever be needed; neither did the pre-flight checklist consider this to be a hazard worth checking.

Computer systems have made the requirements problem worse because we build systems in many different domains. The requirements engineer (or designer) has to understand not only what the user wants, but also the implications of the domain

and what is achievable. Since gathering requirements inevitably involves communication between people, and natural language is prone to misinterpretation, requirements analysis has been a frequent cause of system failure.

The failure to undertake a thorough requirements analysis is illustrated by many well-publicized computer system disaster stories. One of the most prominent in recent years was the London Ambulance Service's Computer Aided Dispatch System (Finkelstein and Dowell, 1996; HMSO, 1993). This was intended to replace the manual system of answering emergency telephone calls from members of the public, finding out where the emergency was and then dispatching one of the available ambulances to the location of the accident. Superficially this was a simple resource allocation problem. While nearly every possible mistake in software development and project management was made in this case, poor RE played its part. The software developer was a small systems house with little experience in the domain of ambulance call-dispatch systems. The systems analysts, nominally in charge of requirements, were the in-house software development team of the London Regional Health Authority. The users were the operators in the call-dispatch centres whose job it was to allocate ambulances to emergency calls, and the ambulance crews themselves. Little, if any, requirements analysis was carried out with these real users. The Regional Health Authority development team specified the requirements. There was no consultation or opportunity for feedback until the system was ready for deployment. At that stage, performance problems were so bad that just getting the system to work was an uphill struggle. Poor requirements analysis failed to detect several problems: radio blackspots where the ambulance crews could not be contacted, poor user interfaces on the mobile data terminals which resulted in the ambulance crews not reporting call progress accurately, with the knock-on effect that the system database became inaccurate, causing the automatic call allocation program to lose ambulances, dispatch ambulance crews who were not free, send several ambulances to the same call, and so on.

The London Ambulance Service is not alone in the litany of RE failures. Government departments and the armed services in particular have a long track record of getting it wrong. The UK social services tried to computerize the claims payment systems in the 1980s, but had to cancel the whole development because the requirements could never be stabilized. The updated UK air traffic control system was supposed to have been implemented in 1998 but at the time of writing has yet to go live, partly because the requirements for the complex 3-D visual displays took so long to finalize. A similar problem has emerged with Eurocontrol (Europe-wide air traffic control) where requirements for flight co-ordination have caused endless problems. In the USA the Pentagon's systems have also had their share of requirements disasters, ranging from Patriot missile radars that could not see a Scud warhead among missile debris (missed requirement that Scud missiles were so badly engineered they fell apart on the way down) to logistics systems that were never implemented (Potts, 1999).

So the penalty of getting RE wrong is high. Systems may fail, but even if they do not, their use is sub-optimal or design costs are wasted. Studies of banking systems showed that users only used 30 per cent of the functions provided (Eason, 1988). There is plenty of evidence that RE is a difficult task. Stories of spectacular system failures (e.g. the London Stock Exchange Taurus system and the London Ambulance call-dispatch system outlined above) point to poor requirements capture or even failure to make any serious attempt to capture user requirements. So is it just a

problem of motivation and education? In spite of several well-publicized disasters and an invisible legacy of system failures, many systems have been developed successfully, and most companies now depend on computerized accounts, payroll, and stock control systems. The key to the problem is change. If organizations kept to the same practices and the world stayed the same, then requirements would be easier; but they don't. Managers want to improve business practices, governments change tax laws, businesses have to respond to challenges from competitors. Consequently, requirements can never be said to be complete and computer system designers are shooting at a moving target. This problem has been elaborated by Lehman and Ramil (2000), who point out that most software has to be designed to evolve.

The motivation for RE is simple: to reduce the high cost of misunderstanding between users and designers, so that computer systems are built to do what the users want, on time and at a reasonable cost. The high cost of errors incurred during many system failures can be attributed to mistakes in requirements analysis (Bell and Thayer, 1976). Although RE only amounts to 10–15 per cent of the overall cost of system development, the consequences of getting requirements wrong can have a disproportionately high impact. The longer a mistake remains undetected during development the more expensive it is to fix. Changing a requirement just takes a bit of word-processing, changing a design involves undoing the specifications and checking ramifications of new requirements, but changing code means throwing away programmers' time and possibly destabilizing a design. The economic arguments for RE are summarized in Figure 1.1.

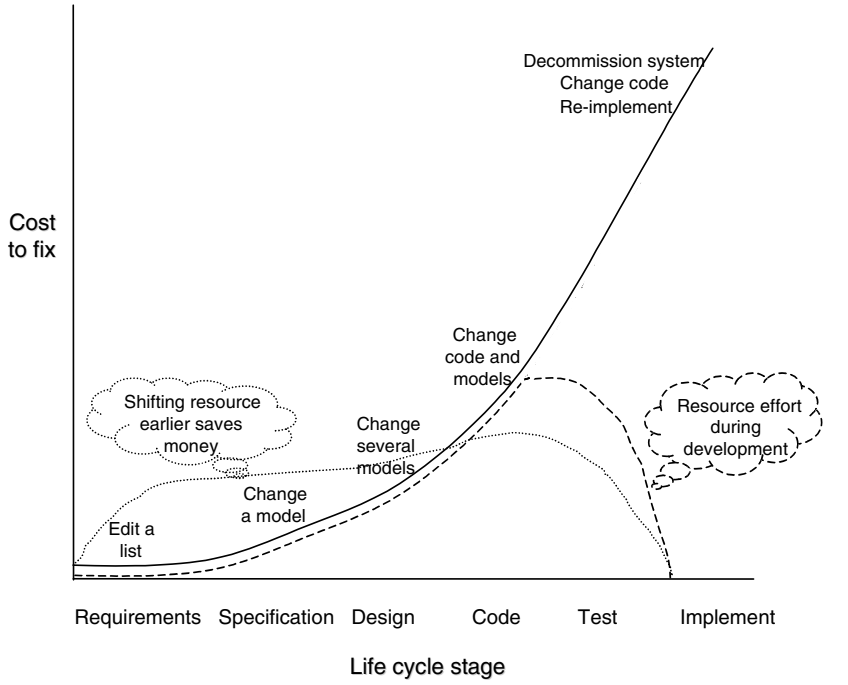

Fig. 1.1 Costs of change and the development cycle.

Investing in RE early will save money later on, but it can be difficult to decide when requirements analysis should stop. First there is the "fickle user problem": people tend to discover new requirements just when the analyst thought everything they wanted had been captured. Secondly, requirements analysis is difficult because of the "moving world problem". All the analyst can do is to get the best possible picture of the world when capturing requirements, try to anticipate the future, and then design software so that it is flexible and can adapt to change. While methods and technology can help with requirements analysis and flexible design, nobody possesses 20/20 foresight, so anticipating the future is a black art. As we shall see later, however, there are ways in which designers can frame their ideas about the future.

1.2 A Little History

Before RE, requirements were subsumed in systems analysis. This area produced structured development methods such as SA/SD (structured analysis/structured design: De Marco, 1978); SADT (systems analysis and design technique: Ross and Schoman, 1977) and SSADM (structured systems analysis and design method: Downs, Clare and Coe, 1992). These methods all referred in passing to requirements analysis and generally started with a divide-and-conquer approach of carving a system into successively smaller pieces and then defining the functions or goals that each part of the system was supposed to achieve.

So why was systems analysis not good enough and a whole new sub-discipline of RE needed? Part of the answer is rooted in academic communities. Systems analysis belonged to information systems, a community concerned with methodology and conceptual modelling. It is probably true to say, however, that the concept of requirements analysis in information systems was suffering from an intellectual strait-jacket of top-down functional decomposition. Requirements were captured and listed as users' goals, then elaborated as a set of functions represented in data flow diagrams, SADT processes or whatever the analyst's favourite conceptual modelling language was. The "structured methods" approach to requirements analysis was challenged in the 1980s with JAD/RAD (joint/rapid applications development) techniques (DSDM, 1995) which tried to overcome the cumbersome and time-consuming process of modelling with the use of workshops, scenarios and brainstorming sessions to encourage more user involvement in design. The "top-down" analysis approach of structured methods was also challenged by the object-oriented community who proposed modelling the domain rather than establishing goals/functions.

This led to the current generation of structured object-oriented methods: object-oriented systems engineering (Jacobson *et al.*, 1992); the object modelling technique (Rumbaugh, 1991); and object oriented analysis/design (Coad and Yourdon, 1991), to cite but a few of the contenders. More recently the object-oriented community created a *de facto* development standard as UML (unified modelling language) and the unified process (Rational Corporation, 1999), but this has little to say about requirements analysis apart from advocating use cases and scenarios, which will be explained later.

The other influence on RE was software engineering, a community who focused on formal specification and delivery of reliable software products, often for real

time telecommunications and safety critical applications. Software engineers found that in spite of years of formal specification many systems were not accepted by users or failed in circumstances that had not been anticipated. RE has grown from, but also contributes to, all these areas. It is related to research in software engineering and conceptual modelling by sharing models and formal languages drawn from that research. RE is also concerned with process, although not always in the structured methods sense. RE research, however, grew from the need for techniques and tools that complement software engineering methods, so RE can be seen as a collection of techniques that are recruited to a more general process according to the users' and designers' needs.

The foundations of RE were set out in a collection of papers (Thayer and Dorfman, 1990) and special issues of *Transactions on Software Engineering* (IEEE-TSE, 1991, 1992). These were followed by IEEE symposia and conferences (Finkelstein and Fickas, 1993; Davies and Hsai, 1994). These events revealed a diversity of research issues and industrial practice that can be loosely associated with defining what to build rather than how. Lubars, Potts and Ritcher (1993), in one of the few investigations of RE practice in industry, reported that ambiguity and changing requirements were a constant problem and that developers preferred organizational to technological solutions for RE problems. More recently, field studies of system development practice (El Emam and Madhavji, 1995) have indicated that changing requirements, lack of trained manpower and inadequate methods are responsible for system failures. RE research has tended to be dominated by large customer-driven systems, typically in the defence sector. In these contexts, requirements are complex and often driven by a super designer's vision, whereas market-driven requirements arise out of a more creative brainstorming approach (Lubars *et al.*, 1993).

1.3 People, Communication and Requirements

Another reason why requirements analysis is hard is because of people. We all have our own ideas and viewpoints. Many are unconscious attitudes that we only become aware of when someone challenges our beliefs. Furthermore, we keep some facts and attitudes to ourselves for a variety of private and political reasons. The upshot of this is that getting a complete set of unadulterated facts from people is unlikely even with a set of co-operative, honest and well-motivated users. The practice of requirements analysis thus has to combat problems of human communication that involve psychology and sociology, such as tacit knowledge, the ambiguity of natural language, and the role of power and personality that influence attitudes and opinions.

- *Tacit knowledge* – as we learn any skill or become familiar with a domain, we no longer think about it in conscious terms. Consider driving. Unless you are a learner driver you don't think about changing gear or steering the car; these are automatic skills acquired years ago. Similarly, you don't think about your route to work; you just know it. So when someone comes to ask you about your experiences in getting to work, you are unlikely to tell them about the route in detail or how you drove your car. The same problem exists in all domains. Users are experts, so they don't tell the requirements analyst about the obvious facts, yet these are often important in understanding how that domain works. And this doesn't take into consideration the facts that people *want* to hide.

- *Ambiguity* – even if we try to report what we do, we frequently do so inaccurately. English and most other natural languages are wonderfully flexible means of expressing ideas, but the disadvantage is that expression is not always precise. A classic example is conjunction in logical procedures; when we say "and" we sometimes mean "both", sometimes "either". For instance, "I go to the bank and building society to get cash" – does this mean I always go to both the bank and the building society, or sometimes to just one of them? Ambiguity is inherent in logical operators, comparisons (greater, equal to, etc.), as well as imprecise expressions of procedures. Sometimes this is a consequence of poor expression, but it may also be a symptom of vague thinking, as we discover when following street directions from a well-meaning but uninformed local inhabitant.
- *Attitudes and opinions* – people tend to pick up attitudes and opinions from their family, peers and news media. When asked to justify an objective or goal, they may give an opinion that has never been thought through. Requirements analysts therefore have to challenge attitudes to find out which ones have a rational justification and which are a matter of folklore. When challenged, people may get into trench warfare based on dogma rather than arguments based on their merits. Furthermore, opinions are often influenced by internal politics and the power structure of organizations. Some employees, conscious of job insecurity or an authoritarian boss, may give false opinions to placate a "party line". Social factors, such as power and responsibility, influence the expression of requirements (Goguen, 1993). People often fear change itself. Requirements analysis exposes problems and hidden agendas in organizations. Fact gathering has to tackle problems of public versus private versions of truth which users may, or may not, wish to communicate (Harker, Eason and Dobson, 1993; Maiden and Rugg, 1994). Eliciting true attitudes is one of the most difficult tasks the requirements engineer has to tackle. Failure to detect political problems can lead to systems being implemented without the consent of all the stakeholders, or to problems remaining hidden until it is too late. Early detection of false attitudes alerts the analyst to potential conflicts and the political danger of getting trapped on one side or another. Reconciling viewpoints to get a consensus view is another difficulty. The analyst can find him or herself in the middle of an inter-departmental war, for example between marketing and customer services who have very different high-level goals and do not understand each other's point of view.

1.4 A Framework for RE

Several definitions of RE have been given. For instance in terms of the outcome, "a requirements specification should tell a designer everything he needs to know to satisfy all the stakeholders – but nothing more" (Sommerville, 1989). Alternatively, the principal RE issue described by Bubenko (1993) is "how to proceed from informal, fuzzy individual statements of requirements to a formal specification that is understood and agreed by all stakeholders". Dubois, Hagelstein and Rifaut (1989) used the term to refer to the part of the development life cycle in which the needs and requirements for the user community are investigated and then abstracted to create formal specifications. Zave (1995) proposed a more elaborate taxonomy of problems in requirements engineering that helps to scope the field. Her definition of RE was "the branch of software engineering concerned with real world goals for,

functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviours, and to their evolution over time and across software families". At the top level Zave divided RE in two dimensions. The first is the *problems* of requirements engineering, decomposed into: specifying system behaviour; problems of investigating the goals, functions and constraints of software systems; and problems of managing the evolution of systems and families of systems. The second dimension is RE *solutions*: the type of research in the field, such as state of practice reports, process-oriented solutions (methods), product-oriented (tools), case study application of solutions in the real world, evaluations of approaches and measurement of success. One of the tensions these definitions reveal is the focus of RE on software or on requirements for large-scale systems (including people), and how reuse fits into the picture.

Trying to state the objectives (or requirements) of RE reveals some of the dilemmas that need to be solved. For instance, the following objectives for RE may conflict:

- to capture a complete set of requirements from users;
- to analyze the users' requirements accurately and understand all the implications inherent in those requirements;
- to specify how those requirements should be met in a design;
- to complete requirements analysis within acceptable constraints of time and cost.

Given patience and infinite resources it may be theoretically possible to get a set of near-perfect requirements; however, within the resource and time constraints of the real world this is unlikely. The adjective "near-perfect" is used because requirements analysis faces another dilemma: the world keeps changing. When the analyst has captured a complete set of requirements they inevitably refer to yesterday; meanwhile the world has altered, and users will have changed their minds. Requirements, therefore, are at best a compromise.

The three dimensions proposed by Pohl (1993) (see Figure 1.2) illustrate three major problems that RE has to solve: namely modelling the future application in a

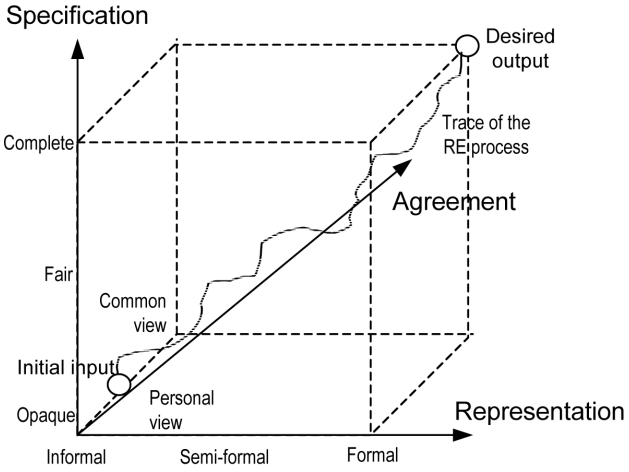

Fig. 1.2 The three dimensions of RE.

more complete manner, modelling with more formality, and with all the stakeholders agreeing requirements. Input to the process starts with coarse-grained and ambiguous statements about the users' requirements for the intended system. Users may have different visions of requirements or only partial and incomplete ideas about what they want. Input is characteristically informal in its representation, imprecise and personal since requirements are initially held by individuals and frequently conflict with one another. However, the desired output from RE is very different. It should be a complete system specification, within the constraints of available resources, using a formal language, and agreed by all involved.

The thread in Figure 1.2 traces the emerging requirements specification, as it becomes more complete, accurate and shared. On the specification dimension, RE has to guide the discovery, refining and validation of requirements as they become more thoroughly understood and complete. Representation has to support expression of requirement statements and models in natural language, semi-formal graphical notations such as data flow diagrams (DFDs) and entity relationship (ER) diagrams, and formal notations. Finally the agreement dimension has to support trade-offs between requirements and different stakeholders' views, negotiation and co-ordination.

Another view of requirements sees the problem of transforming a current system into a new, desired system. This view focuses our attention on the problem of understanding existing systems before we can design the necessary change. The view is elaborated in the four worlds framework (Jarke *et al.*, 1993) which is useful in understanding how requirements change and how they can be partitioned into different viewpoints (Figure 1.3).

The first world is where requirements usually start in the usage world of the user. The usage world describes requirements in their context of a domain and, possibly, an existing system. Requirements reflect the wishes and objectives of users, or their problems with a current system that needs fixing. Requirements are elicited or acquired from users in the usage world, and transferred into the subject world as models and representations of the domain. This involves abstracting a generalized picture from the detail in the usage world. Generalized pictures become models and specifications of the world and the designed system. Notice here the tension between requirements as single goals and requirements embedded as components of a model or specification. Should requirements be simple lists of functions or goals, or do we need models of objects, tasks, and processes to understand them in context? The answer is probably both, but the relative importance of lists and modelling is more difficult to judge. From the subject world, which represents part of modelled reality (also referred to as the universe of discourse) requirements are transformed into the system world. Here requirements become specifications of what will work in the new designed system. Requirements therefore progress from "why" questions of users' goals in the usage world to "what" questions of functional specifications in the subject world and finally "how" questions in the system world.

However, the progress from requirements to design is not as clear as this framework would suggest. Users frequently do not have a clear vision of what they want. If they do, their goals may be poorly formed and fuzzy. Furthermore, requirements are linked to people's knowledge of what is technically possible. This leads to the inevitable intertwining of requirements with design, to paraphrase Swartout and Balzer (1982). Only by creating some vision of the future can we be sure that it embodies our requirements and even then our understanding will be limited by how faithfully the current vision (or prototype) represents the final product. Natu-

rally, the more users change their minds during requirements analysis, the more the original vision will depart from the final design. So at the heart of RE there is a paradox: users don't know what they want until they get it, and when they get it they see how it could be improved or they don't like it. Trying to overcome this 20/20 foresight problem lies at the heart of RE.

The final world in Figure 1.3 is the development world. This is the realm of the programmers and software developers. The development world has its own requirements such as constraints on design for interoperability, maintainability, reliability, etc. In addition, software used to develop computer systems imposes another set of requirements which the requirements engineer has to deal with – for example, can the application be delivered in Java on client–server architecture, and if so is the data environment secure?

Having looked at some of the perspectives of RE, it is now necessary to investigate how different starting points for requirements may affect the process.

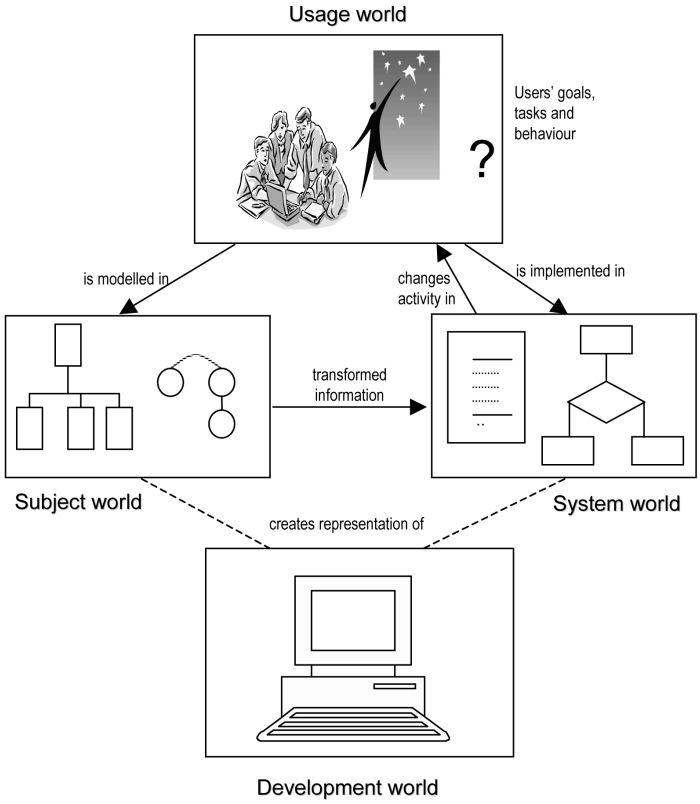

Fig. 1.3 The "four worlds" view of system development.

1.5 Requirements Types and RE Pathways

Although RE is often assumed to start with top-down decomposition to create goal hierarchies, it may also start with problems in an existing system rather than intentions to create a new system. Furthermore, user requirements may also be promoted by examples of other successful systems. Different applications affect the course of the RE process; for instance, COTS (commercial off-the-shelf) software selection is different from bespoke development; while information systems and real time domains make different demands on the RE process. Requirements come in many different shapes and sizes. Some may be high-level goals while others may be detailed rules and constraints. To give some idea of the range, the requirements in a library circulation system could include:

- the need for a complete sub-system or high-level functions: "the system will have facilities for auditing book stock so that old and redundant stock can be eliminated";
- specification of a more detailed function: "the circulation control system should calculate fines on overdue loans";
- a statement about how a function should work: "fines should be calculated as the number of days overdue (current date – due date) multiplied by a configurable fine factor";
- constraints on how the system should operate: "data on reader fines should be secure and not publicly accessible";
- statements about performance: "all search requests must be completed within 30 seconds of submission";
- implementation constraints: "the system must operate on a Linux platform as well as Windows NT".

The above requirements vary in the amount of detail provided, but even ones which look precise suffer from ambiguity. Take the third requirement in the list. This looks precise, but it hides further questions. How many configurable parameters are required, such as separate fine rates for standard readers, old age pensioners, readers who are frequently late, etc.? Many requirements are inaccurate and ambiguous. The RE process has to clarify what is meant as far as possible.

Not all requirements come from people. Requirements can be imposed on a system by laws of physics and facts of nature. For example, in a fly-by-wire avionics system the computer has to control an aircraft so that it flies at a certain speed, direction, height and orientation. Failure to do so will cause the aircraft to stall, or worse still, to become over-stressed and crash. Requirements in this case include the laws of physics such as gravity, aerodynamics and physical stress limits of the aircraft. The system must conform to these facts and process events in a specific order. This problem of requirements emanating from implications of events has been investigated by Jackson (1995) in several refinements of his method which derived entity life histories from explicit consideration of real world facts and behaviour. In more recent papers, the correspondence between real world events and system behaviour has been expressed in terms of optative requirements to which the system must respond (Jackson and Zave, 1993). These are obligations on the system, or required behaviour that it must carry out to ensure successful operation. According to Jackson's approach to RE, it is necessary to distinguish between descriptions of

the world as it exists and designations that reflect assumptions about properties of designed systems. Requirements emerge from understanding how the system should behave – *indicative requirements* – and other necessary responses by the system to change the world – *optative requirements*. Requirements are statements of what machine specifications should do given assumptions about the real world domain.

Some requirements are imposed externally as constraints on design required by law, others arise from problems which need fixing, and some are performance qualities rather than functions which might be implemented in software. Roman (1985) provided the first list of issues in RE and drew attention to the need for modelling functional and non-functional requirements. There are several taxonomies of requirements (see Pohl, 1996; Mazza *et al.*, 1994), which will not be elaborated here; however, it is necessary to explore some fundamental categories. A commonly held but disputed distinction is between functional and non-functional requirements:

- *Functional requirements* are statements about what a system should do, how it should behave, what it should contain, or what components it should have. Functional requirements are initially expressed as goals, e.g. "the search facility will find books that match the user's request"; or activities, e.g. "validate order", "monitor temperature". Later, requirements become specifications expressed as entities, attributes, actions, events or states: the familiar semantics of software engineering modelling languages. Functions are elaborated processes or procedures which can be implemented as software algorithms and data structures.
- *Non-functional requirements* (NFRs) are statements of quality, performance and environment issues with which the system must conform. Some examples are reliability, maintainability, portability (properties of the design), safety, security, scalability, accuracy, usability, and performance. NFRs are qualities and performance criteria that are not directly implementable in software. They can be further sub-divided into:
  - performance criteria such as throughput volumes (system must handle 10,000 transactions per working day), reliability, response time;
  - design-related constraints such as maintainability, interoperability (e.g. system works on both Microsoft and Linux platforms), security, usability.

The distinction between functional and non-functional requirements, when examined in detail, may crumble. NFRs set standards or benchmarks by which the system will be assessed but they also have design implications. Refining NFRs involves decisions about how well a design can meet the required criteria so, rather than being implemented, NFRs are satisfied to some extent by design. Mylopoulos and his colleagues (Mylopoulos, Chung and Nixon, 1992; Yu, 1994) describe NFRs as "soft goals" that, unlike functional requirements, cannot be completely specified in terms of software; instead, these requirements are "satisfyced" to some extent by functional requirements in a design. NFRs do not directly refer to what should be designed; instead they are statements of quality or performance criteria, for instance "the system shall process 1000 orders per day, with an average time per order of 20 seconds". How they are satisfied may involve defining functional requirements, so the dividing line between the two is not sharp. An example might be a safety requirement that "the autopilot system will fly the aircraft to ensure no collisions occur". This is a bit vague so the designer might unpack what "no colli-

sions" means as "the autopilot will detect any aircraft which comes within an envelope of 500 metres and take avoiding action". However, "avoiding action" has not been defined. This will become a set of rules or procedures about how to control the aircraft, such as if an aircraft approaches from behind then speed up, from below then climb, etc. In this manner a non-functional requirement has become refined into several functional requirements for monitoring the proximity of other aircraft and procedures for avoiding action. However, not all NFRs can be unpacked in this manner. Performance requirements become metrics or benchmarks against which the system design will be assessed.

One approach to dealing with benchmark-style NFRs (Briand *et al.*, 1995) is the goals-question-metric technique for software engineering quality assessment, even though Briand *et al.* do not explicitly refer to requirements. The essence of their method is to set goals as quality criteria that are measurable, with a metric linked to a particular need that is expressed in a question. For example:

Goal:	To achieve an acceptable response time of less than 0.02 seconds for user system interaction.
Questions:	How quickly will the system respond to users' editing operations with a normal workload?
Metric:	Delay from the end of a key press to the appearance of a visible change on the screen.

Hence NFRs will become refined into functional requirements as well as setting a necessary and sufficient criterion by which the system can be judged. For example, safety as an NFR will have design implications for how safety is delivered as well as being unpacked as a metric to specify what an acceptable level of safety means: in air traffic control "the system will prevent mid-air collisions, and the near-miss rate, where two aircraft pass within 1 mile or 500 vertical feet separation of each other, will be less than 1 per 100,000 aircraft movements".

1.6 Constraints on Design

Another way of thinking about requirements is to classify users' needs into goals or functions, attributes which describe the properties of qualities of the desired system and constraints that define the limitations on design. To elaborate these categories:

- *Goals* are functional and non-functional requirements that describe what the users want the system to do.
- *Attributes* are qualities or properties of the desired system, which may be functional or non-functional in nature, e.g. the system will be safe, reliable, operate efficiently, inexpensive, have aesthetic appeal.
- *Constraints* are conditions or laws that the system will have to obey during operation or during design, e.g. the system should operate in a temperature range of 0–30°C, and fit within a space of 1 cubic metre. Constraints fall into four subclasses:
  - Constraints on the physical shape and size of the product. In software these may be megabytes of memory or disc storage space; more generally they are limits on the size and shape of the product.

- Environmental constraints that set out the expected means and ranges of operating conditions, e.g. temperature, pressure, vibration, different locations, dust, dirt, noise, etc. While software is shielded from most of these environmental factors, human operators and other equipment are not, hence these constraints are important.
- Cost. This is always a key limitation on the designer's (and users') ambition.
- Legal constraints that set non-functional requirements for safety, reliability, usability, security and privacy.

Ideally, requirements would be developed for systems with an infinite budget and no dependencies on existing systems. Unfortunately such green-field applications rarely exist, and budgets are never unlimited. Requirements have to be specified that are realistic and achievable within the costs and constraints of an organization. Constraints vary from legislation imposed by government that may have implications for health, safety or working conditions, laws that influence the applications itself (e.g. tax laws and payroll calculations), to constraints on processes and procedures imposed by company policy (e.g. not offering discounts to poorly paying customers). Increasingly, requirements also have to be specified for system upgrades or for new systems that have to co-exist with previous implementations. Legacy software consists of suites of old programs, usually written in COBOL, that most companies possess to run their basic transaction processing applications, i.e. sales order processing, inventory control, general ledger accounting, etc. Legacy systems are often critical to a company's survival, so new systems have to be built as front-ends to the older systems. New requirements may imply changes to legacy systems which, given their antiquity, are poorly designed and hard to change. Since requirements were rarely documented in older systems, deciding requirements for modification or how new systems should fit within a legacy system can be a challenge.

Requirements can be reverse engineered or recovered from legacy code but the process is difficult. Programmers are notoriously bad documentors so the analyst is often left with just the code itself. The research area of reverse engineering (for overviews see Layzell, Freeman and Benedusi, 1995) deals with this problem of restructuring old code to make it more maintainable and reliable. If the code is well-designed with a modular structure then there is some chance of success that the original designer's intentions might be discernible; however, most old code has as much structure as a plate of spaghetti, so reverse engineering requirements can be a forlorn task. The users' goals were rarely documented in older systems and guessing them from algorithms and data structures is a nearly impossible task.

Cost is one of the major constraints on any project. Customers and users, if given the choice, will create requirements wish lists that could consume many times the development budget, so the requirements analyst has to prioritize requirements in collaboration with the users. Processes for doing so will be covered in Chapter 4. Some applications may not necessitate development of new code; instead the solution is to purchase COTS software. In this case the constraints are the available products and the range of functionality they provide. The requirements process therefore becomes a goodness-of-fit optimization, matching customer requirements and products' properties within the constraints of costs, delivery time, maintenance support, etc. A variation on COTS is configurable products, such as Enterprise Resource Plans (ERPs) from vendors such as SAP and Peoplesoft. These products provide generalized business application packages such as personnel, payroll, logistics, sales

order processing, etc. Such packages can be tailored to the customer's requirements within certain limits. They impose fewer constraints than COTS products but more than in a bespoke development of new software. For ERP products the requirements process has two phases: first, selecting the ERP vendor and application package; then, requirements for tailoring the package to the customer's business. The first phase follows a COTS-like goodness-of-fit process, while the second phase is constrained by the customization facilities provided (see Sutcliffe, 2002).

In summary, the requirements process is constrained by many realities of the environment in which it takes place: time, costs, laws, standards, and available resources. The transformation of requirements into design is also constrained by possibly having to co-exist with legacy systems, and standards for interoperability on operating systems, networks, etc. Finally, the requirements process might be limited by what is available in the market place for COTS-style developments. The art of successful RE is achieving the optimal within the bounds of the possible.

1.7 Documenting Requirements

Requirements are usually written down as natural language statements, but natural language is prone to misinterpretation. One response has been to advocate use of formal, mathematically based specification languages which make meaning more explicit. Unfortunately, formal specifications are not comprehensible to most users, so formal languages present a communication barrier. There is no easy escape from the dilemma of easy to understand but ambiguous natural language versus formal but inaccessible specification languages (see Chapter 3). Requirements need to express a range of possibilities in general terms during the early stage of the process, while being more precise later on (Fickas and Feather, 1995).

To avoid documents containing long, dense texts that nobody wants to read, requirements are formatted into structured lists to help understanding and retrieval. Structured requirements documents help by classifying and indexing requirements into categories so they can be found quickly and traced. Each requirement is still expressed in natural language but the statement is short and terse to reduce ambiguity. Structured requirements documents such as the standard defined by IEEE 830 (Mazza *et al.*, 1994) recommend the following headings:

1. Introduction
   - 1.1 Purpose of the requirements documents
   - 1.2 Scope of the product
   - 1.3 Definitions of acronyms and abbreviations
   - 1.4 References
   - 1.5 Overview of remainder of the document
2. General description
   - 2.1 Product perspective
   - 2.2 Product function
   - 2.3 User characteristics
   - 2.4 General constraints
   - 2.5 Assumptions and dependencies

3. Specific requirements
   (including functional, non-functional and user interface requirements, performance benchmarks, design and database constraints, system attributes and quality characteristics)
4. Appendices
5. Index

Requirements document management tools (e.g. RequisitePro, DOORS, CRADLE) provide configurable classification structures, with defaults based on the IEEE standard or similar layouts. Requirements as statements and lists need to be accompanied by diagrams and other information that helps their interpretation. Some authors refer to requirements documents as requirements specifications, by which they mean requirements statements accompanied by more formal models that can make the interpretation of natural language more precise. A further means of recording requirements to reduce ambiguity is to use templates. These provide a form that links the requirements with associated facts such as who authored (or captured) the requirements, from whom (the user), when, etc. A good example of requirements documentation templates (Figure 1.4) is given in the Volere method (Robertson and Robertson, 1999).

Requirements may represent the views of several different groups of people, called "stakeholders". For example, requirements documentation may be created by the following stakeholders:

- *Customers* who will be purchasing the system or otherwise financing its development. Customers are interested in making sure requirements fit with their objectives to deliver a business or organizational benefit.

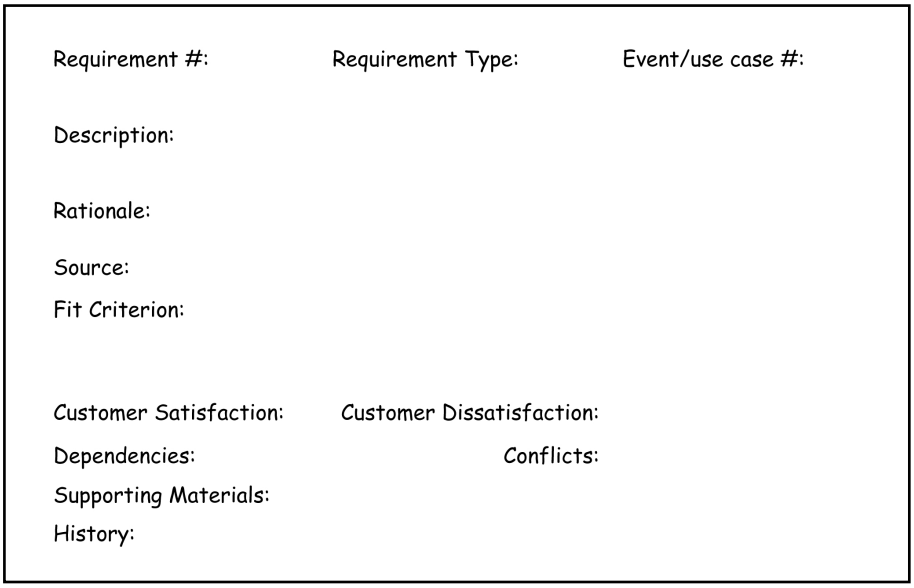

Fig. 1.4 Requirements documentation from the Volere method.

- *Users* – people who will actually operate the system. User stakeholders are concerned about more detailed requirements of system functionality and usability.
- *Managers* – stakeholders who are not direct users but manage the system. Managers may be the same as customers in some systems or may represent the department that is receiving the new system. They are interested in requirements as system outcomes to achieve business objectives.
- *Software engineers* – designers who have to translate requirements into software designs and code. They need requirements to be as precise as possible.
- *System testers* and quality assurance – stakeholders who will be responsible for ensuring that the designed system is reliable and meets with performance criteria. These stakeholders are interested in non-functional requirements and constraints.
- *System maintainers* – people who have to keep the system running when it is in operation and modify it. Maintenance personnel are rarely consulted in requirements analysis yet they depend on accurate requirements documentation more than most.

Different stakeholders have different views on a requirements document. Some want detail, others need to check that requirements will fulfil their objectives. In the early stages of RE there is considerable merit is having incomplete and ambiguous requirements. Trying to nail everything down in detail too early may constrain exploration of the problem and cause people to commit too early to a poorly thought out solution.

As RE is primarily a computer science discipline there is a hidden assumption that all requirements are about software. This is not true. Many requirements will be satisfied by management decisions to allocate resources or change human operating procedures. A useful distinction is between system and software requirements:

- *System requirements* – requirements in the wider sense that record any particular need for a new system. These requirements may specify obligations for human operators, organizational resources, or machinery, as well as implying software requirements.
- *Software requirements* – requirements in a narrower sense that pertain to intended software functions. Software requirements are derived from system requirements and make assumptions about design, i.e. what is going to be automated in software.

In conclusion, requirements documents are formatted lists of natural language statements accompanied by additional text, sketches and diagrams. These documents are used by several different groups of stakeholders who have different interests and focuses when they read requirements documents. Interests can be served by a continuum of representation from information statements in natural language to more precise specifications, but finding a single lingua franca that can be both precise and sufficiently flexible to meet the needs of all stakeholders is a difficult task. Chapter 4 explores how combinations of representations can address this problem.

1.8 Summary

RE has developed to tackle the vexed problem of obtaining the needs for a new system as accurately and completely as possible. Failure to carry out effective requirements analysis has led to many system disasters, and requirements errors become progressively more expensive to cure as system development progresses. The subject area of RE has evolved from information systems, object-oriented development methods and software engineering. RE is difficult because it has to deal with requirements which may never be complete because users discover requirements only when they experience a design and because the external world changes and creates new requirements after a system has been implemented. Requirements analysis has to deal with communication problems and the ambiguity of natural language. People may not provide correct information because their knowledge is tacit, or for political and personal motivations to hide sensitive facts. Requirements can be divided into functional and non-functional requirements. Functional requirements are captured as user goals and refined into specifications for the design. NFRs become benchmarks or performance and quality criteria that a design has to achieve, but during analysis functional requirements are discovered to achieve them. Requirements are documented in structured lists defined in standards and using templates that record the authors, stakeholders and associated information. Different stakeholders will have differing needs and views on requirements documents.