1

Setting the Scene

This chapter introduces requirements engineering (RE) as a specific discipline in relation to others. It defines the scope of RE and the basic concepts, activities, actors and artefacts involved in the RE process. In particular, it explains what requirements there are with respect to other key RE notions such as domain properties and environment assumptions. Functional and non-functional requirements will be seen to play specific roles in the RE process. The quality criteria according to which requirements documents should be elaborated and evaluated will be detailed. We will also see why a careful elaboration of requirements and assumptions in the early stages of the software lifecycle is so important, and what obstacles may impinge on good RE practice.

The chapter also introduces three case studies from which running examples will be taken throughout the book. These case studies will additionally provide a basis for many exercises at the end of chapters. They are taken from quite different domains to demonstrate the wide applicability of the concepts and techniques. Although representative of real-world systems, the case study descriptions have been simplified to make our examples easily understandable without significant domain expertise. The first case study is a typical instance of an information system. The second captures the typical flavour of a system partly controlled by software. The third raises issues that are typical of distributed collaborative applications and product families.

COPYRIGHTED MATERIAL

1.1 What is requirements engineering?

To make sure that a software solution correctly solves a particular problem, we must first correctly understand and define what problem needs to be solved. This seems common sense at first sight. However, as we shall see, figuring out what the right problem is can be surprisingly difficult. We need to discover, understand, formulate, analyse and agree on *what* problem should be solved, *why* such a problem needs to be solved and *who* should be involved in the responsibility of solving that problem. Broadly, this is what requirements engineering is all about.

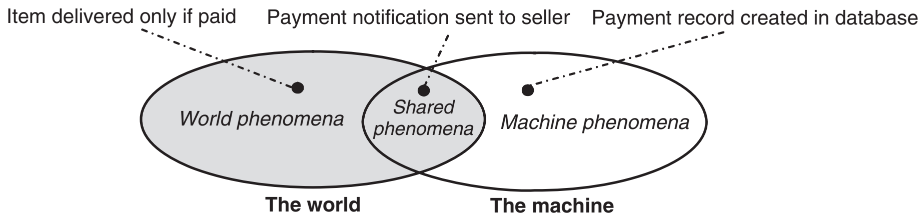
Item delivered only if paid
Payment notification sent to seller
Payment record created in database
World phenomena
Shared phenomena
Machine phenomena
The world
The machine

Figure 1.1 *The problem world and the machine solution*

1.1.1 The problem world and the machine solution

The problem to be solved arises within some broader context. It is in general rooted in a complex organizational, technical or physical *world*. The aim of a software project is to improve this world by building some *machine* expected to solve the problem. The machine consists of software to be developed and installed on some computer platform, possibly together with some input/output devices.

The problem world and the machine solution have their own phenomena while sharing others (Jackson, 1995b). The shared phenomena define the interface through which the machine interacts with the world. The machine monitors some of the shared phenomena while controlling others in order to implement the requirements.

Figure 1.1 illustrates this for a simple e-commerce world. In this example, the world owns the phenomena of items being delivered to buyers only once they have been paid; the machine owns the phenomena of payment records being created in the machine's database. The phenomena of payment notifications being sent to sellers are shared, as the machine can control them whereas the world can monitor them.

Requirements engineering is concerned with the machine's effect on the surrounding world and the assumptions we make about that world. As a consequence, it is solely concerned with world phenomena, including shared ones. Requirements and assumptions have their meaning in the problem world. In contrast, software design is concerned with machine phenomena.

The system-as-is and the system-to-be

In the system engineering tradition, the word *system* will be used throughout the book to denote a set of components interacting with each other to satisfy some global objectives. While being intrinsically composite, a system can be seen as a whole through the global properties emerging from component interactions. Such properties include the objectives underpinning component interactions and laws regulating such interactions.

- *Example 1.* Consider an e-auction system on the Internet. This system is made up of components such as sellers, buyers, shipping companies, an independent e-payment subsystem, e-mail systems, and the software to be developed or extended for inserting and advertising items, handling bids, billing highest bidders, recording evaluations of sellers and buyers, securing transactions and so forth. Global properties emerging from component interactions include the satisfaction of buyers getting wider access to interesting items, the satisfaction of sellers getting wider access to potential buyers, auction rules regulating the system, trustworthiness relationships and so on.

- *Example 2.* A flight management system includes components such as pilots, air traffic controllers, on-board and on-ground instruments, the autopilot software to be developed, an independent collision-avoidance subsystem and so forth. Global properties emerging from component interactions include the objectives of rapid and safe transportation of passengers, regulating laws about wind directions, aircraft speed, minimal distance between aircrafts and so forth.

In a machine-building project, our business as requirements engineers is to investigate the problem world. This leads us to consider two versions of the same system:

- The *system-as-is*, the system as it exists before the machine is built into it.
- The *system-to-be*, the system as it should be when the machine will be built and operated in it.

In the previous example of an auction world, the system-as-is is a standard auction system with no support for electronic bidding. The system-to-be is intended to provide such support in order to make items biddable from anywhere at any time. In a flight management world, the system-as-is might include some autopilot software with limited capabilities; the system-to-be would then include autopilot software with extended capabilities. In the former example the system-to-be is the outcome of a new software project, whereas in the latter example it results from a software evolution project.

Note that there is always a system-as-is. Consider a project aimed at developing control software for a MP4 player, for example. The system-as-is is the conventional system allowing you to listen to your favourite music on a standard hi-fi subsystem. The system-to-be is intended to mimic the listening conditions of the system-as-is while providing convenient, anywhere and any-time access to your music.

The software-to-be and its environment

The machine's software to be developed or modified is just one component of the system-to-be. We will refer to it as the *software-to-be*. Other components will in general pertain to the machine's surrounding world. They will form the *environment* of the software-to-be. Such components may include:

- People or business units playing specific roles according to organizational policies.
- Physical devices operating under specific rules in conformance with physical laws – for example sensors, actuators, measurement instruments or communication media.
- Legacy, off-the-shelf or foreign software components with which the software-to-be needs to interact.

As we are concerned with the problem world, we need to consider both the system-*as-is*, to understand its objectives, regulating laws, deficiencies and limitations, and the system-*to-be*, to elaborate the requirements on the software-to-be accordingly together with assumptions on the environment.

The systems-to-be-next

If we want to build an evolvable machine in our problem world, we need to anticipate likely changes at RE time. During software development or after deployment of the system-to-be, new problems and limitations may arise. New opportunities may emerge as the world keeps changing. We may then even need to consider more than two system versions and foresee what the next system versions are likely to be. Beyond the system-as-is and the system-to-be, there are *systems-to-be-next*. Requirements evolution management is an important aspect of the RE process that will be discussed at length in Chapter 6.

Requirements engineering: A preliminary definition

In this setting, we may apprehend requirements engineering more precisely as a coordinated set of activities for exploring, evaluating, documenting, consolidating, revising and adapting the objectives, capabilities, qualities, constraints and assumptions that the system-to-be should meet based on problems raised by the system-as-is and opportunities provided by new technologies. We will come back to those various activities in Section 1.1.6 and will have a much closer look at them in subsequent chapters.

1.1.2 Introducing our running case studies

To make the nature of RE more apparent and more concrete, we will consider a variety of case studies. The following descriptions are intended to set up the context in which our running examples will be used throughout the book. They will also provide further insights into the scope and dimensions of the problem world. We should not consider them as problem statements, but rather as fragmentary material collected from preliminary investigations of the problem world (perhaps by use of the elicitation techniques discussed in Chapter 2).

Case study 1: Library Management

The University of Wonderland (UWON) wants to convert its library system into a new system to ensure more effective access to state-of-the-art books, periodicals and proceedings while reducing operational costs. The current system consists of multiple unconnected library subsystems, one for each UWON department. Each department subsystem is responsible for its own library according to department-specific procedures for book acquisition, user registration, loan management, bibliographical search and access to library resources. Such services are essentially manual in most UWON libraries. They rely on card indexes maintained by library staff according to some keyword-based classification scheme. Such schemes are specific to each department. A few departments are using rudimentary file-based software written by their members.

Some of the complaints about the current system as reported by university authorities, library staff, department members or students include the following:

- Unnecessary duplicate acquisition, by several departments, of infrequently accessed copies of books or proceedings that are relevant to more than one department.
- Unnecessary subscription, by several departments, to expensive journals that are relevant to more than one department.
- Acquisition of books or proceedings of marginal interest to the university, which could be borrowed from other universities with which UWON has an agreement.
- Subscription to journals of marginal interest to the university, which could be accessed in other universities with which UWON has an agreement.
- Unavailability of requested books, for a variety of reasons such as department budget restrictions, excessive borrowing by the same user, lack of enforcement of rules limiting loan periods, loss or stealing of book copies and so on.
- Unavailability of journal issues while they are being bound into yearly volumes.
- Lack of traceability to previous borrowers when books, proceedings or journal volumes are found to be damaged.
- Inaccuracy of card indexes, e.g. a book is stated as being available whereas it is not found at the appropriate place in the shelves.
- Bibliographical search restricted to library opening hours.
- Slow, tedious bibliographical search due to manipulation of card indexes.
- Inaccurate search results, due to poor classification of books, journals or proceedings within departments.
- Incomplete or ineffective search results, due to relevant books, journals or proceedings being indexed in other UWON department libraries, or unavailable at UWON.

The new UWON library system should address such problems through a software-based solution integrating all department libraries. The new system should interoperate with library systems from partner universities.

It should provide interactive online facilities for book acquisition, user registration, loan management, bibliographical search and book reservation. Access to such facilities should be restricted to specific user categories, according to authorization rules specific to each facility.

The new system should take advantage of opportunities provided by new technologies. In particular, it should support subscriptions to e-journals, provide access to foreign digital libraries (under specific conditions), support e-mail communication between staff and users, enable bibliographical search from anywhere at any time, and provide a Web-based interface for book e-seller comparison, selection, and order submission.

Case study 2: Train Control

Traffic at Wonderland airport (WAX) has increased drastically over the past few years. The increase in the number of companies and flights calls for the building of new terminals. The increase in the number of passengers calls for a new transportation system between all terminals, and between the main terminal and Wonderland City. The current bus transportation system has reached its limits in terms of transportation capacity and quality of service. Buses are slow and often late due to traffic jams; passengers need to stand in long queues, sometimes for an unacceptably long time, which may cause them to miss flight connections, and so on.

The government of Wonderland has decided to replace bus transportation by a train-based system. The envisaged system is aimed at increasing transportation capacity, speed and quality of service. The decision is also motivated by recent regulations for reducing greenhouse gas production.

Preliminary investigations suggest that software-controlled movement of trains will allow for better punctuality, higher frequency and better information to passengers.

A consortium has been set up to undertake this project. It brings together government representatives, airport authorities, Wonderland Railways and the engineering company selected to implement the project. The latter is subcontracting the software part of it to a software house.

In the new system, all terminals will be interconnected through an underground circular, one-track railway. The main terminal and city terminal will be interconnected by a two-track line (one for each direction). The main terminal also has parking tracks for inactive trains, servicing and so on. Each track is divided into track segments of a fixed size called blocks. Each terminal holds one block called a station block. Each block is equipped with an entry signal (or 'virtual gate') and multiple sensors to detect the presence of trains, identify trains and their speed and so on.

The envisioned software is expected to control the acceleration of trains, the opening of train doors, the block signals and the display of information about the current/next station on information panels inside trains. The railway company would also like to reduce operational costs. A fully automated, driverless option is envisaged as an alternative to the standard option. In this standard option, train drivers have to follow recommendations issued by the software and respond to regular stimuli issued by the software to check driver responsiveness. The driverless option is currently being discussed with the unions.

Various concerns about the new system have already emerged at this preliminary stage:

- In order to ensure rapid transportation of passengers, trains should run fast, without unnecessary delays, and at high frequency, during rush hours at least.
- In order to guarantee safe transportation of passengers, the probability of accidents must fall below the threshold imposed by safety regulations. In particular, the distance between two trains following each other must always be sufficient to prevent the back train from hitting the front train in case the latter stops suddenly. The speed of a train on a particular block may never exceed the limit associated with that block. Trains may never enter a block whose entry signal is set to 'stop'. Train doors must always be kept closed while a train is moving.
- In order to ensure comfortable transportation, trains should accelerate/decelerate smoothly. Passengers at a station should be informed in time about trains arriving. Passengers inside a train should be informed in time that the train is departing, which companies are being served by the next stop and so on.

Case study 3: Meeting Scheduling*

With the advent of globalization, companies and organizations are increasingly distributed over multiple sites and countries. Wonderland Software

* *Source*: Adapted from S. Fickas, A. Finkelstein, M. Feather and A. Van Lamsweerde, 1997, with kind permission of Springer Science and Business Media.

Services (WSS) has identified a large potential market for meeting-scheduling software that would exploit Internet-based communication technologies. Scheduling meetings with busy people is generally a nightmare. It is hard to find a date and a place that suit everyone's constraints; meeting organizers need to pester people to get their availability; other people are unnecessarily inconvenienced by messages that do not concern them; when the meeting is scheduled some constraints have changed in the meantime; new scheduling cycles need to be repeated when no date/location is found in a reasonably short period; and so forth. As a result, meetings tend to be organized poorly and late; important people sometimes do not show up; and there is a significant, unnecessary overhead in the scheduling process.

Meetings are typically scheduled as follows. A meeting initiator informs potential participants about the need for a meeting and specifies a date range within which the meeting should take place, asking people to return their availability constraints within that time interval. Constraints are typically expressed as two sets: an exclusion set specifying dates within the date range when the participant could not attend, and an optional preference set specifying dates within the date range on which the participant would prefer the meeting to take place (a date may refer to a full day or a period in a day). In some cases, the initiator may also ask participants who will play an active role in the meeting for specific requirements regarding the meeting room (e.g. projector, laptop, network connection, videoconferencing facilities etc.). 'Important' participants may optionally be asked to state preferences for meeting locations.

The scheduled meeting date should belong to the stated date range and to none of the exclusion sets; it should ideally belong to as many preference sets as possible. The meeting venue should ideally fit the preferences of important participants. A date conflict occurs when no date can be found outside all exclusion sets. A room conflict occurs when no room can be found, at any date outside all exclusion sets, which meets the room requirements. Conflicts can be resolved in several ways: the initiator may extend the date range, some participants may remove dates from their exclusion set, or some participants may decline the invitation to attend. A new scheduling cycle may thus be required in case of conflict.

The envisioned meeting scheduler software should reflect as closely as possible the way meetings are typically managed. It should be useable by administrative staff and provide major improvements in several respects:

- Average participant attendance should increase thanks to the selection of meeting dates and locations that are the most convenient to potential participants.

- Meetings should be scheduled as quickly as possible once they are initiated.
- Meeting dates and locations should be notified as quickly as possible to all potential participants once they are scheduled. In all cases, there should be sufficient time between notification and the meeting date.
- The organizational overhead should be kept as low as possible on the initiator's side. In particular, the meeting scheduler should support all required interactions with participants, for example to communicate requests, get replies (even from participants not reacting promptly), assist in negotiation and conflict-resolution processes, and inform participants on request about the state of the scheduling process.
- The amount of interaction with potential participants for meeting scheduling, in number and length of messages, should be kept as small as possible.

The new meeting scheduler must be able to handle multiple meeting requests in parallel. Meeting requests can be competing by overlapping in time or space. Concurrency must thus be managed under physical constraints; a person may not be at two different places at the same time, and a meeting room may not be allocated to more than one meeting at a time.

To allow as much flexibility as possible, dynamic replanning of meetings should be supported. On the one hand, participants should be allowed to modify their exclusion set, preference set and/or preferred location until the meeting is scheduled. On the other hand, exceptional constraints should be accommodated after a meeting is scheduled, such as the need to schedule an urgent, more important meeting. The original meeting date or location may then need to be changed; sometimes it may even be cancelled. In all cases some way of replanning should be set up.

The system should be flexible enough to accommodate different data formats (e.g. date or address formats) and evolving data (e.g. the set of concerned participants may vary during the scheduling process, and the address at which a participant can be reached may change).

There are also security concerns to be taken into account, such as the following:

- Meeting initiation should be restricted to authorized personnel.
- Confidentiality rules should be enforced, for instance a non-privileged participant should not be aware of constraints stated by other participants, or of other meetings to which the latter are invited.

Rather than a single product, WSS is thinking of a product family. The customization space should cover the following variations:

- Professional meetings, private meetings.
- Single-site meetings, meetings where the target site needs to be determined as well, electronic meetings.
- Regular meetings (e.g. for a university course), occasional meetings.
- Single-level meetings or multi-level meetings where the importance of a person attending a specific meeting is higher or lower with respect to other meetings.
- Single-level participation or multi-level participation where the importance of a meeting getting a specific attendee is higher or lower with respect to other attendees.
- Single-level participants or multi-level participants where some participants are hierarchically more important than others (regardless of a specific meeting) or have less flexibility in changing their constraints.
- Variations on what participating in a meeting means, e.g. full attendance, partial attendance, participation through delegation.
- Variations on what constraints are about, e.g. no preference set, unordered preferences, ordered preferences, date availability dependent on meeting location.
- Parameterizability on explicit conflict-resolution rules that are tunable by the client, e.g. 'best meeting dates and locations should be determined by considering participants with higher importance first', 'in case of date conflict the scheduler will propose a person of lower importance to withdraw from the meeting', 'in case of date conflict the meeting scheduler will propose a participant to withdraw from another meeting of lower importance', 'a date within some exclusion set will be considered if the corresponding participant has high flexibility'.
- Mono-lingual, multi-lingual communication with participants.
- Variations on additional features such as support for elaborating the meeting agenda or the meeting minutes.

1.1.3 The WHY, WHAT and WHO dimensions of requirements engineering

The preceding case-study descriptions give us a preliminary idea of the wide range of issues we need to consider in the RE process.

As noted before, the investigation of the problem world leads us to consider two versions of the system. The system-*as-is* has problems, deficiencies and limitations. For example, money is wasted in the UWON library system-as-is by the acquisition of duplicate or rarely used resources; access to bibliographical search facilities is severely limited in both time and location. In the WAX transportation system-as-is, flight connections are missed due to slow bus transportation and poor information to passengers. In the meeting scheduling system-as-is, meeting initiators are overloaded and meeting dates are not chosen well enough, which sometimes results in poor meeting attendance.

The system-*to-be* is intended to address those problems based on technology opportunities. It will do so only if the software-to-be and the organizational and physical components defining the environment are able to cooperate effectively. In the UWON library system-to-be, the new software has to cooperate effectively with environmental components such as patrons, staff, anti-theft devices, digital libraries and external library systems. In the WAX train system-to-be, the train-control software has to operate in conjunction with environmental components such as track sensors, train actuators, passengers, information panel devices and so forth. In the meeting scheduling system-to-be, the scheduler software has to cooperate effectively with environmental components such as meeting initiators and participants, e-mail systems, e-agenda managers, the communications network and so on. In the end, what really matters is the satisfactory working of the software–environment pair.

The problem world may thus be structured along three dimensions. We need to figure out *why* a system-to-be is needed, *what* needs must be addressed by it, and *who* in this system will take part in fulfilling such needs (see Figure 1.2).

The WHY dimension

The contextual reasons for a new version of a system must be made explicit in terms of objectives to be satisfied by it. Such objectives must be identified with regard to the limitations of the system-as-is and the opportunities to be exploited. This requires some careful analysis. What are those objectives precisely? What are their ramifications? How do they interact? How do they align with business objectives?

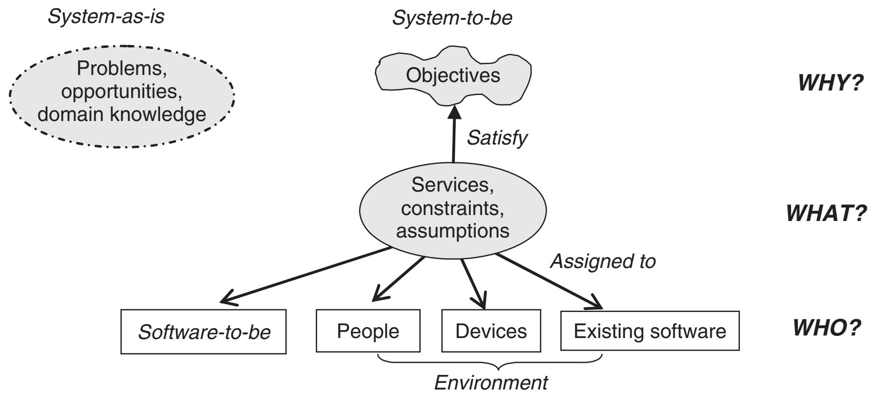

Figure 1.2 *Three dimensions of requirements engineering*

As we will see more thoroughly in subsequent chapters, such analysis along the WHY dimension is generally far from simple.

Acquiring domain knowledge We need to get a thorough understanding of the domain in which the problem world is rooted. This domain might be quite complex in terms of concepts, regulating laws, procedures and terminology. If you are not sufficiently convinced by the complexity of library management or meeting scheduling, think of domains such as air traffic control, proton therapy, power plants or stock exchanges. (Chapter 2 will present techniques to help us acquire domain knowledge.)

Evaluating alternative options in the problem world There can be alternative ways of satisfying the same identified objective. We need to assess the pros and cons of such alternatives in order to select the most preferable one. (Chapters 3 and 16 will present techniques to support this task.)

- *Example: Library management.* We could satisfy the objective of extensive coverage of the literature through a non-selective journal subscription policy or, alternatively, through access to digital libraries. The two alternatives need to be evaluated in relation to their pros and cons.
- *Example: Train control.* We could satisfy the objective of avoiding train collisions by ensuring that there will never be two trains on the same block or, alternatively, by ensuring that trains on the same block will always be separated by some worst-case stopping distance. The pros and cons of each alternative must be assessed carefully.
- *Example: Meeting scheduling.* The objective of knowing the time constraints of invited participants could be satisfied by asking them their constraints via e-mail or, alternatively, by accessing their electronic agenda directly.

Evaluating technology opportunities We also need to acquire a thorough understanding of the oppportunities provided by technologies emerging in the domain under consideration, together with their implications and risks. For example, what are the strengths, implications and risks associated with digital libraries, driverless trains or e-agendas?

Handling conflicts The objectives that the system-to-be should satisfy are generally identified from multiple sources which have conflicting viewpoints and interests. As a result, there may be different perceptions of what the problems and opportunities are; and there may be different views on how the perceived problems should be addressed. In the end, a coherent set of objectives needs to emerge from agreed trade-offs. (Chapters 3, 16 and 18 will present techniques to support that task.)

- *Example: Library management.* All parties concerned with the library system-to-be will certainly agree that access to state-of-the-art books and journals should be made more effective. There were sufficient complaints reported about this in the system-as-is.

Conflicts are likely to arise, though, when this global objective is refined into more concrete objectives in order to achieve it. Everyone will acclaim the objective of improving the effectiveness of bibliographical search. However, university authorities are likely to emphasize the objective of cost reduction through integration of department libraries. Departments might be reluctant to accede to the implications of this, such as losing their autonomy. On the other hand, library staff might be concerned by strict enforcement of rules limiting library opening periods, the length of loan periods or the number of loans to the same patron. In contrast, library patrons might want much more flexible usage rules.

- *Example: Train control.* All parties will agree on the objectives of faster and safer transportation. Conflicts will, however, appear between the railway company management and the unions while exploring the pros and cons of alternative options with or without drivers, respectively.

The WHAT dimension

This RE dimension is concerned with the *functional services* that the system-to-be should provide to satisfy the objectives identified along the WHY-dimension (see Figure 1.2). Such services often rely on specific system *assumptions* to work properly. They need to meet *constraints* related to performance, security, usability, interoperability and cost – among others. Some of the services will be implemented by the software-to-be whereas others will be realized through manual procedures or device operations.

The system services, constraints and assumptions may be identified from the agreed system objectives, from usage scenarios envisioned in the system-to-be, or from other elicitation vehicles discussed in Chapter 2. They must be formulated in precise terms and in a language that all parties concerned understand to enable their validation and realization. They should be traceable back to system objectives so that we can argue that the latter will be satisfied. The formulation of software services must also be mapped to precise specifications for use by software developers (the nature of this mapping will become clearer in Section 1.1.4).

This analysis of the required services, constraints and assumptions is in general far from simple, as we will see in greater detail in subsequent chapters. Some might be missing; others might be inadequate with respect to objectives stated explicitly or left implicit; others might be formulated ambiguously or inconsistently.

- *Example: Library management.* We might envision a bibliographical query facility as a desirable software service in the UWON library system-to-be. To enable validation, we should define this service in terms that are comprehensible by library staff or by the students who would use it. The definition should make it possible to argue that the objectives of increased coverage, information accuracy and wider accessibility will be achieved through that service. One assumption to meet the objective of anywhere/anytime accessibility is that library users do have Web access outside library opening hours. Constraints on the bibliographical query service might refer to the average response time to a query, the interaction mode and query/answer format for useability by non-experts, and user privacy (e.g. non-staff users should not be able to figure out what other users have borrowed).

- *Example: Train control.* For the WAX train system-to-be, we must define the service of computing train accelerations in terms that allow domain experts to establish that the objective of avoiding collisions of successive trains will be guaranteed. There should be critical constraints on maximum delays in transmitting acceleration commands to trains, on the readability of such commands by train drivers so as to avoid confusion and so forth. Assumptions about the train-tracking subsystem should be made explicit and validated.

The WHO dimension

This RE dimension addresses the assignment of responsibilities for achieving the objectives, services and constraints among the components of the system-to-be – humans, devices or software. Decisions about responsibility assignments are often critical; an important objective, service or constraint might not be achieved if the system component responsible for it fails to behave accordingly.

- *Example: Library management.* The objective of accurate book classification will not be achieved if department faculty members, who might be made responsible for it, do not provide accurate keywords when books are acquired in their area. The objective of limited loan periods for increased availability of book copies will not be achieved if borrowers do not respond to warnings or threatening reminders, or if the software that might be responsible for issuing such reminders in time fails to do so.
- *Example: Train control.* The objective of safe train acceleration will not be achieved if the software responsible for computing accelerations produces values outside the safety range, or if the driver responsible for following the safe instructions issued by the software fails to do so.

Responsibility assignments may also require the evaluation of alternative options. The same responsibility might be assignable to different system components, each alternative assignment having its pros and cons. The selected assignment should keep the risks of not achieving important system objectives, services or constraints as small as possible.

- *Example: Library management.* The objective of accurate book classification might be assigned to the software-to-be; the latter would retrieve relevant keywords from electronic abstracts supplied by publishers, and classify books accordingly. The downsides of such an assignment are increased development costs and the risk of sometimes bizarre classifications. The same objective might alternatively be assigned to the relevant department, at the risk of piles of books waiting for overloaded faculty members to classify them.
- *Example: Train control.* The objective of safe train accelerations might be under the direct responsibility of the software-to-be, in a driverless alternative, or under the responsibility of train drivers who would follow indications issued by the software-to-be. Each alternative has associated strengths and risks that need to be analysed carefully.

Elaborating the software-environment boundary As illustrated by the previous examples, alternative responsibility assignments generally yield different system proposals in which more or less functionality is automated. When we select responsibility assignments from multiple alternatives, we make decisions on what is going to be automated in the system-to-be and what is not. The boundary between the software-to-be and its environment thus emerges from such decisions. This boundary is rarely fixed a priori when the RE process starts. Assessing alternative boundaries and deciding on a specific one is an important aspect of the RE process along the WHO dimension.

1.1.4 Types of statements involved in requirements engineering

Throughout the RE process we need to collect, elaborate, correct or adapt statements that may differ in mood and in scope (Jackson, 1995a; Parnas & Madey, 1995).

Descriptive vs prescriptive statements

Descriptive statements state properties about the system that hold regardless of how the system behaves. Such properties hold typically because of some natural law or physical constraint. Descriptive statements are in the indicative mood. For example, the following statements are descriptive:

- If train doors are open, they are not closed.
- The same book copy cannot be borrowed by two different people at the same time.
- A person cannot physically attend two meetings on different continents on the same day.

Prescriptive statements state desirable properties about the system that may hold or not depending on how the system behaves. Such statements need to be enforced by system components. They are in the optative mood. For example, the following statements are prescriptive:

- Train doors shall always remain closed when the train is moving.
- A patron may not borrow more than three books at the same time.
- The meeting date must fit the constraints of all important participants.

The distinction between descriptive and prescriptive statements is essential to make in the context of engineering requirements. We may need to negotiate, weaken, change or find alternatives to prescriptive statements. We cannot negotiate, weaken, change or find alternatives to descriptive statements.

Statement scope

Section 1.1.1 introduced a partition of phenomena into world, machine and shared phenomena to make the point that RE is concerned with the problem world only. If we focus our attention on the software part of the machine we want to build, we obtain a similar

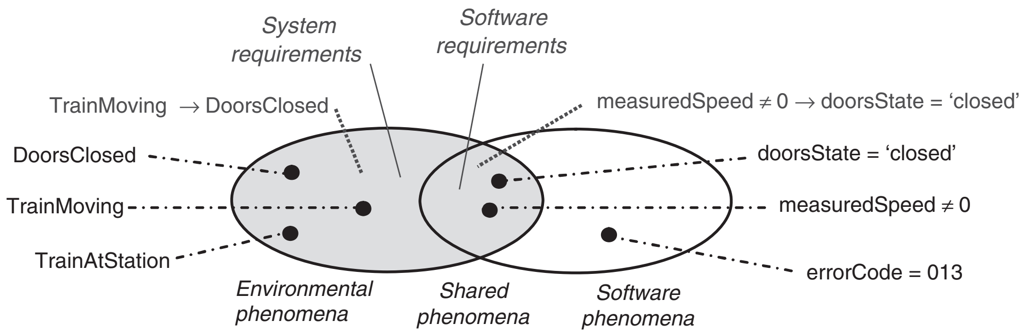

Figure 1.3 *Phenomena and statements about the environment and the software-to-be*

partition. A phenomenon is owned by the software-to-be, by its environment, or shared among them. The environment includes the machine's input/output devices such as sensors and actuators.

For example, the phenomenon of a train physically moving is owned by the environment (see Figure 1.3); the software controller cannot directly observe whether the train is moving or not. The phenomenon of a train's measured speed being non-null is shared by the software and the environment; it is controlled by a speedometer in the environment and observed by the software. The phenomenon of an error variable taking a particular value under a particular condition is owned by the software; the environment cannot directly observe the state of this variable.

The RE process involves statements about the system-to-be that differ in scope. Some statements may refer to phenomena owned by the environment without necessarily being shared with the software-to-be. Other statements may refer to phenomena shared between the environment and the software-to-be; that is, controlled by the software and observed by the environment, or vice versa.

In view of those differences in mood and scope, we can now more precisely define the various types of statement involved in the RE process. Their interrelationships will be discussed next.

Requirements, domain properties, assumptions and definitions

A *system requirement* is a prescriptive statement to be enforced by the software-to-be, possibly in cooperation with other system components, and formulated in terms of environmental phenomena. For example:

- All train doors shall always remain closed while a train is moving.
- Patrons may not borrow more than three books at a time.
- The constraints of a participant invited to a meeting should be known as soon as possible.

Satisfying system requirements may require the cooperation of other system components in addition to the software-to-be. In the first example above, the software train controller might be in charge of the safe control of doors; the cooperation of door actuators is also needed, however (passengers should also be required to refrain from opening doors unsafely).

As we will see in Section 1.1.6, the system requirements are to be understood and agreed by all parties concerned with the system-to-be. Their formulation in terms of environmental phenomena, in the vocabulary used by such parties, will make this possible.

A *software requirement* is a prescriptive statement to be enforced solely by the software-to-be and formulated only in terms of phenomena shared between the software and the environment. For example:

- The doorsState output variable shall always have the value 'closed' when the measuredSpeed input variable has a non-null value.
- The recorded number of loans by a patron may never exceed a maximum number x.
- A request for constraints shall be e-mailed to the address of every participant on the meeting invitee list.

A software requirement constrains the observable behaviours of the software-to-be in its environment; any such behaviour must satisfy it. For example, any software behaviour where *measuredSpeed* $\neq$ *0* and *doorsState* = 'open' is ruled out according to the first software requirement in the above list.

Software requirements are to be used by developers; they are formulated in the vocabulary of developers, in terms of software input/output variables.

Note that a software requirement is a system requirement by definition, while the converse is not true (see Figure 1.3). When no ambiguity arises, we will often use the term *requirement* as a shorthand for 'system requirement'.

The notion of system requirement is sometimes referred as 'user requirement' or 'customer requirement' in the literature or in descriptions of good practice. The notion of software requirement is sometimes referred as 'product requirement', 'specification' or even, misleadingly, 'system requirement'. We will avoid those phrases in view of possible confusion. For example, many 'user requirements' do not come from any software user; a 'system' does not merely consist of software; a 'specification' may refer in the software engineering literature both to a process and to a variety of different products along the software lifecycle (requirement specification, design specification, module specification, test case specification etc.).

A *domain property* is a descriptive statement about the problem world. It is expected to hold invariably regardless of how the system will behave – and even regardless of whether there will be any software-to-be or not. Domain properties typically correspond to physical laws that cannot be broken. For example:

- A train is moving if and only if its physical speed is non-null.
- A book may not be borrowed and available at the same time.
- A participant cannot attend multiple meetings at the same time.

An *assumption* is a statement to be satisfied by the environment and formulated in terms of environmental phenomena. For example:

- A train's measured speed is non-null if and only if its physical speed is non-null.
- The recorded number of loans by a borrower is equal to the actual number of book copies physically borrowed by him or her.
- Borrowers who receive threatening reminders after the loan deadline has expired will return books promptly.
- Participants will promptly respond to e-mail requests for constraints.
- A participant is on the invitee list for a meeting if and only if he or she is invited to that meeting.

Assumptions are generally prescriptive, as they constrain the behaviour of specific environmental components. For example, the first assumption in the previous list constrains speedometers in our train control system.

The formulation of requirements, domain properties and assumptions might be adequate or not. We will come back to this throughout the book. The important point here is their difference in mood and scope.

Definitions are the last type of statement involved in the RE process. They allow domain concepts and auxiliary terms to be given a precise, complete and agreed meaning – the same meaning for everyone. For example:

- TrainMoving is the name for a phenomenon in the environment that accounts for the fact that the train being considered is physically moving on a block.
- A patron is any person who has registered at the corresponding library for the corresponding period of time.
- A person participates in a meeting if he or she attends that meeting from beginning to end.

Unlike statements of other types, definitions have no truth value. It makes no sense to say that a definition is satisfied or not. However, we need to check definitions for accuracy, completeness and adequacy. For example, we might question the above definition of what it means for a person to participate in a meeting; as a result, we might refine the concept of participation into two more specialized concepts instead – namely, full participation and partial participation.

In view of their difference in mood and scope, the statements emerging from the RE process should be 'typed' when we document them (we will come back to this in Section 4.2.1). Anyone using the documentation can then directly figure out whether a statement is a requirement, a domain property, an assumption or a definition.

Relating software requirements to system requirements

The link between the notions of system requirement and software requirement can be made more precise by introducing the following types of variables:

- *Monitored variables* are environmental quantities that the software monitors through input devices such as sensors.
- *Controlled variables* are environmental quantities that the software controls through output devices such as actuators.
- *Input variables* are data items that the software needs as input.
- *Output variables* are quantities that the software produces as output.

These different types of variable yield a more explicit framework for control systems, known as the *four-variable model* (Parnas and Madey, 1995); see Figure 1.4. As we can see there, input/output devices are highlighted as special interface components between the control software and its environment.

In this framework, we can define system requirements and software requirements as distinct mathematical relations. Let us use the standard notations $\subseteq$ and $\times$ for set inclusion and set Cartesian product, respectively.

- A system requirement *SysReq* is a relation between a set M of monitored variables and a corresponding set C of controlled variables:

$$SysReq \subseteq M \times C$$

- A software requirement *SofReq* is a relation between a set I of input variables and a corresponding set O of output variables:

$$SofReq \subseteq I \times O$$

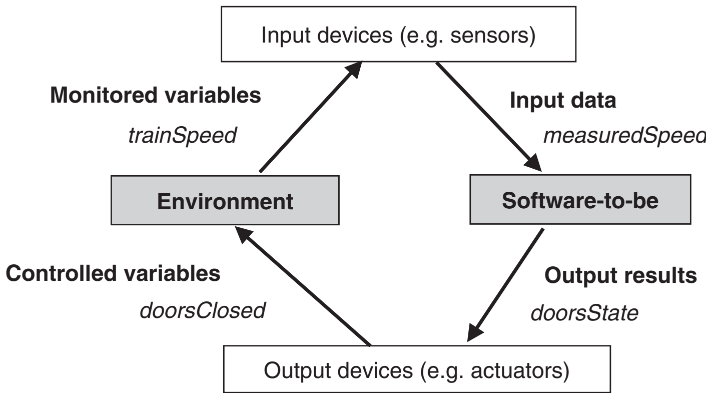

Figure 1.4 *Four-variable model*

A software requirement *SofReq* 'translates' the corresponding system requirement *SysReq* in the vocabulary of the software's input/output variables.

Satisfaction arguments

Such translation of a system requirement into a software requirement is not a mere reformulation obtained by mapping the environment's vocabulary into the software's one. Domain properties and assumptions are often required to ensure the 'correctness' of the translation; that is, the *satisfaction* of the system requirement when the corresponding software requirement holds.

Let us illustrate this very important point. We first introduce some shorthand notations:

$$A \rightarrow B \text{ for '}\textit{if}\ A\ \textit{then}\ B', \quad A \leftrightarrow B \text{ for '}A\ \textit{if and only if}\ B'.$$

We may express the above examples of system requirement, software requirement, domain property and assumption for our train system in the shorter form:

(*SysReq*:)	TrainMoving $\rightarrow$ DoorsClosed
(*SofReq*:)	measuredSpeed $\neq 0 \rightarrow$ doorsState = 'closed'
(*Dom*:)	TrainMoving $\leftrightarrow$ trainSpeed $\neq 0$
(*Asm*:)	measuredSpeed $\neq 0 \leftrightarrow$ trainSpeed $\neq 0$
	DoorsState = 'closed' $\leftrightarrow$ DoorsClosed

To ensure that the software requirement *SofReq* correctly translates the system requirement *SysReq* in this simple example, we need to identify the domain property *Dom* and the assumptions *Asm*, and make sure that those statements are actually satisfied. If this is the case, we can obtain *SysReq* from *SofReq* by the following rewriting: (a) we replace *measuredSpeed* $\neq$ *0* in *SofReq* by *TrainMoving*, thanks to the first equivalence in the assumptions *Asm* and then the equivalence in the domain property *Dom*; and (b) we replace *doorsState* = '*closed*' in *SofReq* by *DoorsClosed* thanks to the second equivalence in *Asm*.

The assumptions in *Asm* are examples of *accuracy* statements, to be enforced here by the speedometer and door actuator, respectively. Accuracy requirements and assumptions form an important class of non-functional statements to be considered in the RE process (see Section 1.1.5). Overlooking them or formulating wrong ones has sometimes been the cause of major software disasters. We will come back to this throughout the book.

Our job as requirements engineers is to elicit, make precise and consolidate requirements, assumptions and domain properties. Then we need to provide *satisfaction arguments* taking the following form:

$$\{SOFREQ, \text{ASM}, \text{DOM}\} \models SysReq$$

which reads:

> **if** the software requirements in set *SOFREQ* are satisfied by the software, the assumptions in set *ASM* are satisfied by the environment, the domain properties in set *DOM* hold and all those statements are consistent with each other,
>
> **then** the system requirements *SysReq* are satisfied by the system.

Such a satisfaction argument could not be provided in our train example without the statements *Asm* and *Dom* previously mentioned. Satisfaction arguments require environmental assumptions and domain properties to be elicited, specified and validated. For example, is it the case that the speedometer and door actuator will always enforce the first and second assumptions in *Asm*, respectively?

In Chapter 6, we will see that satisfaction arguments play an important role in managing the traceability among requirements and assumptions for requirements evolution. In Part II of this book, we will extend them to higher-level arguments for goal satisfaction by requirements and assumptions.

1.1.5 Categories of requirements

In the above typology of statements, the requirements themselves are of different kinds. Roughly, functional requirements refer to services that the software-to-be should provide, whereas non-functional requirements constrain how such services should be provided.

Functional requirements

Functional requirements define the functional effects that the software-to-be is required to have on its environment. They address the 'WHAT' aspects depicted in Figure 1.2. Here are some examples:

- The bibliographical search engine shall provide a list of all library books on a given subject.
- The train control sofware shall control the acceleration of all the system's trains.
- The meeting scheduler shall determine schedules that fit the diary constraints of all invited participants.

The effects characterized by such requirements result from *operations* to be automated by the software. Functional requirements may also refer to environmental conditions under which such operations should be applied. For example:

- Train doors may be opened only when the train is stopped.
- The meeting scheduler shall issue a warning when the constraints entered by a participant are not valid.

Functional requirements characterize units of functionality that we may want to group into coarser-grained functionalities that the software should support. For example, bibliographical search, loan management and acquisition management are overall functionalities of the library software-to-be. Units of functionality are sometimes called *features* in some problem worlds; for example, call forwarding and call reactivation are features generally provided in telephony systems.

Non-functional requirements

Non-functional requirements define constraints on the way the software-to-be should satisfy its functional requirements or on the way it should be developed. For example:

- The format for submitting bibliographical queries and displaying answers shall be accessible to students who have no computer expertise.
- Acceleration commands shall be sent to every train every 3 seconds.
- The diary constraints of a participant may not be disclosed to any other invited participant.

The wide range of such constraints makes it helpful to classify them in a taxonomy (Davis, 1993; Robertson & Robertson, 1999; Chung *et al*., 2000). Specific classes can then be characterized more precisely. Browsing through the taxonomy may help us acquire instances of the corresponding classes that might have been overlooked (Section 2.2.7 will come back to this).

Figure 1.5 outlines one typical classification. The taxonomy there is not meant to be exhaustive, although it covers the main classes of non-functional requirements.

Quality requirements

Quality requirements state additional, quality-related properties that the functional effects of the software should have. They are sometimes called 'quality attributes' in the software engineering literature. Such requirements complement the 'WHAT' aspects with 'HOW WELL' aspects. They appear on the left-hand side in Figure 1.5.

Safety requirements are quality requirements that rule out software effects that might result in accidents, degradations or losses in the environment. For example:

- The controlled accelerations of trains shall always guarantee that a worst-case stopping distance is maintained between successive trains.

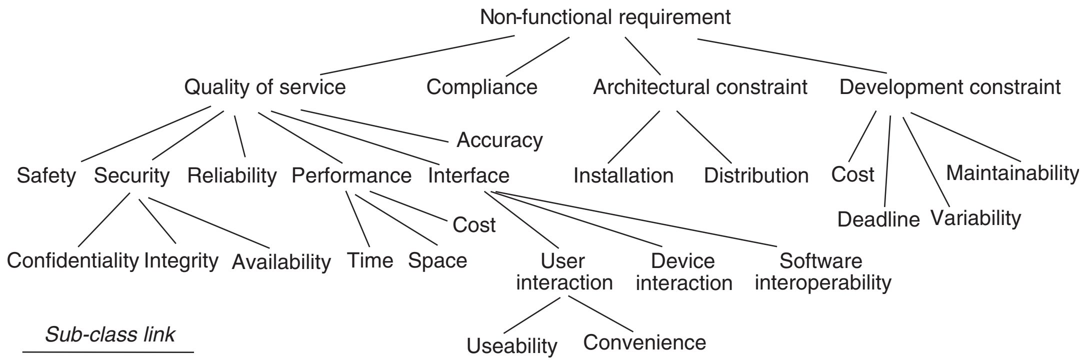

Figure 1.5 *A taxonomy of non-functional requirements*

Security requirements are quality requirements that prescribe the protection of system assets against undesirable environment behaviours. This increasingly critical class of requirements is traditionally split into subcategories such as the following (Amoroso, 1994; Pfleeger, 1997).

Confidentiality requirements state that some sensitive information may never be disclosed to unauthorized parties. For example:

- A non-staff patron may never know which books have been borrowed by others.

Among these, *privacy requirements* state that some private information may never be disclosed without the consent of the owner of the information. For example:

- The diary constraints of a participant may never be disclosed to other invited participants without his or her consent.

Integrity requirements state that some information may be modified only if correctly done and with authorization. For example:

- The return of book copies shall be encoded correctly and by library staff only.

Availability requirements state that some information or resource can be used at any point in time when it is needed and its usage is authorized. For example:

- A blacklist of bad patrons shall be made available at any time to library staff.
- Information about train positions shall be available at any time to the vital station computer.

Reliability requirements constrain the software to operate as expected over long periods of time. Its services must be provided in a correct and robust way in spite of exceptional circumstances. For example:

- The train acceleration control software shall have a mean time between failures of the order of 10^9 hours.

Accuracy requirements are quality requirements that constrain the state of the information processed by the software to reflect the state of the corresponding physical information in the environment accurately. For example:

- A copy of a book shall be stated as available by the loan software if and only if it is actually available on the library shelves.
- The information about train positions used by the train controller shall accurately reflect the actual position of trains up to X metres at most.

- The constraints used by the meeting scheduler should accurately reflect the real constraints of invited participants.

Performance requirements are quality requirements that constrain the software's operational conditions, such as the time or space required by operations, the frequency of their activation, their throughput, the size of their input or output and so forth. For example:

- Responses to bibliographical queries shall take less than 2 seconds.
- Acceleration commands shall be issued to every train every 3 seconds.
- The meeting scheduler shall be able to accommodate up to X requests in parallel.

Performance requirements may concern other resources in addition to time or space, such as money spent in operational costs. For example:

- The new e-subscription facility should ensure a 30% cost saving.

Interface requirements are quality requirements that constrain the phenomena shared by the software-to-be and the environment (see Figure 1.3). They refer to the static and dynamic aspects of software-environment interactions; input/output formats and interaction sequences should be compatible with what the environment expects. Interface requirements cover a wide range of concerns depending on which environmental component the software is interacting with.

For human interaction, *useability requirements* prescribe input/output formats and user dialogues to fit the abstractions, abilities and expectations of the target users. For example:

- The format for bibliographical queries and answers shall be accessible to students from any department.

Other human interaction requirements may constrain software effects so that users feel them to be 'convenient' in some system-specific sense. For example:

- To ensure smooth and comfortable train moves, the difference between the accelerations in two successive commands sent to a train should be at most X.
- To avoid disturbing busy people unduly, the amount of interaction with invited participants for organizing meetings should be kept as low as possible.

For interaction with devices or existing software components, *interoperability requirements* prescribe input/output formats and interaction protocols that enable effective cooperation with those environmental components. For example:

- The meeting scheduling software should be interoperable with the WSS Agenda Manager product.

Figure 1.5 covers other categories of non-functional requirements in addition to quality requirements.

Compliance requirements

Compliance requirements prescribe software effects on the environment to conform to national laws, international regulations, social norms, cultural or political constraints, standards and the like. For example:

- The value for the worst-case stopping distance between successive trains shall be compliant with international railways regulations.
- The meeting scheduler shall by default exclude official holidays associated with the target market.

Architectural requirements

Architectural requirements impose structural constraints on the software-to-be to fit its environment, typically:

- *Distribution constraints* on software components to fit the geographically distributed structure of the host organization, the distribution of data to be processed, or the distribution of devices to be controlled.
- *Installation constraints* to ensure that the software-to-be will run smoothly on the target implementation platform.

Here are some examples of architectural requirements:

- The on-board train controllers shall handle the reception and proper execution of acceleration commands sent by the station computer.
- The meeting scheduling software should cooperate with email systems and e-agenda managers of participants distributed worldwide.
- The meeting scheduling software should run on Windows version X.x and Linux version Y.y.

Architectural requirements reduce the space of possible software architectures. They may guide developers in the selection of an appropriate architectural style, for example an event-based style. We will come back to this in Section 16.5.

Development requirements

Development requirements are non-functional requirements that do not constrain the way the software should satisfy its functional requirements but rather the way it should be developed (see the right-hand part of Figure 1.5). These include requirements on development costs, delivery schedules, variability of features, maintainability, reusability, portability and the like. For example:

- The overall cost of the new UWON library software should not exceed X.
- The train control software should be operational within two years.
- The software should provide customized solutions according to variations in type of meeting (professional or private, regular or occasional), type of meeting location (fixed, variable) and type of participant (same or different degrees of importance).

Possible overlaps between categories of requirements

The distinction between *functional* and *non-functional* requirements should not be taken in a strict, clear-cut sense. The boundary between those two categories is not always clear. For example, consider the following requirement in a safety injection system for a nuclear power plant (Courtois & Parnas, 1993):

- The safety injection signal shall be on whenever there is a loss of coolant except during normal start-up or cool down.

Is this a functional or a safety requirement? Both, we might be inclined to say. Similarly, many functional requirements for a firewall management software are likely to be security requirements as well. Call screening is traditionally considered as a functional feature in telephony software, even though it is about keeping the caller's phone number confidential.

Likewise, some of the non-functional requirements categories in Figure 1.5 may overlap in specific situations. Consider, for example, a denial-of-service attack on a patient's file that prevents surgeons from accessing critical patient data during surgery. Does this violate a security or a safety requirement? Similarly, the requirement to send acceleration commands to trains at very high frequency is related to both performance and safety. More generally, availability requirements often contribute to both security and reliability.

Uses of requirements taxonomies

In spite of possible overlaps in specific situations, what matters in the end are the roles and benefits of a requirements taxonomy in the RE process.

a. *More specific characterization of requirements.* Requirements categories allow us to characterize more explicitly what requirements refer to, beyond our general definition as prescriptive statements to be enforced by the software and formulated in terms of environmental phenomena.

b. *More semantic characterization of requirements.* The distinction between requirements categories allows for a more semantic characterization of requirements in terms of prescribed *behaviours*.

   - There are requirements that prescribe *desired* behaviours. For example, scheduler behaviours should result in a meeting being scheduled every time a corresponding request has been submitted. Many functional requirements are of this kind.

- There are requirements that rule out *unacceptable* behaviours. For example, any train controller behaviour that results in trains being too close to each other must be avoided. Many safety, security and accuracy requirements are of this kind.
- There are requirements that indicate *preferred* behaviours. For example, the requirement that '*participants shall be notified of the scheduled meeting date as soon as possible*' states a preference for scheduler behaviours where notification is sooner over behaviours where notification is later. Likewise, the requirement that '*interactions with participants should be kept as limited as possible*' states a preference for scheduler behaviours where there are fewer interactions (e.g. through e-agenda access) over behaviours where there are more interactions (e.g. through e-mail requests and pestering). Many performance and '-ility' requirements are of this kind, for example useability, reuseability, portability or maintainability requirements. When alternative options are raised in the RE process, we will use such requirements to discard alternatives and select preferred ones (see Chapters 8 and 16).

c. *Differentiation between confined and cross-cutting concerns.* Functional requirements tend to address single points of functionality. In contrast, non-functional requirements tend to address cross-cutting concerns; the same requirement may constrain multiple units of functionality. In the library system, for example, the useability requirement on accessibility of input/output formats to non-expert users constrains the bibliographical search functionality. It may, however, constrain other functionalities as well, for example user registration or book reservation. Similarly, the non-disclosure of participant constraints might affect multiple points of functionality such as meeting notification, information on the current status of planning, replanning and so on.

d. *Basis for RE heuristics.* The characterization of categories in a requirements taxonomy yields helpful heuristics for the RE process. Some heuristics may help elicit requirements that were overlooked, for example:

   - Is there any accuracy requirement on information X in my system?
   - Is there any confidentiality requirement on information Y in my system?

   Other heuristics may help discover conflicts among instances of requirements categories known to be potentially conflicting, for example:

   - Is there any conflict in my system between hiding information on display for better useability and showing critical information for safety reasons?
   - Is there any conflict in my system between password-based authentication and useability requirements?
   - Is there any conflict in my system between confidentiality and accountability requirements?

We will come back to such heuristics in Chapters 2 and 3 while reviewing techniques for requirements elicitation and evaluation. As we will see there, conflict detection is a prerequisite for the elaboration of new requirements for conflict resolution.

1.1.6 The requirements lifecycle: Processes, actors and products

As already introduced briefly, the requirements engineering process is composed of different activities yielding various products and involving various types of actors.

A *stakeholder* is a group or individual affected by the system-to-be, who may influence the way this system is shaped and has some responsibility in its acceptance. As we will see, stakeholders play an important role in the RE process. They may include strategic decision makers, managers of operational units, domain experts, operators, end-users, developers, subcontractors, customers, and certification authorities.

For example in our library management system, stakeholders might include the UWON board of management, department chairs, library staff from the various departments and from partner universities, ordinary users and software consultants. In the WAX transportation system, stakeholders might include airport authorities, government representatives, airline companies, Wonderland Railways, passengers, union representatives and software subcontractors. In the meeting scheduling system, stakeholders might include a variety of people who schedule meetings (local, international, intra- or inter-organization meetings), a variety of people who attend such meetings under different positions, secretaries and software consultants.

Note that the set of stakeholders may vary slightly from the system-as-is to the system-to-be. In the WAX transportation system-to-be, for example, bus drivers will no longer be involved whereas railways personnel will.

In spite of their difference in aim and supporting techniques, the activities composing the RE process are highly intertwined. We review them individually first and then discuss their interaction.

Domain understanding

This activity consists of studying the system-as-is within its organizational and technical context. The aim is to acquire a good understanding of:

- The domain in which the problem world is rooted.
- What the roots of the problem are.

More specifically, we need to get an accurate and comprehensive picture of the following aspects:

- The organization within which the system-as-is takes place: its structure, strategic objectives, business policies, roles played by organizational units and actors, and dependencies among them.
- The scope of the system-as-is: its underlying objectives, the components forming it, the concepts on which it relies, the tasks involved in it, the information flowing through it, and the constraints and regulations to which the system is subject.
- The set of stakeholders to be involved in the RE process.

- The strengths and weaknesses of the system-as-is, as perceived by the identified stakeholders.

The product of this activity typically consists of the initial sections in a preliminary draft proposal that describe those contextual aspects. This proposal will be expanded during the elicitation activity and then used by the evaluation activity that comes after.

In particular, a *glossary of terms* should be established to provide definitions of key concepts on which everyone should agree. For example, in the library system-as-is, what precisely is a patron? What does it mean to say that a requested book is being reserved? In the train system, what precisely is a block? What does it mean to say that a train is at a station? In the meeting scheduling system, what is referred to by the term 'participant'? What does it mean to say that a person is invited to a meeting or participates in it? What precisely are participant constraints?

A glossary of terms will be used throughout the RE process, and even beyond, to ensure that the same term does not refer to different concepts and the same concept is not referred to under different terms.

Domain understanding is typically performed by studying key documents, investigating similar systems and interviewing or observing the identified stakeholders. The cooperation of the latter is obviously essential for our understanding to be correct. Chapter 2 will review techniques that may help us in this task.

Requirements elicitation

This activity consists of discovering candidate requirements and assumptions that will shape the system-to-be, based on the weaknesses of the system-as-is as they emerge from domain understanding. What are the symptoms, causes and consequences of the identified deficiencies and limitations of the system-as-is? How are they likely to evolve? How could they be addressed in the light of new opportunities? What new business objectives could be achieved then?

The aim is thus to explore the problem world with stakeholders and acquire the following information:

- The opportunities arising from the evolution of technologies and market conditions that could address the weaknesses of the system-as-is while preserving its strengths.
- The improvement objectives that the system-to-be should meet with respect to such weaknesses and opportunities, together with alternative options for satisfying them.
- The organizational and technical constraints that this system should take into account.
- Alternative boundaries that we might consider between what will be automated by the software-to-be and what will be left under the responsibility of the environment.
- Typical scenarios illustrating desired interactions between the software-to-be and its environment.
- The domain properties and assumptions about the environment that are necessary for the software-to-be to work properly.

- The requirements that the software-to-be should meet in order to conform to all of the above.

The requirements are by no means there when the project starts. We need to discover them incrementally, in relation to higher-level concerns, through exploration of the problem world. Elicitation is a *cooperative learning* process in which the requirements engineer and the system stakeholders work in close collaboration to acquire the right requirements. This activity is obviously critical. If done wrong, it will result in poor requirements and, consequently in poor software.

The product of the elicitation activity typically consists of additional sections in the preliminary draft proposal initiated during the domain understanding activity. These sections document the items listed above. The resulting draft proposal will be used as input to the evaluation activity coming next.

The elicitation process can be supported by a variety of techniques, such as knowledge reuse, scenarios, prototyping, interviews, observation and the like. Chapter 2 will discuss these.

Evaluation and agreement

The aim of this activity is to make informed decisions about issues raised during the elicitation process. Such decisions are often based on 'best' trade-offs on which the involved parties should agree. Negotiation may be required in order to reach a consensus.

- *Conflicting concerns* must be identified and resolved. These often arise from multiple viewpoints and different expectations.
- There are *risks* associated with the system that is being shaped. They must be assessed and resolved.
- The *alternative options* identified during elicitation must be compared with regard to quality objectives and risks, and best options must be selected on that basis.
- Requirements *prioritization* is often necessary for a number of reasons:
  a. Favouring higher-priority requirements is a standard way of resolving conflicts.
  b. Dropping lower-priority requirements provides a way of integrating multiple wishlists that would together exceed budgets and deadlines.
  c. Priorities make it easier to plan an incremental development process, and to replan the project during development as new constraints arise such as unanticipated delays, budget restrictions, deadline contractions etc.

The product of this activity typically consists of final sections in the preliminary draft proposal initiated during the preceding activities. These sections document the decisions made after assessment and negotiation. They highlight the agreed requirements and assumptions about the selected system-to-be. The system proposal thereby obtained will serve as input to the specification activity coming next.

The evaluation process can be supported by a variety of qualitative and quantitative techniques. Chapters 3 and 16 will provide a comprehensive sample of these.

Specification and documentation

This activity consists of detailing, structuring and documenting the agreed characteristics of the system-to-be as they emerge from the evaluation activity.

The resulting product is the *requirements document* (RD). In this document, the objectives, concept definitions, relevant domain properties, responsibilities, system requirements, software requirements and environmental assumptions are specified precisely and organized into a coherent structure. These specifications form the core of the RD. Satisfaction arguments should appear there as well (see Section 1.1.4.). Other sections in the RD may include a description of likely variants and revisions, acceptance test data and cost figures. The RD may also be complemented by annexes such as the preliminary system proposal after domain understanding, elicitation and evaluation, to provide the context and rationale for decisions taken, as well as technical annexes about the domain.

The requirements document will be used for a variety of purposes throughout the software lifecycle, as we will see in Section 1.1.9 (see Figure 1.7). To enable validation and commitment, any RD portion that concerns specific parties, such as customers, domain experts, (sub)contractors, developers or users, must be specified in a form understandable by them.

A wide range of techniques can be used to support the specification and documentation process, including structured natural language templates, diagrammatic notations and formal specifications. Chapter 4 will discuss these.

Requirements consolidation

The purpose of this activity is quality assurance. The specifications resulting from the preceding activity must be carefully analysed. They should be *validated* with stakeholders in order to pinpoint inadequacies with respect to actual needs. They should also be *verified* against each other in order to find inconsistencies and omissions before the software requirements are transmitted to developers. Any error found must be fixed. The sooner an error is found, the cheaper the fix will be.

The main product of this activity is a consolidated requirements document, where the detected errors and flaws have been fixed throughout the document. Other products may include a prototype or mock-up built for requirements validation, additional test data coming out of verification, a proposed development plan, the contract linking the client and the software developer, and a call for tenders in the case of development subcontracting.

Section 1.1.7 will detail the quality criteria addressed by this activity more precisely, together with the various types of errors and flaws that may need to be fixed. Section 1.2 will discuss the consequences of not fixing them. Chapter 5 will present techniques for requirements quality assurance.

Requirements engineering: A spiral process

The above activities are sometimes called *phases* of the RE process. There are, of course, *data dependencies* among them. Consolidation requires input from specification; specification requires input from evaluation; evaluation requires input from elicitation; and elicitation requires input from domain understanding. We should not think of these phases as being applied in a strict sequence, however. They are generally intertwined, they may overlap, and backtracking from one phase to preceding ones may be required.

Overall, the RE process can be viewed as an iteration on successive increments according to a spiral model (Boehm, 1988; Kotonya & Sommerville, 1997). Figure 1.6 shows such process model for the activities previously discussed in this section.

Each iteration in Figure 1.6 is triggered by the need to revise, adapt or extend the requirements document through addition, removal or modification of statements such as requirements, assumptions or domain properties.

A new iteration may take place at different stages of the software lifecycle:

- Within the RE process itself, as such statements are found during consolidation to be missing, inadequate or inconsistent with others.
- During software development, as such statements turn out to be missing, unfeasible or too costly to implement, incompatible with new implementation constraints, or no longer adequate as the problem world has evolved in the meantime.
- After software deployment, as the problem world has evolved or must be customized to specific contexts.

'Late' iterations of the RE process will be further discussed in Chapter 6 on evolution management and in Section 16.5 where the interplay between RE and architectural design will appear more clearly.

The spiral process model depicted in Figure 1.6 is fairly general and flexible. It may need to be specialized and adapted to the specificities of the problem world and to the standards

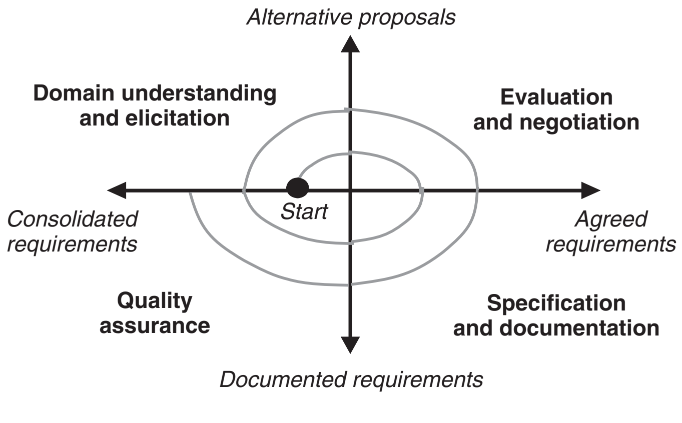
Alternative proposals
Domain understanding and elicitation
Evaluation and negotiation
Consolidated requirements
Start
Agreed requirements
Quality assurance
Specification and documentation
Documented requirements

Figure 1.6 *The requirements engineering process*

of the host organization, for example by further defining the nature of each increment or the intertwining with software development cycles. The important points, though, are the range of issues to consider, the complementarity and difference among RE activities, their data dependencies and the iterative nature of the RE process.

1.1.7 Target qualities and defects to avoid

Elaborating a good requirements document is difficult. We need to cater for multiple and diverse quality factors. Each of these may be hard to reach.

Quality factors define the goals of the RE process. They provide the basis for evaluating successive versions of the requirements document. This section defines them precisely together with their opposite, that is, the requirements defects that we must avoid. References to those qualities and defects will appear throughout the book. In particular, Chapter 5 and Parts II and III will detail a variety of techniques for checking them. Let us start with the qualities first.

- *Completeness.* The requirements, assumptions and domain properties, when taken together, must be sufficient to ensure that the system-to-be will satisfy all its objectives. These objectives must themselves be fully identified, including quality-related ones. In other words, the needs addressed by the new system must be fully covered, without any undesirable outcomes. In particular, we must have anticipated incidental or malicious behaviours of environmental components so that undesirable software effects are ruled out through dedicated requirements. A requirement on software behaviour must prescribe a desired output for all possible inputs. The specification of requirements and assumptions must also be sufficiently detailed to enable subsequent software development.
- *Consistency.* The requirements, assumptions and domain properties must be satisfiable when taken together. In other words, they must be compatible with each other.
- *Adequacy.* The requirements must address the *actual* needs for a new system – explicitly expressed by stakeholders or left implicit. The software requirements must be adequate translations of the system requirements (see Section 1.1.4). The domain properties must correctly describe laws in the problem world. The environmental assumptions must be realistic.
- *Unambiguity.* The requirements, assumptions and domain properties must be formulated in a way that precludes different interpretations. Every term must be defined and used consistently.
- *Measureability.* The requirements must be formulated at a level of precision that enables analysts to evaluate alternative options against them, developers to test or verify whether an implementation satisfies them, and users to determine whether they are met or not in the system under operation. The assumptions must be observable in the environment.
- *Pertinence.* The requirements and assumptions must all contribute to the satisfaction of one or several objectives underpinning the system-to-be. They must capture elements of the problem world rather than elements of the machine solution.

- *Feasibility.* The requirements must be realizable in view of the budget, schedule and technology constraints.
- *Comprehensibility.* The formulation of requirements, assumptions and domain properties must be comprehensible by the people who need to use them.
- *Good structuring.* The requirements document should be organized in a way that highlights the structural links among its elements – refinement or specialization links, dependency links, cause–effect links, definition–use links and so forth. The definition of a term must precede its use.
- *Modifiability.* It should be possible to revise, adapt, extend or contract the requirements document through modifications that are as local as possible.
- *Traceability.* The *context* in which an item of the requirements document was created, modified or used should be easy to retrieve. This context should include the rationale for creation, modification or use. The *impact* of creating, modifying or deleting that item should be easy to assess. The impact may refer to dependent items in the requirements document and to dependent artefacts subsequently developed – architectural descriptions, test data, user manuals, source code etc. (Traceability management will be discussed at length in Section 6.3.)

Note that critical qualities such as completeness, adequacy and pertinence are not defined in an absolute sense; they are *relative* to the underlying objectives and needs of a new system. The latter may themselves be implicit, unclear or even unidentified. Those qualities can therefore be especially hard to enforce.

Section 1.2.1 will review some facts and figures about project failures that are due to poor-quality requirements. Elaborating a requirements document that meets all of the above qualities is essential for the success of a software project. The techniques described in this book are aimed at supporting this task. As a prerequisite, we should be aware of the corresponding types of defect to avoid.

Requirements errors and flaws

Table 1.1 lists various types of defects frequently found in requirements documents. Each entry in Table 1.1 corresponds to the opposite of one of the preceding qualities. Table 1.2 and Table 1.3 suggest examples of defects that we might find in requirements documents for our case studies.

The defect types in Table 1.1 can be divided into two classes according to the potential severity of their consequences.

There are *errors* whose occurrence may have fatal effects on the quality of the software-to-be:

- Omissions may result in the software failing to implement an unstated critical requirement, or failing to take into account an unstated critical assumption or domain property.
- We cannot produce a correct implementation from a set of requirements, assumptions and domain properties that contradict each other.

Omission	Problem world feature not stated by any RD item – e.g. missing objective, requirement or assumption; unstated software response to some input.
Contradiction	RD items defining a problem world feature in an incompatible way.
Inadequacy	RD item not adequately stating a problem world feature.
Ambiguity	RD item allowing a problem world feature to be interpreted in different ways – e.g. ambiguous term or statement.
Unmeasurability	RD item stating a problem world feature in a way that cannot be precisely compared with alternative options, or cannot be tested or verified in machine solutions.
Noise	RD item yielding no information on any problem world feature.
Overspecification	RD item stating a feature not pertaining to the problem world but to the machine solution.
Unfeasibility	RD item that cannot be realistically implemented within the assigned budget, schedule or development platform.
Unintelligibility	RD item stated in an incomprehensible way for those who need to use it.
Poor structuring	RD items not organized according to any sensible and visible structuring rule.
Forward reference	RD item making use of problem world features that are not defined yet.
Remorse	RD item stating a problem world feature too late or incidentally.
Poor modifiability	RD items whose modification may need to be globally propagated throughout the RD.
Opacity	RD item whose rationale, authoring or dependencies are invisible.

Table 1.1 *Defects in a requirements document (RD)*

- Inadequacies may result in a software implementation that meets requirements, assumptions or domain properties that are not the right ones.
- Ambiguous and unmeasurable statements may result in a software implementation built from interpretations of requirements, assumptions or domain properties that are different from the intended ones.

In addition to errors, there are *flaws* whose consequences are in general less severe. In the best cases they result in a waste of effort and associated risks:

- Useless effort in finding out that some noisy or overspecified aspects are not needed – with the risk of sticking to overspecified aspects that may prevent better solutions from being taken.
- Useless effort in determining what requirements to stick to in unfeasible situations – with the risk of dropping important requirements.

Omission	No requirement about the expected state of train doors in case of emergency stop.
Contradiction	Train doors must always be kept closed between stations.
	And elsewhere:
	Train doors must be opened once a train is stopped after an emergency signal.
Inadequacy	If a book copy has not been returned one week after the third reminder has been issued, the negligent borrower shall be notified that he or she has to pay a fine of £*X*.
	Rather than
	If a book has not been returned one week after the third reminder has been issued, a fine of £*x* shall be retained from the borrower's registration deposit and a notification will be sent to the borrower.
Ambiguity	Train doors shall be opened as soon as the train is stopped at a platform.
	(*Possible interpretations:*)
	The front of the train is (stopped) at a platform *or* The whole train is (stopped) at a platform?
Unmeasurability	Information panels inside trains shall be user-friendly.

Table 1.2 *Errors in a requirements document: Examples*

- Useless effort in the understanding or reverse engineering of unintelligible, poorly defined, poorly structured or poorly traceable aspects – with the risk of wrong understanding or wrong reverse engineering.
- Excessive effort in revising or adapting a poorly modifiable RD – with the risk of incorrect change propagation.

The various types of defect in Table 1.1 may originate from any RE activity – from elicitation to evaluation to documentation to consolidation (see Figure 1.6). Omissions, which are the hardest errors to detect, may happen at any time. Contradictions often originate from conflicting viewpoints that emerged during elicitation and were left unresolved at the end of the RE process. Inadequacies often result from analyst–stakeholder mismatch during elicitation and negotiation. Some flaws are more likely to happen during documentation phases – such as noise, unintelligibility, forward reference and remorse, poor structuring, poor modifiability and opacity.

Overspecifications are frequently introduced in requirements documents written by developers or people who want to jump promptly to technical solutions. They may take the form of

Noise	Every train car will be equipped with a software-controlled information panel together with non-smoking signs posted on every window.
Overspecification	The setAlarm method must be invoked on receipt of a stopAlarm message.
Unfeasibility	The meeting scheduler will also make travel arrangements such as flight, car and hotel reservations for every participant who needs to travel to attend the meeting.
Unintelligibility	A requirement statement containing five acronyms.
Poor structuring	Intertwining of book acquisition and loan management aspects.
Forward reference	Multiple uses of the concept of 'participating in a meeting' in the requirements document and then, several pages later, the definition: A person *participates* in a meeting if he or she attends that meeting from beginning to end.
Remorse	After multiple uses of the undefined concept of 'participating in a meeting', the last one is directly followed by an incidental definition between brackets such as: (a person *participates* in a meeting if he or she attends that meeting from beginning to end).
Poor modifiability	Use of fixed numerical values for quantities throughout the requirements document (e.g. for *maximum loan period, meeting notification deadline* or *train speed thresholds*), when such values are subject to change over time or from one variant to another.
Opacity	A requirement such as: the commanded speed of a train must always be at least 7 mph above its physical speed, without any contextual information about the origin of and rationale for this requirement, and its impact on other requirements.

Table 1.3 *Flaws in a requirements document: Examples*

flowcharts, variables that are internal to the software (rather than shared with the environment, cf. Figure 1.3), statements formulated in terms of programming constructs such as sequential composition, iterations or go-tos. 'Algorithmic requirements' implement declarative requirements that are left implicit. They might incorrectly implement these hidden requirements. They cannot be verified or tested against them. They may preclude some alternative 'implementation' of the hidden requirements that might prove more effective with respect to other quality requirements.

In view of their potentially harmful consequences, requirements errors and flaws should be detected and fixed in the requirements document. Chapter 5 will review a variety of techniques for requirements quality assurance. In particular, Table 1.1 may be used as a basis for requirements inspection checklists (see Section 5.1.3). Model-based quality assurance will be discussed at length in Parts II and III.

1.1.8 Types of software projects

There are different types of projects for which requirements need to be elaborated. As we will see, different project types may entail variations in the RE process discussed in Section 1.1.6.

Greenfield vs brownfield projects

In a *greenfield* project, a brand new software solution is built from scratch to address problems with the system-as-is and exploit new opportunities from technology evolution or market conditions. In a *brownfield* project, the system-as-is already offers software solutions; the software-to-be needs to integrate, improve, adapt or extend such solutions. Note that a greenfield project may become brownfield as the software evolves after deployment. As examples:

- The WAX train transportation system is a greenfield project.
- The UWON library project would be brownfield if we needed to integrate legacy software from some departments.

Greenfield projects are sometimes specialized further into *normal design* vs *radical design* projects (Vicenti, 1993). In a normal design project, engineers solve problems by making improvements to existing technologies or by using them in new ways. They have a good idea of what features the target artefact will provide. In contrast, radical design projects result in fundamentally new technologies. The creators of the target artefact have little idea at the beginning of how this artefact will work and how its components should be arranged. Radical design projects are much less common. They are exploratory by nature.

Customer-driven vs market-driven projects

In a *customer-driven* project, a software solution is developed to address the actual needs of one specific customer in the context of one specific organization. In a *market-driven* project, a software solution is developed to address the potential needs of a whole market segment. There are projects lying between those extremes where the software-to-be is aimed at a specific class of customers within a specific domain. As examples:

- The WAX train transportation system is a customer-driven project.
- The meeting scheduler system is a market-driven project.
- The UWON library project lies somewhere in between, as other universities might be potentially interested in such software to integrate and manage their libraries.

In-house vs outsourced projects

In an *in-house* project, the same company or consortium is carrying out all project phases. In an *outsourced* project, the development is carried out by subcontractors – usually once the project requirements have been established. In general, the contractor is selected by evaluating proposals in response to a call for tenders. There are again projects lying in between, where only specific development phases are being subcontracted. As examples:

- The meeting scheduler is a WSS in-house project.
- The WAX train transportation project is likely to be an outsourced one.

Single-product project vs product-line projects

In a *single-product* project, a single product version is developed for the target customer(s). In a *product-line* project, a product family is developed to cover multiple variants. Each variant customizes the product to a specific class of users or a specific class of items to be managed or controlled. It usually shares commonalities with other variants while differing at specific variation points. Note that a greenfield, single-product project may evolve into a brownfield, product-line one where the single product initially delivered evolves into multiple variants. As examples:

- The WAX train transportation system is a single-product project (at least at inception).
- The meeting scheduler is a product-line project. Variability might refer to the type of customer or the type of meeting.
- If we consider the in-car light-control software for a car manufacturer, variability might refer to different car categories where the software should be installed.

A software project is generally multi-type along the above dimensions. For example, the meeting scheduler might be a greenfield, market-driven, in-house, product-line project.

As far as RE is concerned, these project types have commonalities and differences. On the commonality side, they all need to be based on some form of requirements document at some development stage or another. For example, there is no way of developing a high-quality software product in a brownfield, market-driven, in-house, product-line project without any formulation of the requirements for the software and the assumptions on the environment. Differences from one project type to the other may lie in the following aspects of the RE process:

- *Respective weights of requirements elicitation, evaluation, documentation, consolidation and evolution.* Documentation has been observed to be more prominent in customer-driven projects, whereas prioritization is more prominent in market-driven projects (Lubars *et al.*, 1993). Consolidation is likely to be more prominent in greenfield, customer-driven, mission-critical projects.

- *Use of specific techniques to support RE activities.* For example, greenfield projects may require prototyping techniques for requirements elicitation and risk-based evaluation techniques for decision making (see Chapters 2 and 3). Product-line projects may require feature diagrams for capturing multiple system variants (see Chapter 6).
- *Intertwining between requirements engineering and product design.* In greenfield projects, and in radical design projects in particular, requirements might emerge only once critical design decisions have been made or a product prototype is available.
- *Respective weights of functional and non-functional requirements.* Brownfield projects are often concerned with improving product quality. Non-functional requirements are therefore prominent in such projects.
- *Types of stakeholder involved in the process.* A market-driven project might involve specific types of stakeholder such as technology providers, service providers, retailers, consumers, legislator and the like.
- *Types of developer involved.* The skills required in an outsourced project might be limited to implementation skills, whereas an in-house, greenfield project might require advanced analysis skills.
- *Specific uses of the requirements document.* In an outsourced project, the RD is often used as an annex to the call for tenders, as a reference for evaluating submitted proposals and as a basis for progress monitoring and product evaluation.

1.1.9 Requirements in the software lifecycle

As we saw before, the requirements document is the main product of the RE process. It defines the system-to-be in terms of its objectives, constraints, referenced concepts, responsibility assignments, requirements, assumptions and relevant domain properties. It may also describe system variants and likely evolutions.

Requirements engineering is traditionally considered as the preliminary phase of a software project. The requirements document may indeed be used subsequently in a variety of contexts throughout the software lifecycle. Figure 1.7 summarizes the impact of the requirements document on various software engineering artefacts. The arrows there indicate impact links (which may be bidirectional). Let us briefly review lifecycle activities where the requirements document may be used.

Software prototyping In development processes that integrate a prototyping phase, the requirements already elicited provide input for building an initial prototype or mock-up.

Architectural design A software architecture defines the organization of the software in terms of configurations of components, connectors capturing the interactions among components, and constraints on the components, connectors and configurations (Shaw & Garlan, 1996; Bosch, 2000). The architecture designed must obviously meet the software requirements. In

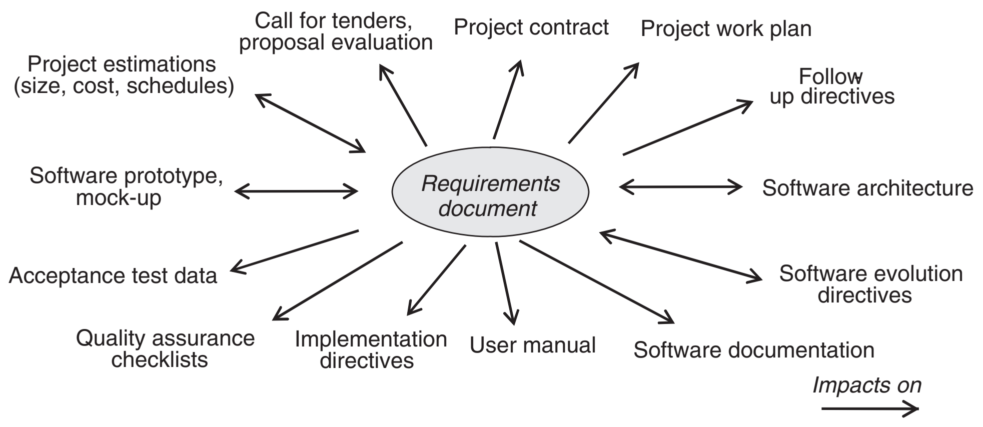
Call for tenders, proposal evaluation
Project contract
Project work plan
Project estimations (size, cost, schedules)
Follow up directives
Software prototype, mock-up
Requirements document
Software architecture
Acceptance test data
Software evolution directives
Quality assurance checklists
Implementation directives
User manual
Software documentation
Impacts on

Figure 1.7 *Requirements in the software lifecycle*

particular, architectural choices may have a deep impact on non-functional requirements (Perry & Wolf, 1992). The requirements document is therefore an essential input for architectural design activities such as:

- The identification of architectural components and connectors.
- Their specification to meet the requirements.
- The selection of appropriate architectural styles.
- The evaluation of architectural options against non-functional requirements.

Software quality assurance The requirements document provides the ultimate reference for quality assurance activities. In particular:

- Requirements provide the basis for elaborating acceptance test data that cover them.
- They are used to define checklists for software inspections and reviews.

Implementation and integration These later steps of the software lifecycle must take non-functional requirements such as interface and installation requirements into account.

Documentation The requirements document, possibly in summarized form, is an important component of the software documentation. Parts of it may be used for writing user manuals.

Maintenance The requirements document, together with problem reports and approved modification requests, provides the input material for revising, adapting, extending or contracting the software product.

Project management The requirements provide a solid basis for project management tasks such as:

- Estimating project size, cost and schedules, e.g. through function points (Low & Jeffery, 1990).
- Planning development activities.
- Writing a call for tenders and evaluating proposals (for outsourced projects).
- Writing the contract linking the developer and the customer.
- Reviewing progress during an incremental development.
- Assessing development team productivity.
- Evaluating the final product.

Many software process models and development methodologies recognize the important role of requirements throughout the software lifecycle. For example, the diagrams summarizing the RUP Unified Process show how the requirements document permeates all project phases from inception to elaboration to construction to transition (Jacobson *et al.*, 1999).

The inevitable intertwining of RE, system design and software architecture design

We might think of RE and design ideally as two completely separate processes coming one after the other in a waterfall-like fashion. This is rarely the case in practice. A complex problem is solved by identifying subproblems, specifying them and solving them, which recursively yields new subproblems (Nilsson, 1971). The recursive nature of problem solving makes the problem and solution spaces intertwined. This applies, in particular, when we elaborate requirements, a corresponding system-to-be and a corresponding software architecture.

Such intertwining occurs at places where we need to make decisions among alternative options based on quality requirements, in particular:

- When we have elicited a system objective and want to decompose it into sub-objectives – different decompositions might be envisioned, and we need to select a preferred one.
- When we have identified a likely and critical risk – different countermeasures might be envisioned, and we need to select a preferred one.
- When we have detected a conflict between requirements and want to resolve it – different resolutions might be envisioned, and we need to select a preferred one.
- When we realize a system objective through a combination of functional services, constraints and assumptions – different combinations might be envisioned, and we need to select a preferred one.

- When we consider alternative assignments of responsibilities among components of the system-to-be – a more suitable one must eventually be selected. In this process, we might consider alternative component granularities as well.

All these situations involve *system* design decisions. Once such a decision has been made, we need to recursively elicit, evaluate, document and consolidate new requirements and assumptions based on it. Different decisions may result in different proposals for the system-to-be, which, in turn, are likely to result in different *software* architectures. Conversely, while elaborating the software architecture we might discover new requirements or assumptions that had been overlooked thus far.

Let us illustrate this intertwining of RE and design in our case studies.

In the meeting scheduler, the objective of knowing the constraints of invited participants might be decomposed into a sub-objective of knowing them through e-mail requests or, alternatively, a sub-objective of knowing them through access to their electronic agenda. The architecture of a meeting scheduler based on e-mail communication for getting constraints will be different in places from one based on e-agendas. Likewise, there will be architectural differences between an alternative where meeting initiators are taking responsibility for handling constraint requests and a more automated version where a software component is responsible for this.

In our train control system, the computation of train accelerations and the transmission of acceleration commands to trains might be under responsibility of software components located at specific stations. Alternatively, this responsibility might be assigned, for the acceleration of a specific train, to the on-board software of the train preceding it. These are *system* design options that we need to evaluate while engineering the system requirements, so that preferred options can be selected for further requirements elaboration. Those two alternatives result in very different software architectures – a semi-centralized architecture and a fully distributed one. The alternative with an ultra-reliable component at specific stations is likely to be selected in order to better meet safety requirements.

1.1.10 The relationship of requirements engineering to other disciplines

Section 1.1.3 discussed how wide the scope of RE is. Section 1.1.6 showed how diverse its activities and actors are. It is therefore not surprising that some areas of RE are connected to other disciplines and research communities.

The discipline of RE has a primary interaction with, of course, *software engineering* (SE). As frequently said, the former is about getting the right system whereas the latter is about getting the software right in this system. The previous section discussed numerous interactions between RE and SE processes and products. In addition, RE benefits from SE technology for designing tools to support its activities – such as smart editors, prototyping tools, analysers, documentation tools, configuration managers and the like.

There are other disciplines to which RE is connected. We just mention the connections here; they will appear more clearly as we discuss RE techniques, by activity, in Chapters 2 to 6.

Domain understanding and requirements elicitation

Analysing system objectives, tasks and roles is a concern shared with *systems engineering*, sometimes called systems analysis in the business application domain.

For control systems, *control theory* provides techniques for modelling controllers and controlled processes. These techniques can be used to analyse the environment and the interactions that the software should have with it.

The system-as-is and the system-to-be are generally grounded within an organization. The structure, business objectives, policies and operational procedures of this organization need to be understood and analysed. Effective techniques for doing so are found in *management science* and *organization theory*.

The quality of communication between requirements engineers and stakeholders is a necessary condition for the effectiveness of domain understanding and requirements elicitation. Principles and guidelines can be borrowed from *behavioural psychology* here. Some of the techniques used for elicitation originate in *sociological theories* of human groups; others are grounded on ethnographical principles from *anthropology* (see Section 2.3.2).

Requirements elicitation amounts to a form of knowledge acquisition for which techniques have been developed in *artificial intelligence*.

The elicitation and analysis of specific categories of non-functional requirements may be supported by dedicated techniques found in other disciplines, for example *reliability theory* for safety requirements, *security engineering* for security requirements, probabilistic *performance evaluation* for performance requirements, *cognitive psychology* and *human-computer interaction* (HCI) for useability requirements. In particular, the HCI literature contains a significant number of dedicated techniques for eliciting, evaluating, specifying and validating user interface requirements based on user models and task models.

Requirements evaluation and agreement

The assessment of alternative options against qualities and risks is a general issue addressed by *decision theory* and the literature on *risk management*. Specific techniques such as multicriteria analysis are highly relevant in this context.

Management science also provides principles and theories on the art of *negotiation* and *conflict management*. These issues are addressed from a different perspective in *artificial intelligence* in the context of multi-agent planning.

Requirements specification, documentation and consolidation

There is some partial overlap between the techniques available to support these activities and the languages, structuring mechanisms and analysis tools found in the software engineering literature on *software specification*. Some of the software design notations can be lifted up to RE, including a subset of the UML, as we will see in Part II. *Formal methods* provide technological solutions for analysing requirements when the latter are available in fully formalized, machine-processable form. They will be introduced in Sections 4.4 and 5.4 and further discussed in Chapters 17 and 18.

Requirements evolution

From a managerial perspective, this area intersects with the area of *change management* in management science. From a technical perspective, it intersects with the area of version control and *configuration management* in software engineering.

Modelling as a transversal activity

Parts II and III of this book will present a model-driven approach to requirements engineering where multifaceted models of the system-as-is and the system-to-be are built as a common interface to the various RE activities. Requirements modelling is connected to other disciplines in computing science where models are used, notably:

- Conceptual models in *databases* and *management information systems*.
- Task models in *human–computer interaction*.
- Models for representing domain knowledge and structuring problem spaces in *artificial intelligence*.

1.2 Why engineer requirements?

Now that we have a better idea of what RE is about, we could ask ourselves whether it is worth the effort. This section discusses why and to what extent the engineering of high-quality requirements is an essential precondition for the success of a software project. We first review some citations and facts that provide anecdotal evidence about the importance of RE and the consequences of poor RE. Then we discuss the role and critical impact of RE in the software lifecycle from a more general perspective.

1.2.1 Facts, data and citations about the requirements problem

The phrase 'requirements problem' refers to software project failures that have been attributed to poor or non-existent requirements.

An old problem

The requirements problem is among the oldest in software engineering. An early empirical study of a variety of software projects revealed that incomplete, inadequate, inconsistent or ambiguous requirements are numerous and have a critical impact on the quality of the resulting software (Bell & Thayer, 1976). These authors concluded that 'the requirements for a system do not arise naturally; instead, they need to be engineered and have continuing review and revision'. This was probably the first reference to the phrase 'requirements engineering', suggesting the need for systematic, repeatable procedures for building high-quality artefacts.

A consensus has been rapidly growing that such engineering is difficult. As Brooks noted in his landmark paper on the essence and accidents of software engineering, 'the hardest single part of building a sofware system is deciding precisely what to build ... Therefore, the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements' (Brooks, 1987).

Requirements errors are the most expensive software errors

Lots of time and money can be saved if requirements errors and flaws are detected and fixed at the RE stage rather than later. Boehm and Papaccio reported that it costs 5 times more to detect and fix requirements defects during design, 10 times more during implementation, 20 times more during unit testing and up to 200 times more after system delivery (Boehm & Papaccio, 1988).

Requirements errors are numerous and persistent

Requirements errors are not only usually costly; they are numerous and persistent over the software lifecycle. Jones states that US companies average one requirements error per function point (Jones, 1995). According to an earlier study, for management information systems 55% of software faults can be traced to the requirements and design phases; the figure is 50% for military software and 45% for systems software. The overall figure, weighted by occurrences, is 52%, with 25% from the requirements phase and 27% from the design phase (Jones, 1991). In her study of software errors in NASA Voyager and Galileo programs, Lutz consistently reported that the primary cause of safety-related faults was errors in functional and interface requirements (Lutz, 1993).

Other studies have confirmed the requirements problem on a much larger scale. A survey over 8000 projects undertaken by 350 US companies suggested that only 16% of them were considered to be successful; 33% of them failed without having ever been completed; and 51% succeeded only partially; that is, with partial functionalities, major cost overruns and significant delays (Standish Group, 1995). When asked about the main reasons for this, about 50% of the project managers identified requirements-related problems as the primary cause; specifically, the lack of user involvement (13%), requirements incompleteness (13%), changing requirements (11%), unrealistic expectations (9%) and unclear objectives (5%). An independent survey of 3800 European organizations in 17 countries led to parallel conclusions. When asked where their main software problems were, more than half of the managers ranked requirements specification and management in first position (Ibanez & Rempp, 1996).

The requirements problem has been echoed in various business reports about the lack of alignment between business problems and IT solutions. An Accenture study in 2003 pointed out the mismatch between IT investments and business objectives. A Giga Group report in 2001 consistently recommended that IT projects be prioritized according to their contribution to business objectives. Another report by Meta Group in 2003 claimed that 60–70% of IT project failures are to be attributed to poor requirements gathering, analysis and management.

Other studies led independently to the consistent conclusion that many software problems and failures are to be attributed to poor RE, for example Lyytinen and Hirscheim, 1987 and Jones, 1996.

Requirements errors are the most dangerous software errors

The requirements problem gets even worse in the case of mission-critical systems such as safety-critical or security-critical systems. Many harmful and sometimes tragic software failures were recognized to be traceable back to defective requirements (Leveson, 1995; Neumann, 1995; Knight, 2002).

For example, 290 people were killed when a civil IranAir A300 Airbus was confused with a hostile F-14 aircraft and shot by the US *Vincennes* warship in July 1988. Investigations revealed that the origins of this tragedy were a mix of threatening conditions and missing requirements on the AEGIS combat software. Some timing requirements for successive input events to be threatening were missing. Furthermore, critical information on aircraft displays to allow pilots to assess threats correctly was missing, such as current altitude and ascending/descending mode of 'target' aircraft (US Department of Defense, 1988).

Neumann reports on several cases in the London underground system where people were killed due to doors opening or closing in unexpected circumstances without alarm notification to the train driver (Neumann, 1995).

As noted before, omissions and inadequacies do not refer to requirements only. Many reported problems originate in missing, inadequate, inaccurate or changing assumptions and properties about the environment in which the software operates. An early study of software engineering practice already made that point (Curtis *et al.*, 1988). Sadly enough, its conclusions remained valid. Let us first mention a few cases of inadequate assumptions or domain properties.

The first version of the London ambulance despatching system was based on a series of assumptions about the environment, for example that radio communication would work in all circumstances, that the ambulance localization system would always work properly, that crews would always select the ambulance being allocated to them by the software, that they would always go to the incident assigned to them by the software, that they would always press ambulance availability buttons correctly and when needed, that no incident data could be lost and so forth. The two tragic failures of this system from October to November 1992 resulted from a combination of circumstances that violated many such assumptions (LAS, 1993).

Hooks and Farry mention an aerospace project where 49% of requirements errors were due to incorrect facts about the problem world (Hooks & Farry, 2000).

An inadequate assumption about the environment of the flight guidance system may have contributed to the tragic crash of an American Airlines Boeing 757 in Cali (Colombia) in December 1995 (Modugno *et al.*, 1997). The information about the point in space where the pilot was expected to initiate the flap extension was assumed to arrive before the plane actually reached that point in space. The aircraft landing in Cali had already passed that point, which resulted in the guidance software ordering the plane to turn around towards a mountain.

Domain properties, used explicitly or implicitly for elaborating requirements, may be wrong as well. A famous example is the Lufthansa A320 Airbus flight to Warsaw, in which the plane ran off the end of the runway, resulting in injuries and loss of life. The reverse thrust was disabled for up to nine seconds after landing on a waterlogged runway (Ladkin, 1995). In terms of the satisfaction argument discussed in Section 1.1.4, the problem might be recollected in simplified form as follows (Jackson, 1995a). The autopilot had the system requirement that reverse thrust be enabled if and only if the plane is moving on the runway:

(*SysReq*:) ReverseThrustEnabled $\leftrightarrow$ MovingOnRunway

The software requirement given to developers in terms of software input/output variables was:

(*SofReq*:) reverse = 'on' $\leftrightarrow$ WheelPulses = 'on'

An argument that this software requirement entails the corresponding system requirement had to rely on assumptions on the wheels sensor and reverse thrust actuator, respectively:

(*Asm*:) WheelPulses = 'on' $\leftrightarrow$ WheelsTurning
reverse = 'on' $\leftrightarrow$ ReverseThrustEnabled,

together with the following domain property:

(*Dom*:) MovingOnRunway $\leftrightarrow$ WheelsTurning

This domain property proved to be inadequate on the waterlogged Warsaw runway. Due to aquaplaning, the plane there was moving on the runway without wheels turning.

A similar case occurred recently where a car driver was run over by his luxurious computerized car while opening a gate in front of it. The software controlling the handbrake release had the system requirement:

'The handbrake shall be released if and only if the driver wants to start.'

The software requirement was:

'The handbrake control shall be ''off'' if and only if the normal running of the motor is raised.'

The assumption that

'The driver wants to start if and only if he presses the acceleration pedal'

is adequate; but the domain property stating:

'The normal running of the motor is raised if and only if the acceleration pedal is pressed'

proved to be inadequate on a hot summer's day. The car's air conditioner started automatically, due to the car's door being open while the driver was opening the gate in front, which resulted in the normal running of the motor being raised and the handbrake being released.

In addition to cases of wrong assumptions or wrong domain properties, there are cases where failure originates from environmental *changes* that render the original assumptions no longer adequate. A concrete example showing the problems with changing environments, in the context of our train control case study, is the June 1995 New York subway crash. The investigation revealed that the distance between signals was shorter than the worst-case stopping distance of trains; the assumption that a train could stop in the space allowed after the signal was adequate for 1918 trains but inadequate for the faster, longer and heavier trains running in 1995 (16 June 1995 *New York Times* report, cited in Hammond *et al.*, 2001).

The well-known *Ariane 5* rocket failure is another example where environmental assumptions, set for requirements satisfaction, were no longer valid. Software components were reused from the *Ariane 4* rocket with ranges of input values that were different from the expected ones due to changes in rocket features (Lions, 1996). In the same vein, the Patriot anti-missile system that hit US military barracks during the first Gulf War had been used for more than 100 hours. The system was assuming missions of 14 hours at most (Neumann, 1995).

Missing or inadequate requirements/assumptions may have harmful consequences in security-critical systems as well. For example, a Web banking service was reported to have no adequate requirements about how the software should behave when a malicious user is searching for all bank accounts that match some given 4-digit PIN number (dos Santos *et al.*, 2000).

As we will see in Chapter 5, there are fortunately techniques for spotting errors in requirements and assumptions. For example, such techniques uncovered several dangerous omissions and ambiguities in TCAS II, a widely used aircraft collision-avoidance system (Heimdahl & Leveson, 1996). This important topic will be covered in depth in Chapters 5, 9, 16 and 18.

1.2.2 The role and stakes of requirements engineering

The bottom line of the previous section is that engineering high-quality requirements is essential, as errors in requirements, assumptions and domain properties tend to be numerous, persistent, costly and dangerous. To support that conclusion, we may also observe the prominent role that RE plays with respect to multiple stakes.

Technical stakes As we saw in Section 1.1.9, the requirements document (RD) provides a basis for:

- Deriving acceptance test data.
- Designing the software architecture and specifying its components/connectors.
- Defining quality-assurance checklists.
- Writing the documentation and user manuals.
- Handling requests for software evolution.

Communication stakes The RD provides the main reference through which the various parties involved in a software project can communicate with each other.

Project management stakes The RD provides a basis for determining the project costs, required resources, development steps, milestones, review points and delivery schedules.

Legal stakes The RD forms the core of the contract linking the software provider, customers and subcontractors (if any).

Certification stakes Quality norms are increasingly enforced by law or regulations on projects in specific domains such as medical, transportation, aerospace or nuclear. They may also be requested by specific customers in other domains. Such norms constrain the development process and products. At the *process* level, maturity models such as CMMI, SPICE or ISO9001 require RE to be taken seriously. For example, CMMI Maturity Level 2 imposes a requirements management activity as a necessary condition for process repeatability; Level 3 requires a repeatable requirements development process (Ahern *et al.*, 2003). At the *product* level, standards such as IEEE-STD-830 or ESA PSS-05 impose a fairly elaborate structure on the requirements document (see Section 4.2.2).

Economic stakes The consequences of numerous, persistent and dangerous errors related to requirements can be economically devastating.

Social stakes When not sufficiently user centred, the RE process may overlook important needs and constraints. This may cause severe deteriorations in working conditions, and a wide range of reactions from partial or diverted use of the software to mere rejection of it. Such reactions may have severe consequences beyond user dissatisfaction. For example, in the London ambulance system mentioned in the previous section, misuse and rejection of the new system by ambulance drivers were reported to be among the main causes of failure.

1.3 Obstacles to good requirements engineering practice

Many practitioners have heard about the requirements problem and may have experienced it. The critical role of RE in the success of a software project is widely recognized. Process maturity models promote spending effort in RE activities. A recent large-scale study has confirmed that almost any project includes some RE activity, whatever its type and size (Jones, 2003).

In spite of all this, the current state of RE practice is still, by and large, fairly limited in terms of effort spent on this activity and technology used to support it (Glass, 2003). Practitioners are in a sense like cigarette smokers who know that smoking is pretty unhealthy but keep smoking. The reasons for this may be in the following *obstacles* to spending effort and money in the RE process:

- Such effort generally needs to be spent before the project contract is signed, without a guarantee that a contract will be signed.

- There might be stronger concerns and pressure on tight schedules, short-term costs and catching up on the latest technology advances.
- Too little research work has been devoted to RE economics. On one hand, the benefits and cost saving from using RE technology have not been quantified. They are hard to measure and not enough evidence has been gained from large-scale empirical studies. On the other hand, progress in RE activities is harder to measure than in design or implementation activities.
- Practitioners sometimes feel that the requirements document is exceedingly big and complex (Lethbridge *et al*., 2003). In such cases it might not be maintained as the project evolves, and an outdated document is no longer of any use.
- The requirements document may be felt to be too far away from the executable product for which the customer is paying. In fact, the quality of requirements does not indicate much about the quality of the executable product.
- RE technology is sometimes felt to be too heavyweight by some practitioners, and too vague by others.
- Beyond general guidelines, the transfer of effective RE techniques through courses, textbooks and pilot studies has been much more limited than in other areas of software engineering.

We need to be aware of such obstacles to find ways of overcoming them. Chapters 2–6 will review standard techniques to support the RE process more effectively. In this framework, the next parts of the book will detail a systematic method for building a multifaceted system model from which a well-structured requirements document can be generated. This method will make the elicitation, evaluation, documentation, consolidation and evolution efforts more focused and more effective.

1.4 Agile development processes and requirements engineering

More agility in the RE process might address some of the previously mentioned obstacles in some software projects.

Agile processes are aimed at *early* and *continuous* provision of functionality of value to the customer by reducing both the RE effort and the requirements-to-code distance.

To achieve this, the spiral RE process in Figure 1.6 iterates on very short cycles, where each cycle is directly followed by a short implementation cycle:

- A RE cycle is shortened by eliciting some useful functional increment directly from the user, and by shortcutting the evaluation, specification and consolidation phases; or by making these very rudimentary to expedite them. For example, the specification phase may amount to the definition of test cases that the implementation must pass.
- The implementation cycle next to a RE cycle is shortened as (a) the functional increment from this RE cycle is expected to be small; and (b) this increment is implemented by a

small team of programmers working at the same location, following strict programming rules, doing their own unit testing and staying close to the user to get instant feedback for the next RE cycle.

The functional increment elicited at a RE cycle is sometimes called *user story*. It captures some unit of functionality of direct value that the user can write and deliver easily to the programming team.

Agile processes have emerged in certain development communities and projects as a reaction against overly heavyweight practices, sometimes resulting from the misinterpretation of process models and the amount of 'ceremony' and reporting they require. However, it is important to highlight the underlying *assumptions* that a project must fulfil for an agile process to work successfully. Such assumptions delimit the applicability of agile processes:

- All stakeholder roles, including the customer and user roles, can be reduced to one single role.
- The project is sufficiently small to be assignable to a single, small-size, single-location development team.
- The user can be made available at the development site or can interact promptly and effectively.
- The project is sufficiently simple and non-critical to disregard or give little consideration to non-functional aspects, environmental assumptions, underlying objectives, alternative options and risks.
- The user can provide functional increments quickly, consistently (so that no conflict management is required) and gradually from essential to less important requirements (so that no prioritization is required).
- The project requires little documentation for work coordination and subsequent product maintenance. Precise requirements specification before coding is not an issue.
- Requirements verification before coding is less important than early release.
- New or changing requirements are not likely to require major code refactoring and rewrite, and the people in charge of product maintenance are likely to be the product developers.

These assumptions are quite strong. Many projects obviously do not meet them all, if any – in particular, projects for mission-critical systems. We would obviously not like our air traffic control, transportation, power plant, medical operation or e-banking systems to be obtained through agile development of critical parts of the software.

Agility is not a binary notion, however. Depending on which of the preceding assumptions can be fulfilled and which cannot, we can achieve more or less agility by paying more or less attention to the elicitation, evaluation, specification and consolidation phases of an RE cycle, making it longer or shorter.

From this perspective, the approach discussed in Parts II and III is intended to make RE cycles shorter by:

- Supporting functional goals and scenarios as units of value to stakeholders.
- Focusing on declarative formulations for incremental elaboration, and incremental analysis only when and where needed.
- Providing constructive guidance in model-based RE through a variety of heuristics and patterns.
- Integrating tool support for effort reduction by elimination of clerical work.

Summary

- The focus of RE is the investigation, delineation and precise definition of the *problem world* that a machine solution is intended to improve. The scope of investigation is broad. It involves two system versions. Next to the system-as-is, the system-to-be comprises the software to be developed and its environment. The latter may comprise people playing specific roles, physical devices operating under physical laws, and pre-existing software. The questions to be addressed about the system-to-be include WHY, WHAT, HOW WELL and WHO questions. Such questions can be answered in a variety of ways, leading to a range of alternative options to consider, each having associated strengths and risks.
- Requirements engineers are faced with multiple transitions to handle: from the problem world to the machine interface with it; from a partial set of conflicting concerns to a complete set of consistent statements; from imprecise formulations to precise specifications; from unstructured material to a structured document; from informal wishes to a contractual document. There are multiple levels of abstraction to consider, with strategic objectives at the top and technical requirements at the bottom. Multiple abstraction levels call for satisfaction arguments, as we need to show that the higher-level concerns are satisfied by the lower-level ones.
- The RE process is an iteration of intertwined activities for eliciting, evaluating, documenting, consolidating and changing the objectives, functionalities, assumptions, qualities and constraints that the system-to-be should meet based on the opportunities and capabilities provided by new technologies. Those activities involve multiple stakeholders that may have conflicting interests. The relative weight of each activity may depend on the type of project.
- The RE process involves different types of statements. Requirements are prescriptive statements about software functionalities, qualities and development constraints. They are expressed in the vocabulary of the problem world. Domain properties are

descriptive statements about this world. Assumptions are statements about expected behaviours of environmental components. We need to make appropriate assumptions and identify correct domain properties to elaborate the right requirements.

- These different types of statements have to be specified and structured in the requirements document. Their specification must meet multiple qualities, among which completeness and adequacy are most critical. The requirements document is a core artefact in the software lifecycle, as many software engineering activities rely on it. Its quality has a strong impact on the software project – notably, its successful completion, the development and maintenance costs, the rate of user acceptance and satisfaction, system security and safety. Studies on the requirements problem have consistently shown that requirements errors are numerous, persistent, costly and dangerous. Wrong hidden assumptions can be the source of major problems.
- There are a few misconceptions and confusions about RE to avoid:

a. The target of investigation is not the software but a system of which the software is one component.

b. RE does not amount to some translation of pre-existing problem formulations.

c. RE and design are not sequentially composed in a waterfall-like fashion. RE involves system design. In view of the alternative options arising in the RE process, we need to make decisions that may subsequently influence software design. Conversely, some requirements might sometimes emerge only in the later stages of software design.

d. Unlike domain properties, requirements may need to be negotiated, weakened or changed.

e. 'Precise' does not mean 'formal'. Every statement must have a unique, accurate interpretation without necessarily being machine processable.

f. A set of notations may be a necessary condition for a RE method but certainly not a sufficient one. A method should provide systematic guidance for building complex requirements documents.

Notes and Further Reading

The grounding of machine requirements in the problem world is amply discussed in Jackson (1995b). One of the first characterizations of RE as investigation of WHY, WHAT and HOW issues appeared in Ross and Schoman (1977a). This seminal paper emphasized the importance of analysing the contextual objectives that the system-to-be needs to address. It introduced viewpoints as a composition mechanism for RE. Twenty years later, Zave consistently argued that the relationship between objectives, functionalities, constraints and software requirements is a key aspect of the RE process (Zave, 1997). Requirements evolution along variants and revisions is also discussed there.

The important distinction between descriptive and prescriptive statements appeared first in Jackson & Zave (1993) and was echoed in Jackson (1995a) and Zave and Jackson (1997). The differentiation between system requirements and software requirements is discussed in Jackson (1995a), where the latter are called 'specifications'. Similar distinctions were made in the more explicit setting of the four-variable model in Parnas and Madey (1995).

Satisfaction arguments have been known for a long time in programming methodology. When we build a program P in some environment E the program has to satisfy its specification S. Therefore we need to argue that $P, E \models S$. Such argumentation was first lifted up to the RE phase in Yue (1987). The need for satisfaction arguments at RE time is discussed in Jackson (1995a) and convincingly illustrated in Hammond *et al.* (2001) in the context of the REVEAL methodology for requirements engineering. Such arguments were made explicit in terms of goal refinement and goal operationalization in Dardenne *et al.* (1991), Dardenne *et al.* (1993) and van Lamsweerde (2000b).

The spiral model of software development is described in Boehm (1988). An adaptation to requirements development was suggested first in Kotonya & Sommerville (1997). Agile processes in the context of RE are briefly introduced in Leffingwell and Widrig (2003). The need for early delivery of useful subsystems was recognized in Parnas (1979).

Numerous books and papers propose requirements taxonomies, notably Thayer and Dorfman (1990), Davis (1993), Robertson and Robertson (1999) and Chung *et al.* (2000).

A thorough discussion of specification errors will be found in Meyer's paper on the specifier's "seven sins" (Meyer, 1985). Those 'sins' are illustrated there on a published specification of a text formatting problem, where most defects are found in a few lines! Yue was probably the first to define requirements completeness and pertinence with respect to underlying objectives (Yue, 1987). The best discussion on requirements measurability is in Robertson and Robertson (1999), which proposes so-called fit criteria as a way of checking whether a requirement is measurable (we come back to this in Sections 4.2 and 5.1). Some of the qualities expected for a requirements document are also presented in Davis (1993).

The distinction between customer-specific and market-driven projects is discussed from an RE perspective in Lubars *et al.* (1993). Radical design projects are contrasted with normal design ones from an engineering perspective in Vicenti (1993).

The view of RE as a composite system design activity is elaborated technically in Feather (1987) and Fickas and Helm (1992). The inevitable intertwining of RE and architectural design is argued in Nuseibeh (2001). To some extent it is a transposition, to the earlier phases of the software lifecycle, of an argument made before for specification and implementation (Swartout & Balzer, 1982).

Stories and analyses of poor RE in mission-critical systems can be found in Leveson (1995) and Neumann (1995). For regular updates check the RISKS Digest Forum Web site, moderated by P. G. Neumann under the auspices of the ACM Committee on Computers and Public Policy.

An obvious indication of the vitality of RE as an autonomous discipline is the number of introductory textbooks on the subject. They cover general insights, principles, guidelines and modelling notations for RE. A sample of these includes Gause and Weinberg (1989). Davis (1993), Loucopoulos and Karakostas (1995), Kotonya and Sommerville (1997), Kovitz (1999), Robertson and Robertson (1999), Maciaszek (2001), Lauesen (2002) and Bray (2003). Among these, Davis (1993) contains an extensive annotated bibliography on early work. A good overview of requirements capture within the software lifecycle is provided in Pfleeger (2001).

The running case studies in this book have some origins in the literature. The library system significantly expands on a toy specification benchmark (Wing, 1988), to address RE issues based on a real library system in a large university environment. The train control system is partially inspired by the BART system (Winter *et al.*, 1999), the old-time McGean train system (Feather, 1994) and the author's experience of missing flight connections in large airports due to poor local transportation facilities. The meeting scheduling system expands on an earlier version (van Lamsweerde *et al.*, 1993), partially published in Feather *et al.* (1998), based on personal experience in organizing international meetings.

Exercises

- Consider the library world suggested by the case study description in Section 1.1.2. Draw a world-and-machine diagram similar to Figure 1.1 showing where the following phenomena are located: BookCopyReturned (the return of a book by a patron); ReturnEncoded (the encoding of a returned book by library staff at a terminal); LoanRecordUpdated (the corresponding database update); BookCopyInShelves (the physical availability of a book copy in library shelves); BookAvailabilityDisplayed (the displaying at a terminal of a book's availability status); BookCoversThisTopic (the fact that a book covers such or such topic); BookKeywordsEncoded; DatabaseSearched; and QueryAnswerDisplayed.
- Distribute the list of complaints, reported in the case study description of the library system in Section 1.1.2, among different viewpoints associated with specific stakeholders.
- From the case study description of the library system, elaborate alternative options to meet the objective of reduced book stealing at UWON. Assess each alternative against its risks.

- Consider the case study description of the train control system. Discuss the respective strengths and risks associated with alternative responsibility assignments of the prescriptive statement *TrainDoorsClosedWhileMoving*; namely, to the train driver, to a dedicated clerk, to passengers or to an on-board software controller.
- Identify a sample of strategic objectives, functional services and environmental assumptions from the case study descriptions of the library, train control and meeting scheduling systems.
- Draw a diagram similar to Figure 1.3 for the meeting scheduling case study. Make your diagram more precise by depicting a four-variable model of it.
- Repeat the previous exercise on your favourite cashpoint machine (ATM).
- Provide a sample of descriptive statements and a sample of prescriptive statements from the case study descriptions of the library, train control and meeting scheduling systems, respectively.
- Consider a simple traffic light system to regulate safe pedestrian crossing on a busy lane. Consider the following system requirement:

  (*SysReq*:) The traffic lights shall allow pedestrians to safely cross the lane by stopping cars

  together with the following software requirements:

  (*SofReq1*:) The light switch for pedestrians will be set to 'green' within *x* seconds after the pedestrian button has been pressed.

  (*SofReq2*:) The light switch for cars will be set to 'red' at least *y* seconds before the light switch for pedestrians is set to 'green'.

  Find missing environment assumptions and domain properties that are necessary to build the following satisfaction argument:

  $$\{SofReq1,\ SofReq2,\ \text{assumptions?},\ \text{domain properties?}\} \models SysReq$$

  Are the missing domain properties adequate? Are the missing assumptions realistic?
- Find out where such a satisfaction argument fails in the car handbrake control story reported in Section 1.2.1.
- Section 1.1.4 gives a few examples of non-functional requirements for our running case studies. Identify additional non-functional requirements mentioned in the case study descriptions, and classify them according to the taxonomy in Figure 1.5. For example, to what class does the following requirement belong?

  The meeting date and location should be notified to participants *x* weeks before the meeting at latest.

- Extend the case study descriptions in Section 1.1.2 with other non-functional requirements that might be worth considering. To elicit them, browse through the requirements taxonomy in Figure 1.5 and look for system-specific instances of the various categories.
- Find or invent other examples of requirement/assumption defects in the case study descriptions, in addition to those in Tables 1.2 and 1.3.
- Imagine that you need to convince your manager that the project budget has to cover *X* person-months for the RE task. Prepare a full argument for this.