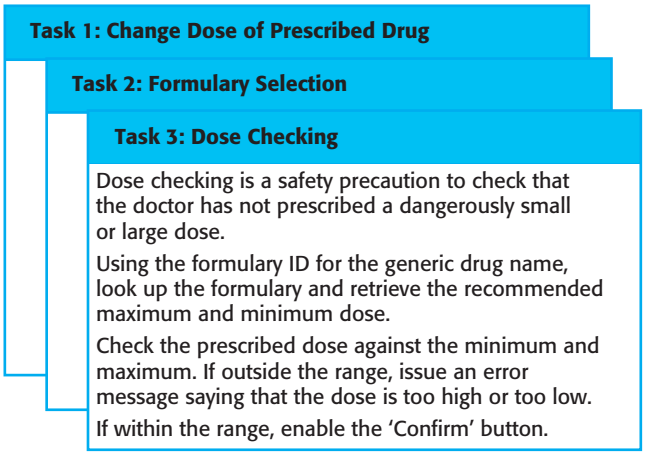


Figure 3.6 Examples of task cards for prescribing medication.

A general problem with incremental development is that it tends to degrade the software structure, so changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system.

Extreme programming tackles this problem by suggesting that the software should be constantly refactored. This means that the programming team look for possible improvements to the software and implement them immediately. When a team member sees code that can be improved, they make these improvements even in situations where there is no immediate need for them. Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of code with calls to methods defined in a program library. Program development environments, such as Eclipse (Carlson, 2005), include tools for refactoring which simplify the process of finding dependencies between code sections and making global code modifications.

In principle then, the software should always be easy to understand and change as new stories are implemented. In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require the architecture of the system to be modified.

In practice, many companies that have adopted XP do not use all of the extreme programming practices listed in Figure 3.4. They pick and choose according to their local ways of working. For example, some companies find pair programming helpful; others prefer to use individual programming and reviews. To accommodate different levels of skill, some programmers don't do refactoring in parts of the system they did not develop, and conventional requirements may be used rather than user stories. However, most companies who have adopted an XP variant use small releases, test-first development, and continuous integration.

3.3.1 Testing in XP

As I discussed in the introduction to this chapter, one of the important differences between incremental development and plan-driven development is in the way that the system is tested. With incremental development, there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to incremental development have a very informal testing process, in comparison with plan-driven testing.

To avoid some of the problems of testing and system validation, XP emphasizes the importance of program testing. XP includes an approach to testing that reduces the chances of introducing undiscovered errors into the current version of the system.

The key features of testing in XP are:

1. Test-first development,
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.

Test-first development is one of the most important innovations in XP. Instead of writing some code and then writing tests for that code, you write the tests before you write the code. This means that you can run the test as the code is being written and discover problems during development.

Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. This approach can be adopted in any process in which there is a clear relationship between a system requirement and the code implementing that requirement. In XP, you can always see this link because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation. The adoption of test-first development in XP has led to more general test-driven approaches to development (Astels, 2003). I discuss these in Chapter 8.

In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of 'test-lag'. This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

User requirements in XP are expressed as scenarios or stories and the user prioritizes these for development. The development team assesses each scenario and breaks it down into tasks. For example, some of the task cards developed from the story card for prescribing medication (Figure 3.5) are shown in Figure 3.6. Each task generates one or more unit tests that check the implementation described in that task. Figure 3.7 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

Test 4: Dose Checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose × frequency is too high and too low.
4. Test for inputs where single dose × frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Figure 3.7 Test case description for dose checking

The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system. As I discuss in Chapter 8, acceptance testing is the process where the system is tested using customer data to check that it meets the customer's real needs.

In XP, acceptance testing, like development, is incremental. The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs. For the story in Figure 3.5, the acceptance test would involve scenarios where (a) the dose of a drug was changed, (b) a new drug was selected, and (c) the formulary was used to find a drug. In practice, a series of acceptance tests rather than a single test are normally required.

Relying on the customer to support acceptance test development is sometimes a major difficulty in the XP testing process. People adopting the customer role have very limited available time and may not be able to work full-time with the development team. The customer may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test automation is essential for test-first development. Tests are written as executable components before the task is implemented. These testing components should be standalone, should simulate the submission of input to be tested, and should check that the result meets the output specification. An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution. Junit (Massol and Husted, 2003) is a widely used example of an automated testing framework.

As testing is automated, there is always a set of tests that can be quickly and easily executed. Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test-first development and automated testing usually results in a large number of tests being written and executed. However, this approach does not necessarily lead to thorough program testing. There are three reasons for this:

1. Programmers prefer programming to testing and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.

2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.

3. It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage. Crucial parts of the system may not be executed and so remain untested.

Therefore, although a large set of frequently executed tests may give the impression that the system is complete and correct, this may not be the case. If the tests are not reviewed and further tests written after development, then undetected bugs may be delivered in the system release.

3.3.2 Pair programming

Another innovative practice that has been introduced in XP is that programmers work in pairs to develop the software. They actually sit together at the same workstation to develop the software. However, the same pairs do not always program together. Rather, pairs are created dynamically so that all team members work with each other during the development process.

The use of pair programming has a number of advantages:

1. It supports the idea of collective ownership and responsibility for the system. This reflects Weinberg's (1971) idea of egoless programming where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.

2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews (covered in Chapter 24) are very successful in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. Although pair programming is a less formal process that probably doesn't find as many errors as code inspections, it is a much cheaper inspection process than formal program inspections.

3. It helps support refactoring, which is a process of software improvement. The difficulty of implementing this in a normal development environment is that effort in refactoring is expended for long-term benefit. An individual who practices refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as

two individuals working alone. There have been various studies of the productivity of paid programmers with mixed results. Using student volunteers, Williams and her collaborators (Cockburn and Williams, 2001; Williams et al., 2000) found that productivity with pair programming seems to be comparable with that of two people working independently. The reasons suggested are that pairs discuss the software before development so probably have fewer false starts and less rework. Furthermore, the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

However, studies with more experienced programmers (Arisholm et al., 2007; Parrish et al., 2004) did not replicate these results. They found that there was a significant loss of productivity compared with two programmers working alone. There were some quality benefits but these did not fully compensate for the pair-programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.

3.4 Agile project management

The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project. They supervise the work of software engineers and monitor how well the software development is progressing.

The standard approach to project management is plan-driven. As I discuss in Chapter 23, managers draw up a plan for the project showing what should be delivered, when it should be delivered, and who will work on the development of the project deliverables. A plan-based approach really requires a manager to have a stable view of everything that has to be developed and the development processes. However, it does not work well with agile methods where the requirements are developed incrementally; where the software is delivered in short, rapid increments; and where changes to the requirements and the software are the norm.

Like every other professional software development process, agile development has to be managed so that the best use is made of the time and resources available to the team. This requires a different approach to project management, which is adapted to incremental development and the particular strengths of agile methods.

The Scrum approach (Schwaber, 2004; Schwaber and Beedle, 2001) is a general agile method but its focus is on managing iterative development rather than specific technical approaches to agile software engineering. Figure 3.8 is a diagram of the Scrum management process. Scrum does not prescribe the use of programming practices such as pair programming and test-first development. It can therefore be used with more technical agile approaches, such as XP, to provide a management framework for the project.

There are three phases in Scrum. The first is an outline planning phase where you establish the general objectives for the project and design the software architecture.

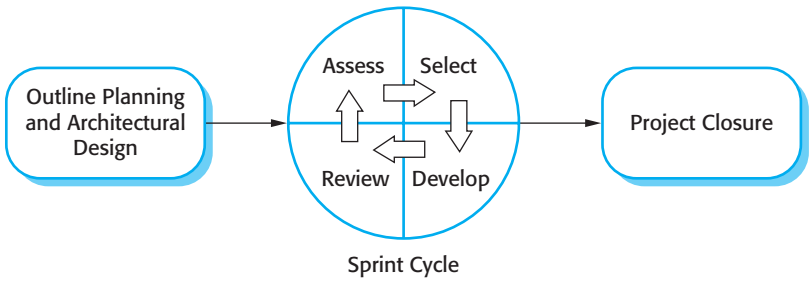


Figure 3.8 The Scrum process

This is followed by a series of sprint cycles, where each cycle develops an increment of the system. Finally, the project closure phase wraps up the project, completes required documentation such as system help frames and user manuals, and assesses the lessons learned from the project.

The innovative feature of Scrum is its central phase, namely the sprint cycles. A Scrum sprint is a planning unit in which the work to be done is assessed, features are selected for development, and the software is implemented. At the end of a sprint, the completed functionality is delivered to stakeholders. Key characteristics of this process are as follows:

1. Sprints are fixed length, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
2. The starting point for planning is the product backlog, which is the list of work to be done on the project. During the assessment phase of the sprint, this is reviewed, and priorities and risks are assigned. The customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each sprint.
3. The selection phase involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.
4. Once these are agreed, the team organizes themselves to develop the software. Short daily meetings involving all team members are held to review progress and if necessary, reprioritize work. During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called ‘Scrum master’. The role of the Scrum master is to protect the development team from external distractions. The way in which the work is done depends on the problem and the team. Unlike XP, Scrum does not make specific suggestions on how to write requirements, test-first development, etc. However, these XP practices can be used if the team thinks they are appropriate.
5. At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

The idea behind Scrum is that the whole team should be empowered to make decisions so the term ‘project manager’, has been deliberately avoided. Rather, the

'Scrum master' is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog, and communicates with customers and management outside of the team.

The whole team attends the daily meetings, which are sometimes 'stand-up' meetings to keep them short and focused. During the meeting, all team members share information, describe their progress since the last meeting, problems that have arisen, and what is planned for the following day. This means that everyone on the team knows what is going on and, if problems arise, can replan short-term work to cope with them. Everyone participates in this short-term planning—there is no top-down direction from the Scrum master.

There are many anecdotal reports of the successful use of Scrum available on the Web. Rising and Janoff (2000) discuss its successful use in a telecommunication software development environment, and they list its advantages as follows:

1. The product is broken down into a set of manageable and understandable chunks.
2. Unstable requirements do not hold up progress.
3. The whole team has visibility of everything and consequently team communication is improved.
4. Customers see on-time delivery of increments and gain feedback on how the product works.
5. Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Scrum, as originally designed, was intended for use with co-located teams where all team members could get together every day in stand-up meetings. However, much software development now involves distributed teams with team members located in different places around the world. Consequently, there are various experiments going on to develop Scrum for distributed development environments (Smits and Pshigoda, 2007; Sutherland et al., 2007).

3.5 Scaling agile methods

Agile methods were developed for use by small programming teams who could work together in the same room and communicate informally. Agile methods have therefore been mostly used for the development of small and medium-sized systems. Of course, the need for faster delivery of software, which is more suited to customer needs, also applies to larger systems. Consequently, there has been a great deal of interest in scaling agile methods to cope with larger systems, developed by large organizations.

Denning et al. (2008) argue that the only way to avoid common software engineering problems, such as systems that don't meet customer needs and budget overruns, is to find ways of making agile methods work for large systems. Leffingwell (2007) discusses which agile practices scale to large systems development. Moore and Spens (2008) report on their experience of using an agile approach to develop a large medical system with 300 developers working in geographically distributed teams.

Large software system development is different from small system development in a number of ways:

1. Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.
2. Large systems are 'brownfield systems' (Hopkins and Jenkins, 2008); that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development. Political issues can also be significant here—often the easiest solution to a problem is to change an existing system. However, this requires negotiation with the managers of that system to convince them that the changes can be implemented without risk to the system's operation.
3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.
4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, etc.
5. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
6. Large systems usually have a diverse set of stakeholders. For example, nurses and administrators may be the end-users of a medical system but senior medical staff, hospital managers, etc. are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

There are two perspectives on the scaling of agile methods:

1. A 'scaling up' perspective, which is concerned with using these methods for developing large software systems that cannot be developed by a small team.

2. A ‘scaling out’ perspective, which is concerned with how agile methods can be introduced across a large organization with many years of software development experience.

Agile methods have to be adapted to cope with large systems engineering. Leffingwell (2007) argues that it is essential to maintain the fundamentals of agile methods—flexible planning, frequent system releases, continuous integration, test-driven development, and good team communications. I believe that the critical adaptations that have to be introduced are as follows:

1. For large systems development, it is not possible to focus only on the code of the system. You need to do more up-front design and system documentation. The software architecture has to be designed and there has to be documentation produced to describe critical aspects of the system, such as database schemas, the work breakdown across teams, etc.
2. Cross-team communication mechanisms have to be designed and used. This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress. A range of communication channels such as e-mail, instant messaging, wikis, and social networking systems should be provided to facilitate communications.
3. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible when several separate programs have to be integrated to create the system. However, it is essential to maintain frequent system builds and regular releases of the system. This may mean that new configuration management tools that support multi-team software development have to be introduced.

Small software companies that develop software products have been amongst the most enthusiastic adopters of agile methods. These companies are not constrained by organizational bureaucracies or process standards and they can change quickly to adopt new ideas. Of course, larger companies have also experimented with agile methods in specific projects, but it is much more difficult for them to ‘scale out’ these methods across the organization. Lindvall, et al. (2004) discuss some of the problems in scaling-out agile methods in four large technology companies.

It is difficult to introduce agile methods into large companies for a number of reasons:

1. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
2. Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Sometimes, these are supported by software

tools (e.g., requirements management tools) and the use of these tools is mandated for all projects.

3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and people with lower skill levels may not be effective team members in agile processes.

4. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Change management and testing procedures are examples of company procedures that may not be compatible with agile methods. Change management is the process of controlling changes to a system, so that the impact of changes is predictable and costs are controlled. All changes have to be approved in advance before they are made and this conflicts with the notion of refactoring. In XP, any developer can improve any code without getting external approval. For large systems, there are also testing standards where a system build is handed over to an external testing team. This may conflict with the test-first and test-often approaches used in XP.

Introducing and sustaining the use of agile methods across a large organization is a process of cultural change. Cultural change takes a long time to implement and often requires a change of management before it can be accomplished. Companies wishing to use agile methods need evangelists to promote change. They must devote significant resources to the change process. At the time of writing, few large companies have made a successful transition to agile development across the organization.

KEY POINTS

- Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads, and producing high-quality code. They involve the customer directly in the development process.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team, and the culture of the company developing the system.
- Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement, and customer participation in the development team.
- A particular strength of extreme programming is the development of automated tests before a program feature is created. All tests must successfully execute when an increment is integrated into a system.

- The Scrum method is an agile method that provides a project management framework. It is centered around a set of sprints, which are fixed time periods when a system increment is developed. Planning is based on prioritizing a backlog of work and selecting the highest-priority tasks for a sprint.
- Scaling agile methods for large systems is difficult. Large systems need up-front design and some documentation. Continuous integration is practically impossible when there are several separate development teams working on a project.

FURTHER READING

Extreme Programming Explained. This was the first book on XP and is still, perhaps, the most readable. It explains the approach from the perspective of one of its inventors and his enthusiasm comes through very clearly in the book. (Kent Beck, Addison-Wesley, 2000.)

'Get Ready for Agile Methods, With Care'. A thoughtful critique of agile methods that discusses their strengths and weaknesses, written by a vastly experienced software engineer. (B. Boehm, *IEEE Computer*, January 2002.) http://doi.ieeecomputersociety.org/10.1109/2.976920.

Scaling Software Agility: Best Practices for Large Enterprises. Although focused on issues of scaling agile development, this book also includes a summary of the principal agile methods such as XP, Scrum, and Crystal. (D. Leffingwell, Addison-Wesley, 2007.)

Running an Agile Software Development Project. Most books on agile methods focus on a specific method but this book takes a different approach and discusses how to put XP into practice in a project. Good, practical advice. (M. Holcombe, John Wiley and Sons, 2008.)

EXERCISES

3.1. Explain why the rapid delivery and deployment of new systems is often more important to businesses than the detailed functionality of these systems.

3.2. Explain how the principles underlying agile methods lead to the accelerated development and deployment of software.

3.3. When would you recommend *against* the use of an agile method for developing a software system?

3.4. Extreme programming expresses user requirements as stories, with each story written on a card. Discuss the advantages and disadvantages of this approach to requirements description.

3.5. Explain why test-first development helps the programmer to develop a better understanding of the system requirements. What are the potential difficulties with test-first development?

3.6. Suggest four reasons why the productivity rate of programmers working as a pair might be more than half that of two programmers working individually.

3.7. Compare and contrast the Scrum approach to project management with conventional plan-based approaches, as discussed in Chapter 23. The comparisons should be based on the effectiveness of each approach for planning the allocation of people to projects, estimating the cost of projects, maintaining team cohesion, and managing changes in project team membership.

3.8. You are a software manager in a company that develops critical control software for aircraft. You are responsible for the development of a software design support system that supports the translation of software requirements to a formal software specification (discussed in Chapter 13). Comment on the advantages and disadvantages of the following development strategies:

a. Collect the requirements for such a system from software engineers and external stakeholders (such as the regulatory certification authority) and develop the system using a plan-driven approach.

b. Develop a prototype using a scripting language, such as Ruby or Python, evaluate this prototype with software engineers and other stakeholders, then review the system requirements. Redevelop the final system using Java.

c. Develop the system in Java using an agile approach with a user involved in the development team.

3.9. It has been suggested that one of the problems of having a user closely involved with a software development team is that they 'go native'; that is, they adopt the outlook of the development team and lose sight of the needs of their user colleagues. Suggest three ways how you might avoid this problem and discuss the advantages and disadvantages of each approach.

3.10. To reduce costs and the environmental impact of commuting, your company decides to close a number of offices and to provide support for staff to work from home. However, the senior management who introduce the policy are unaware that software is developed using agile methods, which rely on close team working and pair programming. Discuss the difficulties that this new policy might cause and how you might get around these problems.

REFERENCES

Ambler, S. W. and Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process.* New York: John Wiley & Sons.

Arisholm, E., Gallis, H., Dyba, T. and Sjoberg, D. I. K. (2007). 'Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise'. *IEEE Trans. on Software Eng.*, **33** (2), 65–86.

Astels, D. (2003). *Test Driven Development: A Practical Guide.* Upper Saddle River, NJ: Prentice Hall.

Beck, K. (1999). 'Embracing Change with Extreme Programming'. *IEEE Computer,* **32** (10), 70–8.

Beck, K. (2000). *extreme Programming explained.* Reading, Mass.: Addison-Wesley.

Carlson, D. (2005). *Eclipse Distilled.* Boston: Addison-Wesley.

Cockburn, A. (2001). *Agile Software Development.* Reading, Mass.: Addison-Wesley.

Cockburn, A. (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams.* Boston: Addison-Wesley.

Cockburn, A. and Williams, L. (2001). 'The costs and benefits of pair programming'. ***In Extreme programming examined. (ed.).*** Boston: Addison-Wesley.

Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum.* Boston: Addison-Wesley.

Demarco, T. and Boehm, B. (2002). 'The Agile Methods Fray'. *IEEE Computer,* **35** (6), 90–2.

Denning, P. J., Gunderson, C. and Hayes-Roth, R. (2008). 'Evolutionary System Development'. *Comm. ACM,* **51** (12), 29–31.

Drobna, J., Noftz, D. and Raghu, R. (2004). 'Piloting XP on Four Mission-Critical Projects'. *IEEE Software,* **21** (6), 70–5.

Highsmith, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* New York: Dorset House.

Hopkins, R. and Jenkins, K. (2008). *Eating the IT Elephant: Moving from Greenfield Development to Brownfield. Boston,* Mass.: IBM Press.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process.* Englewood Cliff, NJ: Prentice Hall.

Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises.* Boston: Addison-Wesley.

Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. and Kahkonen, T. (2004). 'Agile Software Development in Large Organizations'. *IEEE Computer,* **37** (12), 26–34.

Martin, J. (1981). *Application Development Without Programmers.* Englewood Cliffs, NJ: Prentice-Hall.

Massol, V. and Husted, T. (2003). *JUnit in Action.* Greenwich, Conn.: Manning Publications Co.

Mills, H. D., O'Neill, D., Linger, R. C., Dyer, M. and Quinnan, R. E. (1980). 'The Management of Software Engineering'. *IBM Systems.* J., **19** (4), 414–77.

Moore, E. and Spens, J. (2008). 'Scaling Agile: Finding your Agile Tribe'. *Proc. Agile 2008 Conference,* Toronto: IEEE Computer Society. 121–124.

Palmer, S. R. and Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development.* Englewood Cliffs, NJ: Prentice Hall.

Parrish, A., Smith, R., Hale, D. and Hale, J. (2004). 'A Field Study of Developer Pairs: Productivity Impacts and Implications'. *IEEE Software,* **21** (5), 76–9.

Poole, C. and Huisman, J. W. (2001). 'Using Extreme Programming in a Maintenance Environment'. *IEEE Software,* **18** (6), 42–50.

Rising, L. and Janoff, N. S. (2000). 'The Scrum Software Development Process for Small Teams'. *IEEE Software,* **17** (4), 26–32.

Schwaber, K. (2004). *Agile Project Management with Scrum.* Seattle: Microsoft Press.

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum.* Englewood Cliffs, NJ: Prentice Hall.

Smits, H. and Pshigoda, G. (2007). 'Implementing Scrum in a Distributed Software Development Organization'. Agile 2007, Washington, DC: IEEE Computer Society.

Stapleton, J. (1997). *DSDM Dynamic Systems Development Method.* Harlow, UK: Addison-Wesley.

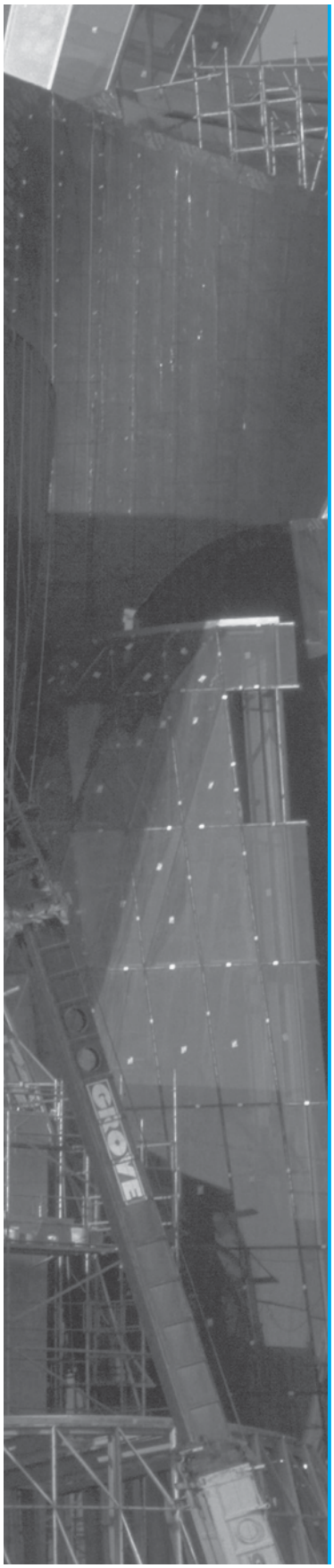
Stapleton, J. (2003). *DSDM: Business Focused Development, 2nd ed.* Harlow, UK: Pearson Education.

Stephens, M. and Rosenberg, D. (2003). *Extreme Programming Refactored.* Berkley, Calif.: Apress.

Sutherland, J., Viktorov, A., Blount, J. and Puntikov, N. (2007). 'Distributed Scrum: Agile Project Management with Outsourced Development Teams'. 40th Hawaii Int. Conf. on System Sciences, Hawaii: IEEE Computer Society.

Weinberg, G. (1971). *The Psychology of Computer Programming.* New York: Van Nostrand.

Williams, L., Kessler, R. R., Cunningham, W. and Jeffries, R. (2000). 'Strengthening the Case for Pair Programming'. *IEEE Software,* **17** (4), 19–25.



4

Requirements engineering

Objectives

The objective of this chapter is to introduce software requirements and to discuss the processes involved in discovering and documenting these requirements. When you have read the chapter you will:

- understand the concepts of user and system requirements and why these requirements should be written in different ways;
- understand the differences between functional and nonfunctional software requirements;
- understand how requirements may be organized in a software requirements document;
- understand the principal requirements engineering activities of elicitation, analysis and validation, and the relationships between these activities;
- understand why requirements management is necessary and how it supports other requirements engineering activities.

Contents

The requirements for a system are the descriptions of what the system should do—the services that it provides and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

The term 'requirement' is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function. Davis (1993) explains why these differences exist:

> *If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.*

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term 'user requirements' to mean the high-level abstract requirements and 'system requirements' to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.
2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Different levels of requirements are useful because they communicate information about the system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from a mental health care patient management system (MHC-PMS) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.

1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.

1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.

1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.

1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Figure 4.1 User and system requirements

You need to write requirements at different levels of detail because different readers use them in different ways. Figure 4.2 shows possible readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

In this chapter, I present a 'traditional' view of requirements rather than requirements in agile processes. For most large systems, it is still the case that there is a clearly identifiable requirements engineering phase before the implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, there are usually subsequent changes to the requirements and user requirements may be expanded into more detailed system requirements. However, the agile approach of concurrently eliciting the requirements as the system is developed is rarely used for large systems development.

4.1 Functional and non-functional requirements

Software system requirements are often classified as functional requirements or non-functional requirements:

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system

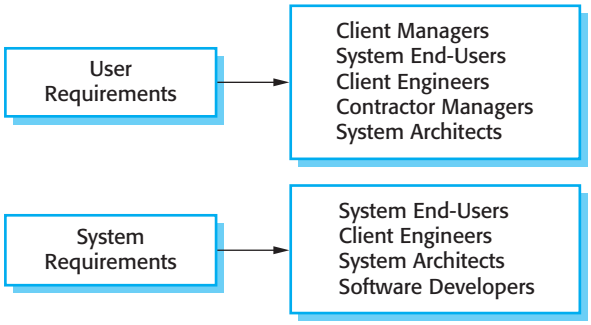


Figure 4.2 Readers of different types of requirements specification

should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

In reality, the distinction between different types of requirement is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a non-functional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered properly.

4.1.1 Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.

Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems. For example, here are examples of functional

Domain requirements

Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They may be new functional requirements in their own right, constrain existing functional requirements, or set out how particular computations must be carried out.

The problem with domain requirements is that software engineers may not understand the characteristics of the domain in which the system operates. They often cannot tell whether or not a domain requirement has been missed out or conflicts with other requirements.

http://www.SoftwareEngineering-9.com/Web/Requirements/DomainReq.html

requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document and they show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

Imprecision in the requirements specification is the cause of many software engineering problems. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

For example, the first example requirement for the MHC-PMS states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, irrespective of the clinic.

The medical staff member specifying this may expect 'search' to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement in a different way and may implement a search so that the user has to choose a clinic then carry out the search. This obviously will involve more user input and so take longer.

In principle, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory

definitions. In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness. One reason for this is that it is easy to make mistakes and omissions when writing specifications for complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification. The problems may only emerge after deeper analysis or after the system has been delivered to the customer.

4.1.2 Non-functional requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements.

Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the need for interoperability with other software or

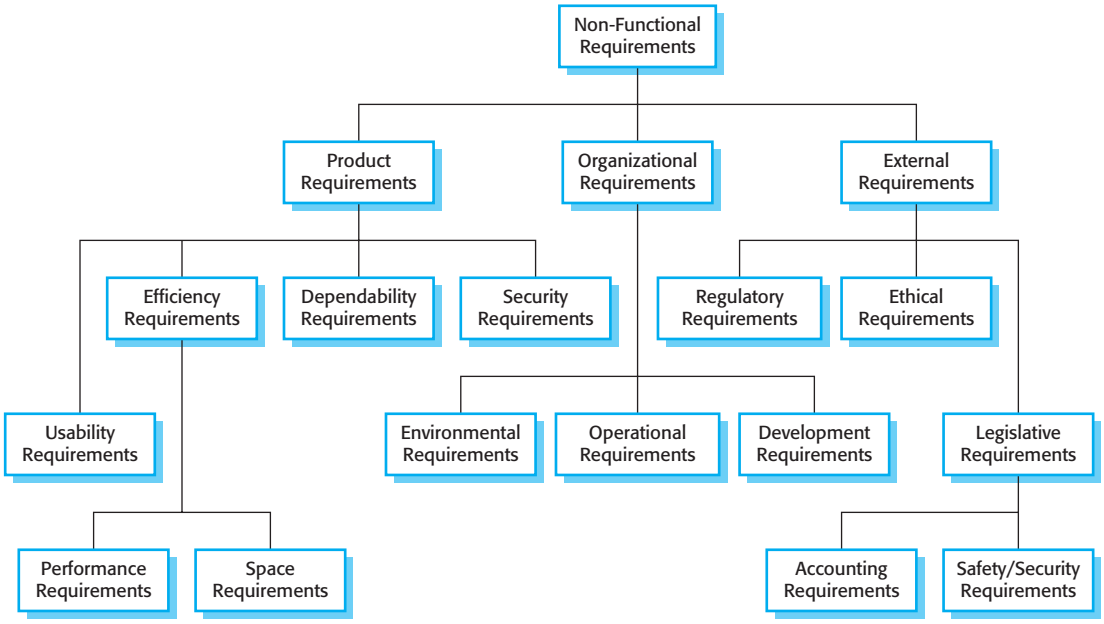


Figure 4.3 Types of non-functional requirement

hardware systems, or external factors such as safety regulations or privacy legislation. Figure 4.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or from external sources:

1. *Product requirements* These requirements specify or constrain the behavior of the software. Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.
2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization. Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.
3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a central bank; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

PRODUCT REQUIREMENT
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Figure 4.4 Examples of non-functional requirements in the MHC-PMS

Figure 4.4 shows examples of product, organizational, and external requirements taken from the MHC-PMS whose user requirements were introduced in Section 4.1.1. The product requirement is an availability requirement that defines when the system has to be available and the allowed down time each day. It says nothing about the functionality of MHC-PMS and clearly identifies a constraint that has to be considered by the system designers.

The organizational requirement specifies how users authenticate themselves to the system. The health authority that operates the system is moving to a standard authentication procedure for all software where, instead of users having a login name, they swipe their identity card through a reader to identify themselves. The external requirement is derived from the need for the system to conform to privacy legislation. Privacy is obviously a very important issue in healthcare systems and the requirement specifies that the system should be developed in accordance with a national privacy standard.

A common problem with non-functional requirements is that users or customers often propose these requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. For example, the following system goal is typical of how a manager might express usability requirements:

> *The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.*

I have rewritten this to show how the goal could be expressed as a 'testable' non-functional requirement. It is impossible to objectively verify the system goal, but in the description below you can at least include software instrumentation to count the errors made by users when they are testing the system.

> *Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.*

Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Figure 4.5 shows metrics that you can use to specify non-functional system properties. You can measure these characteristics

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 4.5 Metrics for specifying non-functional requirements

when the system is being tested to check whether or not the system has met its non-functional requirements.

In practice, customers for a system often find it difficult to translate their goals into measurable requirements. For some goals, such as maintainability, there are no metrics that can be used. In other cases, even when quantitative specification is possible, customers may not be able to relate their needs to these specifications. They don't understand what some number defining the required reliability (say) means in terms of their everyday experience with computer systems. Furthermore, the cost of objectively verifying measurable, non-functional requirements can be very high and the customers paying for the system may not think these costs are justified.

Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, the authentication requirement in Figure 4.4 obviously requires a card reader to be installed with each computer attached to the system. However, there may be another requirement that requests mobile access to the system from doctors' or nurses' laptops. These are not normally equipped with card readers so, in these circumstances, some alternative authentication method may have to be allowed.

It is difficult, in practice, to separate functional and non-functional requirements in the requirements document. If the non-functional requirements are stated separately from the functional requirements, the relationships between them may be hard to understand. However, you should explicitly highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

Requirements document standards

A number of large organizations, such as the U.S. Department of Defense and the IEEE, have defined standards for requirements documents. These are usually very generic but are nevertheless useful as a basis for developing more detailed organizational standards. The U.S. Institute of Electrical and Electronic Engineers (IEEE) is one of the best-known standards providers and they have developed a standard for the structure of requirements documents. This standard is most appropriate for systems such as military command and control systems that have a long lifetime and are usually developed by a group of organizations.

http://www.SoftwareEngineering-9.com/Web/Requirements/IEEE-standard.html

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems. I cover these requirements in Chapter 12, where I describe specific techniques for specifying dependability and security requirements.

4.2 The software requirements document

The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. Sometimes, the user and system requirements are integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented in a separate document.

Requirements documents are essential when an outside contractor is developing the software system. However, agile development methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort is largely wasted. Rather than a formal document, approaches such as Extreme Programming (Beck, 1999) collect user requirements incrementally and write these on cards as user stories. The user then prioritizes requirements for implementation in the next increment of the system.

For business systems where requirements are unstable, I think that this approach is a good one. However, I think that it is still useful to write a short supporting document that defines the business and dependability requirements for the system; it is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Figure 4.6, taken from my book with Gerald Kotonya on requirements engineering (Kotonya and Sommerville, 1998) shows possible users of the document and how they use it.

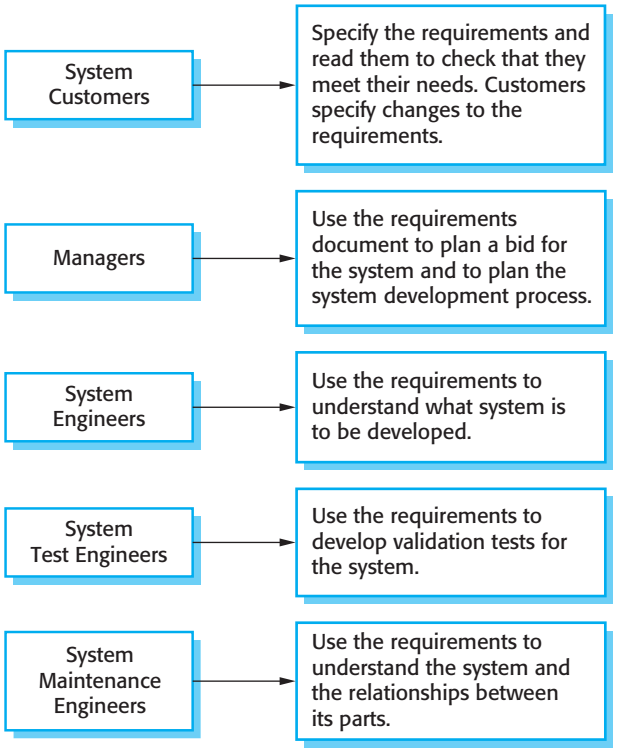


Figure 4.6 Users of a requirements document

The diversity of possible users means that the requirements document has to be a compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers, and including information about possible system evolution. Information on anticipated changes can help system designers avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements.

The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. Critical systems need to have detailed requirements because safety and security have to be analyzed in detail. When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise. If an in-house, iterative development process is used, the requirements document can be much less detailed and any ambiguities can be resolved during development of the system.

Figure 4.7 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE, 1998). This standard is a generic standard that can be adapted to specific uses. In this case, I have extended the standard to include information about predicted system evolution. This information helps the maintainers of the system and allows designers to include support for future system features.

Naturally, the information that is included in a requirements document depends on the type of software being developed and the approach to development that is to be used. If an evolutionary approach is adopted for a software product (say), the

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Figure 4.7 The structure of a requirements document

requirements document will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, non-functional system requirements. In this case, the designers and programmers use their judgment to decide how to meet the outline user requirements for the system.

However, when the software is part of a large system project that includes interacting hardware and software systems, it is usually necessary to define the requirements

Problems with using natural language for requirements specification

The flexibility of natural language, which is so useful for specification, often causes problems. There is scope for writing unclear requirements, and readers (the designers) may misinterpret requirements because they have a different background to the user. It is easy to amalgamate several requirements into a single sentence and structuring natural language requirements can be difficult.

http://www.SoftwareEngineering-9.com/Web/Requirements/NL-problems.html

to a fine level of detail. This means that the requirements documents are likely to be very long and should include most if not all of the chapters shown in Figure 4.7. For long documents, it is particularly important to include a comprehensive table of contents and document index so that readers can find the information that they need.

4.3 Requirements specification

Requirements specification is the process of writing down the user and system requirements in a requirements document. Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent. In practice, this is difficult to achieve as stakeholders interpret the requirements in different ways and there are often inherent conflicts and inconsistencies in the requirements.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is practically impossible to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract.

Figure 4.8 Ways of writing a system requirements specification

the different sub-systems that make up the system. As I discuss in Chapters 6 and 18, this architectural definition is essential if you want to reuse software components when implementing the system.

2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements (such as N-version programming to achieve reliability, discussed in Chapter 13) may be necessary. An external regulator who needs to certify that the system is safe may specify that an already certified architectural design be used.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language but other notations based on forms, graphical system models, or mathematical system models can also be used. Figure 4.8 summarizes the possible notations that could be used for writing system requirements.

Graphical models are most useful when you need to show how a state changes or when you need to describe a sequence of actions. UML sequence charts and state charts, described in Chapter 5, show the sequence of actions that occur in response to a certain message or event. Formal mathematical specifications are sometimes used to describe the requirements for safety- or security-critical systems, but are rarely used in other circumstances. I explain this approach to writing specifications in Chapter 12.