Software Engineering Fundamentals

Myanmar IT Consulting
Zero to Pro CS Bootcamp Level-3

Introduction

1.1 Professional software development

1.2 Software engineering ethics

1.3 Case studies

Objectives

- to introduce software engineering and to provide a framework for understanding the rest of the book.
- understand what software engineering is and why it is important
- understand that the development of different types of software systems may require different software engineering techniques
- understand some ethical and professional issues that are important for software engineers

1.1 Professional software development

We can’t run the modern world without software.

Software systems are abstract and intangible.

There are many different types of software systems, from simple embedded systems to complex, worldwide information systems.

software failures

1. *Increasing demands*
2. *Low expectations*

Professional software development

Professional software, intended for use by someone apart from its developer, is usually developed by teams rather than individuals. It is maintained and changed throughout its life.

Software engineering is intended to support professional software development, rather than individual programming.

A professionally developed software system is often more than a single program.

if you are writing software that other people will use and other engineers will change then you usually have to provide additional information as well as the code of the program.

Two kinds of software products:

Generic products These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them.

Customized (or bespoke) products These are systems that are commissioned by a particular customer.

More and more systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer.

a large and complex system is adapted for a company by incorporating information about business rules and processes, reports required, and so on.

1.1.1 Software engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

Engineering discipline Engineers make things work. They apply theories, methods, and tools where these are appropriate.

All aspects of software production Software engineering is not just concerned with the technical processes of software development.

Engineering is about getting results of the required quality within the schedule and budget.

Important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project.

The systematic approach that is used in software engineering is sometimes called a software process.

1. Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. Software development, where the software is designed and programmed.
3. Software validation, where the software is checked to ensure that it is what the customer requires.
4. Software evolution, where the software is modified to reflect changing customer and market requirements.

Three general issues that affect many different types of software:

1. *Heterogeneity* Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.
2. *Business and social change* Business and society are changing incredibly quickly as emerging economies develop and new technologies become available.
3. *Security and trust* As software is intertwined with all aspects of our lives, it is essential that we can trust that software.

1.1.2 Software engineering diversity

1. *Stand-alone applications*
2. *Interactive transaction-based applications*
3. *Embedded control systems*
4. *Batch processing systems*
5. *Entertainment systems*
6. *Systems for modeling and simulation*
7. *Data collection systems*
8. *Systems of systems*

1.1.3 Software engineering and the Web

Instead of writing software and deploying it on users' PCs, the software was deployed on a web server.

The next stage in the development of web-based systems was the notion of web services.

Web services are software components that deliver specific, useful functionality and which are accessed over the Web.

In the last few years, the notion of 'software as a service' has been developed.

1.2 Software engineering ethics

1. *Confidentiality* You should normally respect the confidentiality of your employ- ers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
2. *Competence* You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
3. *Intellectual property rights* You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.
4. *Computer misuse* You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses or other malware).

Case Studies

An embedded system This is a system where the software controls a hardware device and is embedded in that device.

An information system This is a system whose primary purpose is to manage and provide access to a database of information.

A sensor-based data collection system This is a system whose primary purpose is to collect data from a set of sensors and process that data in some way.

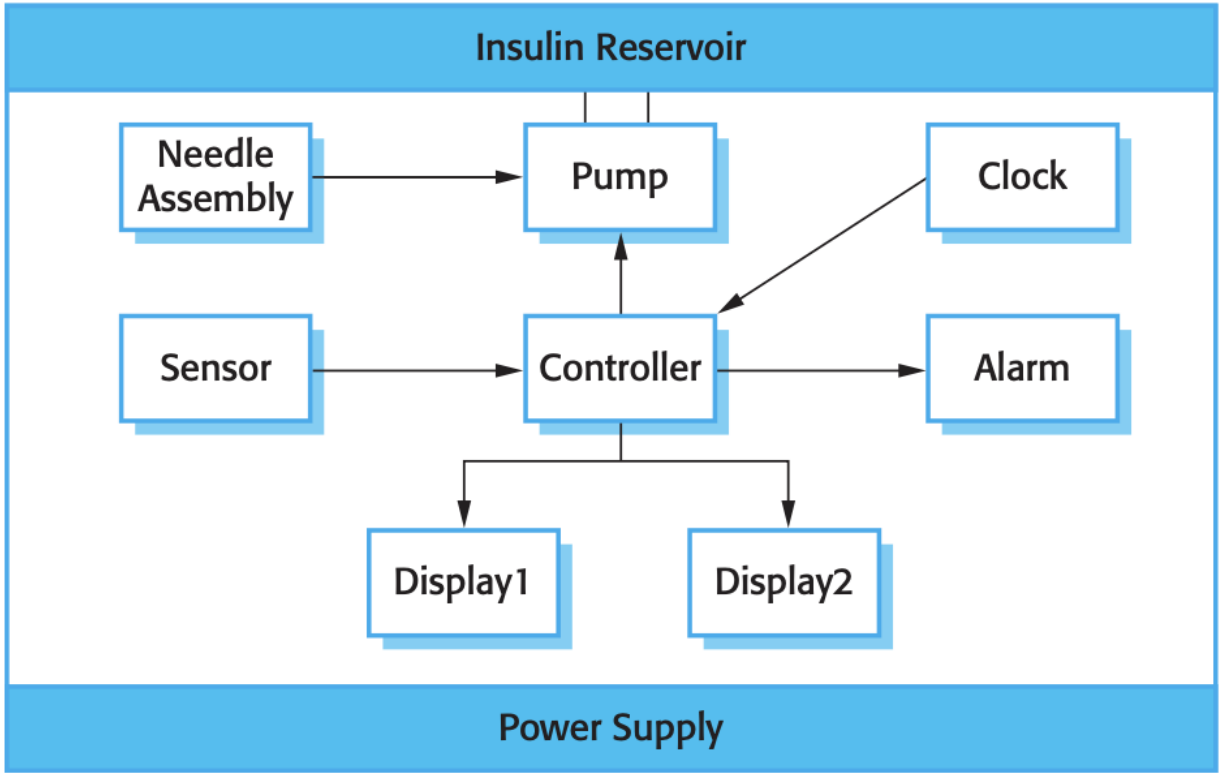

Figure 1.4 Insulin pump hardware

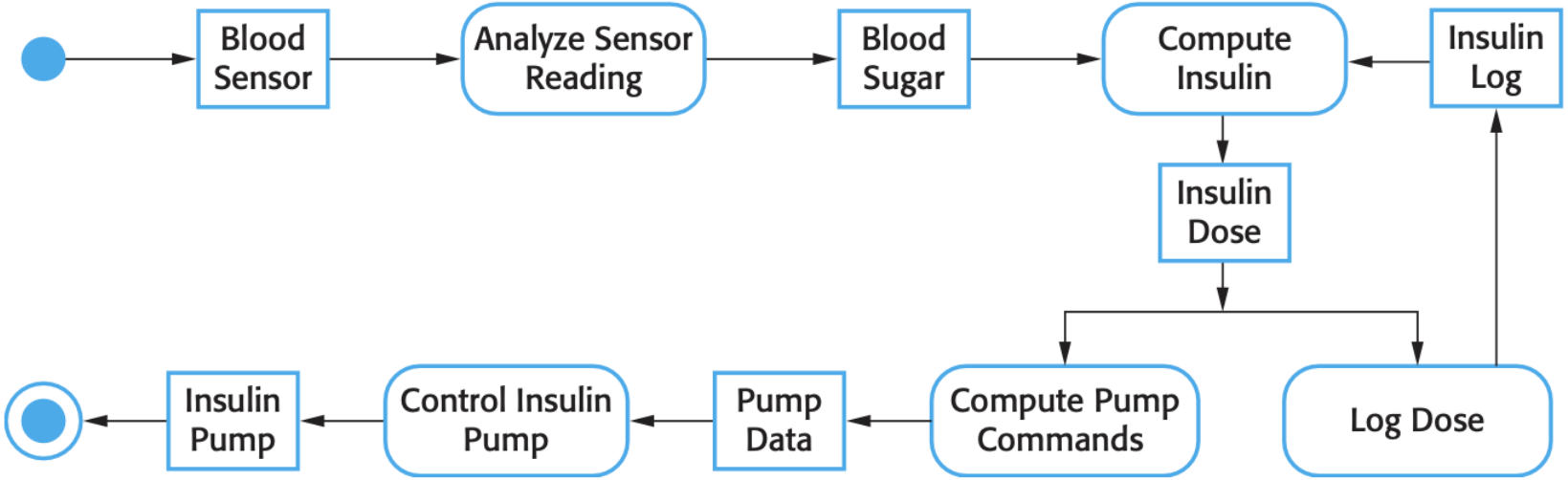

Figure 1.5 Activity model of the insulin pump

Figure 1.4 shows the hardware components and organization of the insulin pump. To understand the examples in this book, all you need to know is that the blood sensor measures the electrical conductivity of the blood under different conditions and that these values can be related to the blood sugar level. The

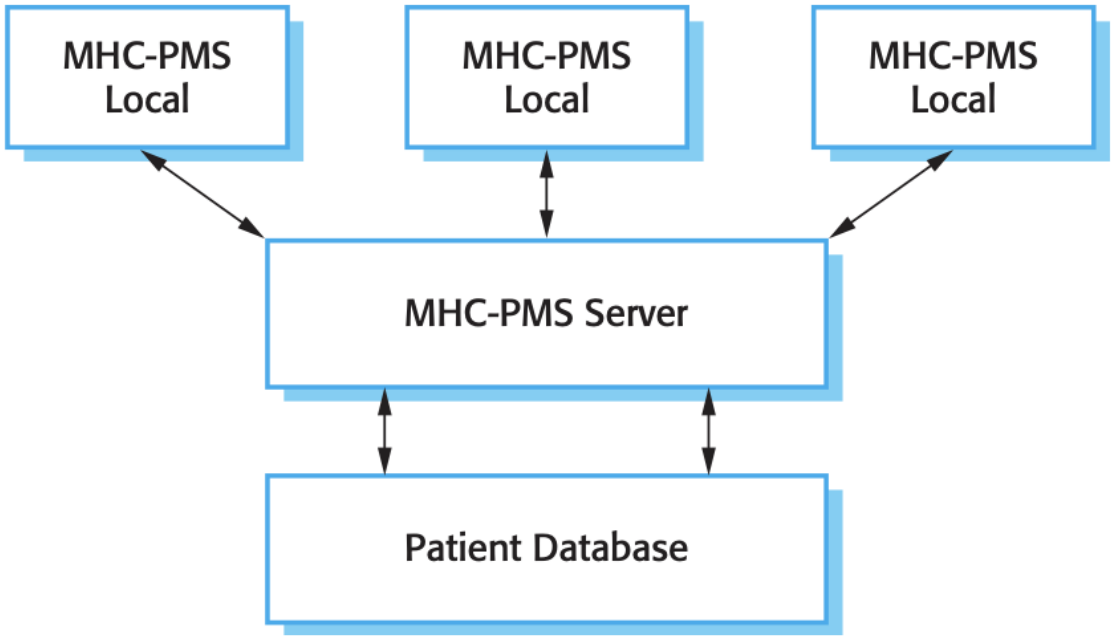

Figure 1.6 The organization of the MHC-PMS

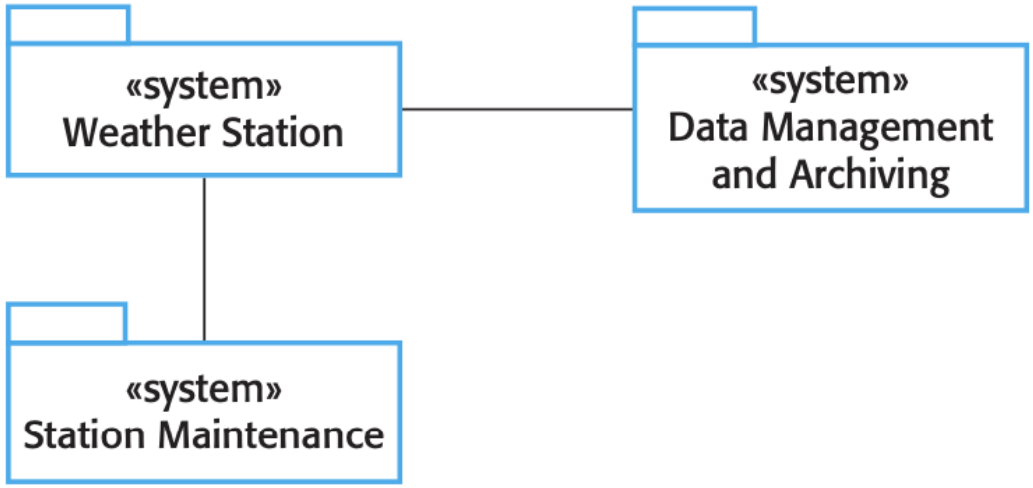

Figure 1.7 The weather station's environment

Chapter(2)

Software Processes

Software processes

to introduce you to the idea of a software process—a coherent set of activities for software production.

understand the concepts of software processes and software process models;

know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;

understand why processes should be organized to cope with changes in the software requirements and design;

understand how the Rational Unified Process integrates good software engineering practice to create adaptable software processes.

Contents

four activities that are fundamental to software engineering:

A software process is a set of related activities that leads to the production of a software product.

1. *Software specification* The functionality of the software and constraints on its operation must be defined.
2. *Software design and implementation* The software to meet the specification must be produced.
3. *Software validation* The software must be validated to ensure that it does what the customer wants.
4. *Software evolution* The software must evolve to meet changing customer needs.

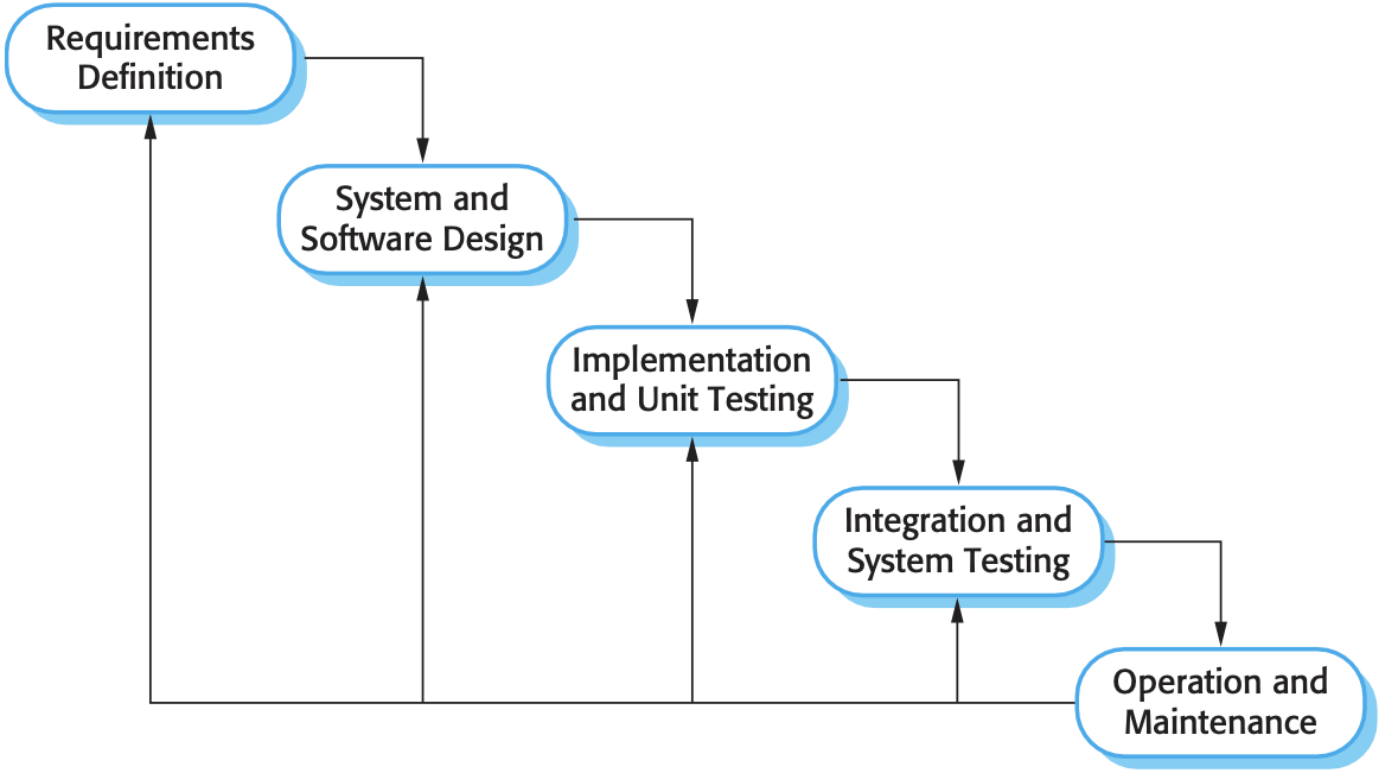

Figure 2.1 The waterfall model

1. *Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. *System and software design* The systems design process allocates the require- ments to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamen- tal software system abstractions and their relationships.
3. *Implementation and unit testing* During this stage, the software design is real- ized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. *Operation and maintenance* Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

Incremental development

1. *Incremental development* This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.

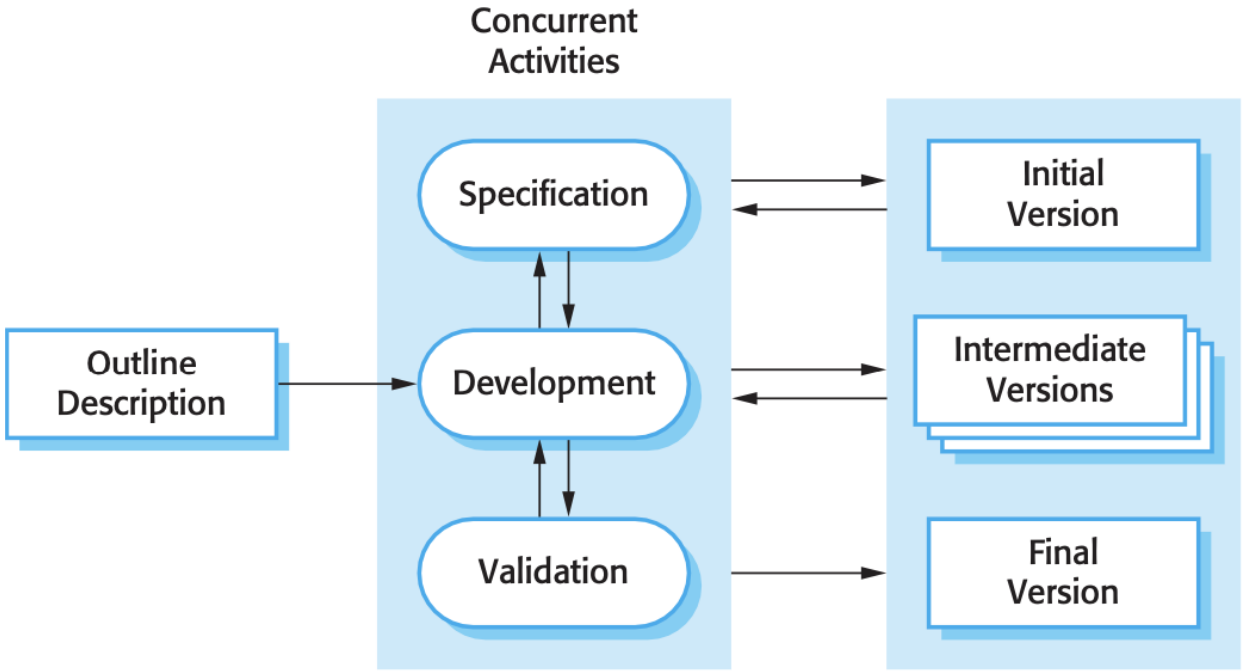

Figure 2.2 Incremental development

Reuse-oriented software engineering

This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

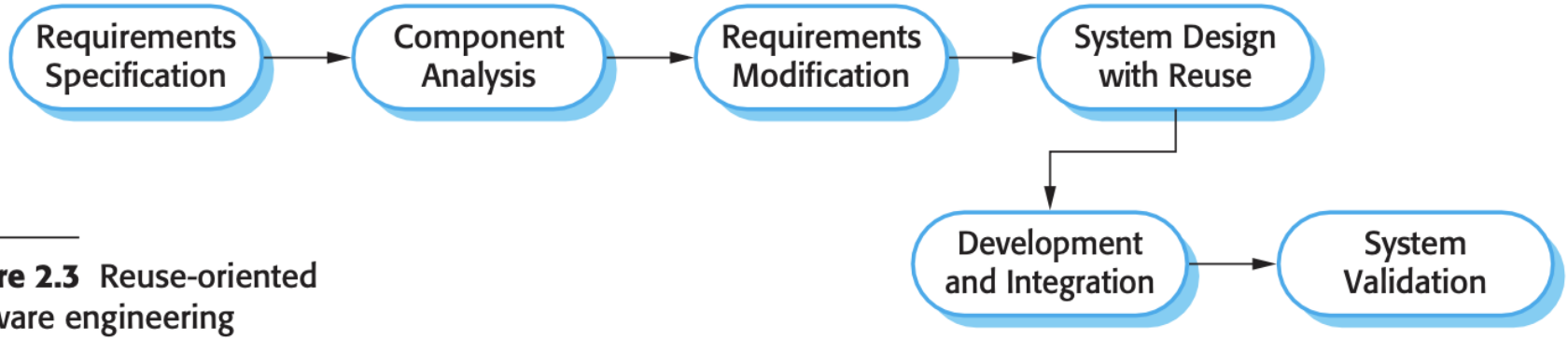

Figure 2.3 Reuse-oriented software engineering

2.2 Process activities

- Specification,
- Development,
- Validation,
- Evolution

2.2.1 Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system’s operation and development.

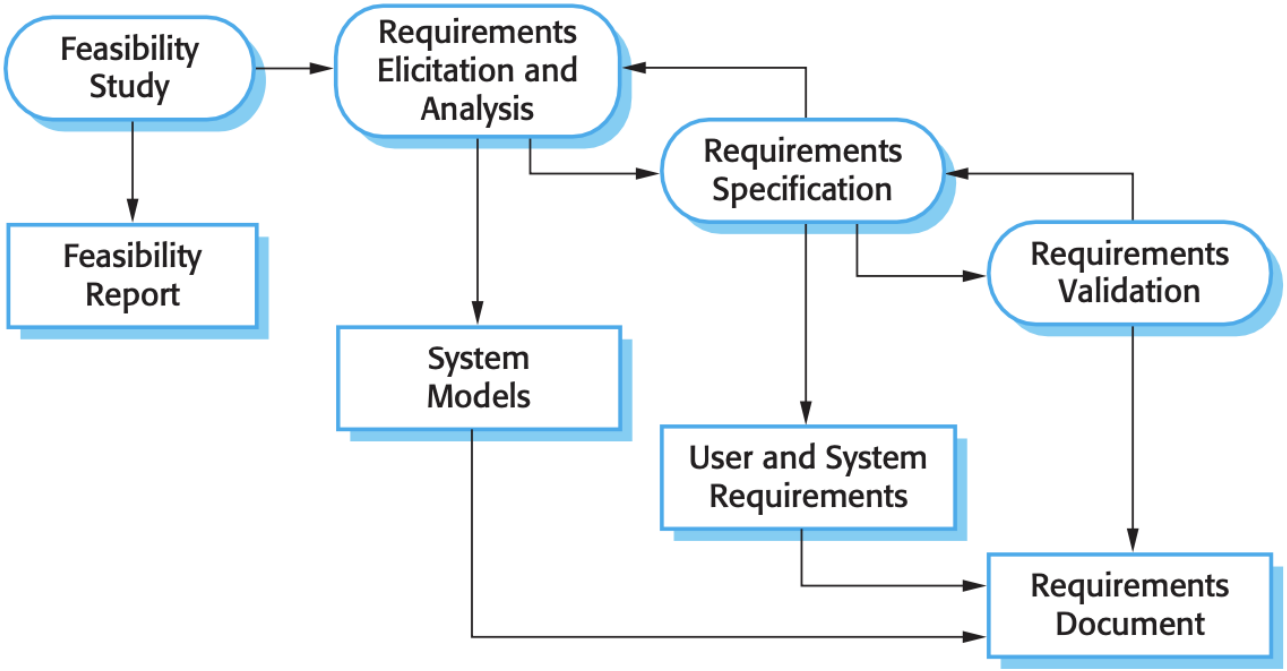

Figure 2.4 The requirements engineering process

Four main activities

1. *Feasibility study* An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies.
2. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on.
3. *Requirements specification* Requirements specification is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements.
4. *Requirements validation* This activity checks the requirements for realism, consistency, and completeness.

2.2.2 Software design and implementation

1. *Architectural design,* where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
2. *Interface design,* where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
3. *Component design,* where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
4. *Database design,* where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

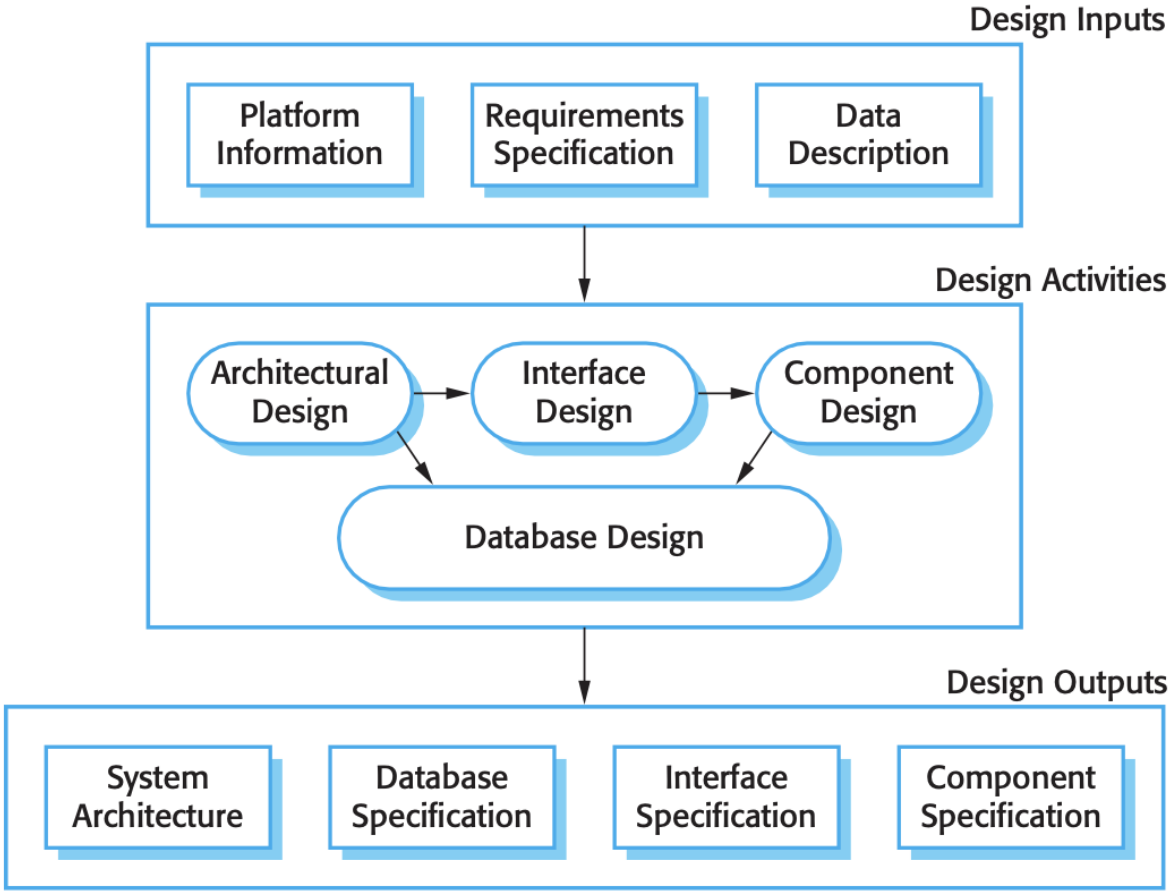

Figure 2.5 A general model of the design process

2.2.3 Software validation

Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer.

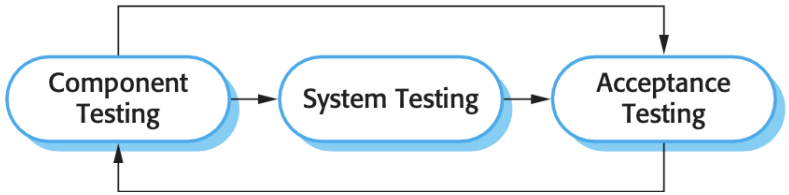

Figure 2.6 Stages of testing

Development testing The components making up the system are tested by the people developing the system.

System testing System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems.

Acceptance testing This is the final stage in the testing process before the system is accepted for operational use.

2.2.4 Software evolution

The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems.

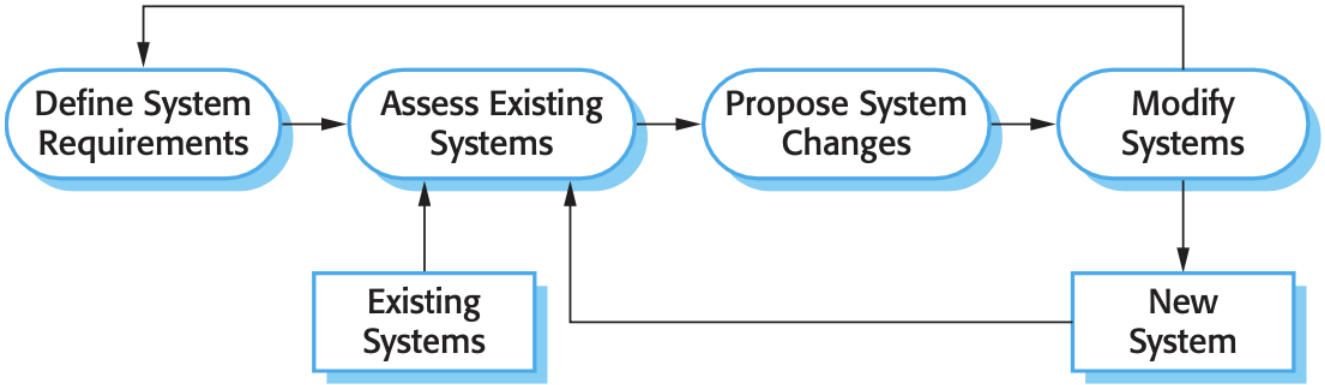

Figure 2.8 System evolution

2.3 Coping with change

Change is inevitable in all large software projects. The system requirements change as the business procuring the system responds to external pressures and management priorities change.

1. Change avoidance, where the software process includes activities that can anticipate possible changes before significant rework is required.
2. Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost.

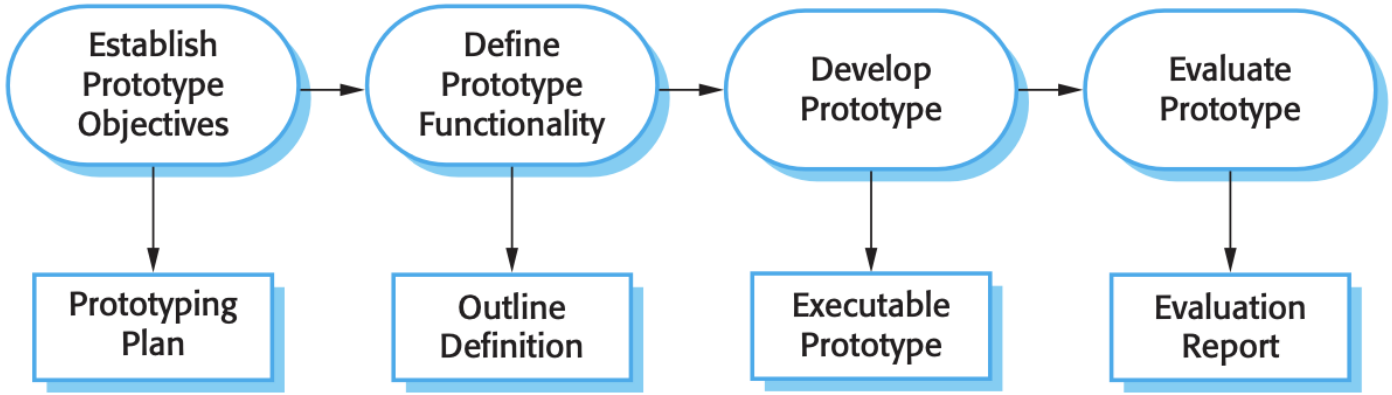

Figure 2.9 The process of prototype development

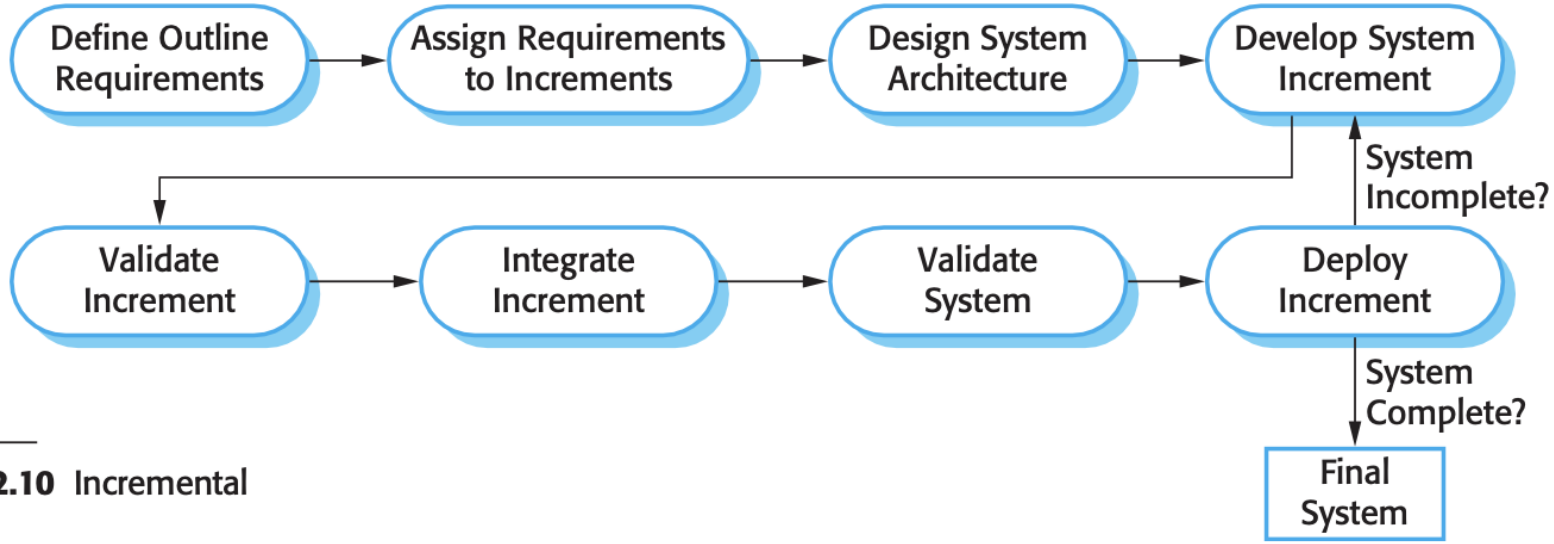

Figure 2.10 Incremental delivery

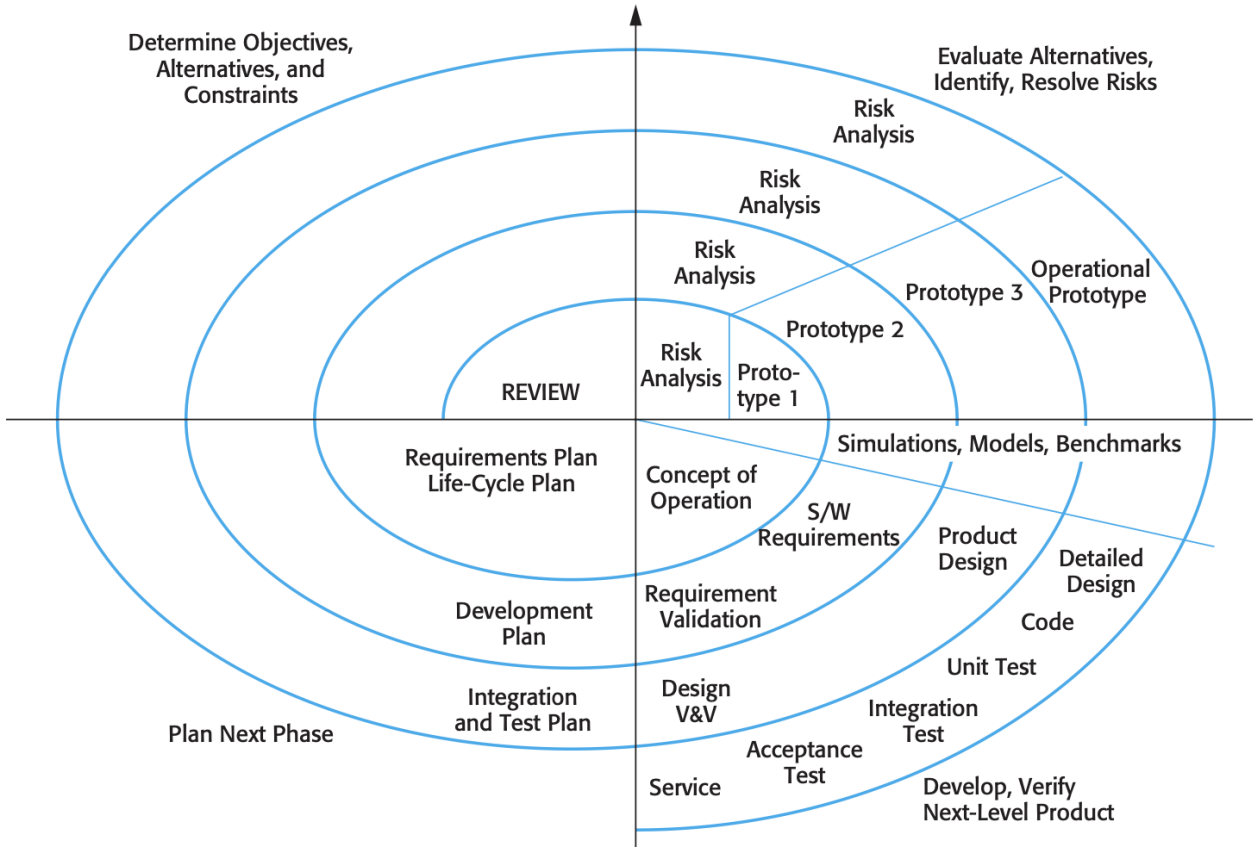

Figure 2.11 Boehm's spiral model of the software process (©IEEE 1988)

changes are a result of project risks and includes explicit risk management activities to reduce these risks.

Each loop in the spiral is split into four sectors:

2.3.1 Prototyping

A prototype is an initial version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions.

2.3.2 Incremental delivery

Incremental delivery is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in an operational environment.

2.3.3 Boehm’s spiral model

A risk-driven software process framework (the spiral model) was proposed by Boehm (1988)
The spiral model combines change avoidance with change tolerance.

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models, and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users, and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Figure 2.13 Static workflows in the Rational Unified Process

The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development. Six fundamental best practices are recommended:

Chapter 3

Agile software development

Objectives

- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan- driven development;
- know the key practices in extreme programming and how these relate to the general principles of agile methods;
- understand the Scrum approach to agile project management;
- be aware of the issues and problems of scaling agile development

methods to the development of large software systems.

Contents

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Figure 3.1 The principles of agile methods

As I discuss in the final section of this chapter, the success of agile methods has meant that there is a lot of interest in using these methods for other types of software development. However, because of their focus on small, tightly integrated teams, there are problems in scaling them to large systems. There have also been experi-

3.1 Agile methods

Agile methods are incremental development methods in which the increments are small and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

Probably the best-known agile method is extreme programming
Other agile approaches include
Scrum , Crystal , Adaptive Software Development , DSDM , and Feature Driven Development
.

3.2 Plan-driven and agile development

Agile approaches to software development consider design and implementation to be the central activities in the software process.

In a plan-driven approach, iteration occurs within activities with formal documents used to communicate between stages of the process.

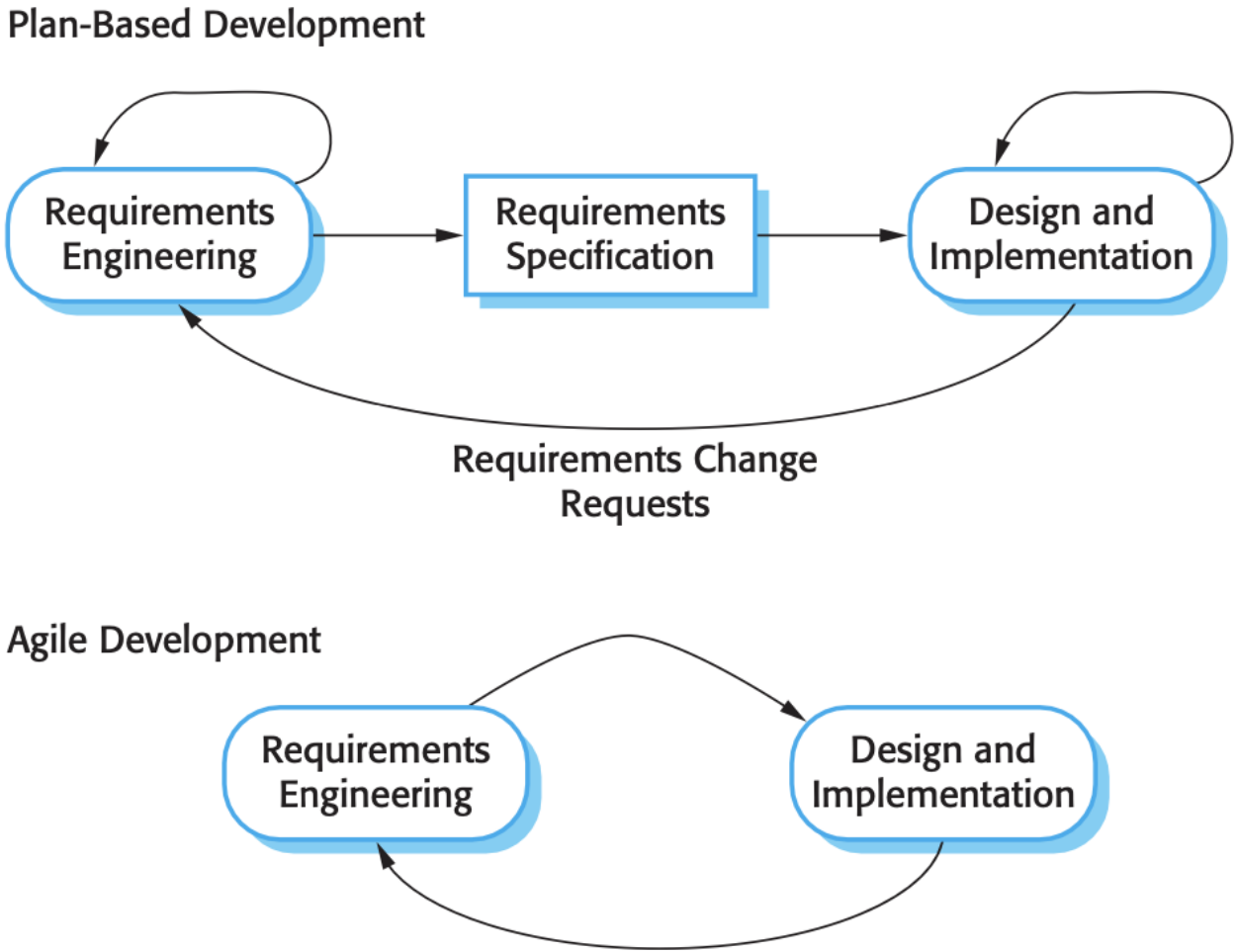

Figure 3.2 Plan-driven and agile specification

3.3 Extreme programming

Extreme programming (XP) is perhaps the best known and most widely used of the agile methods.
several new versions of a system may be developed by different programmers, integrated and tested in a day.

Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system.

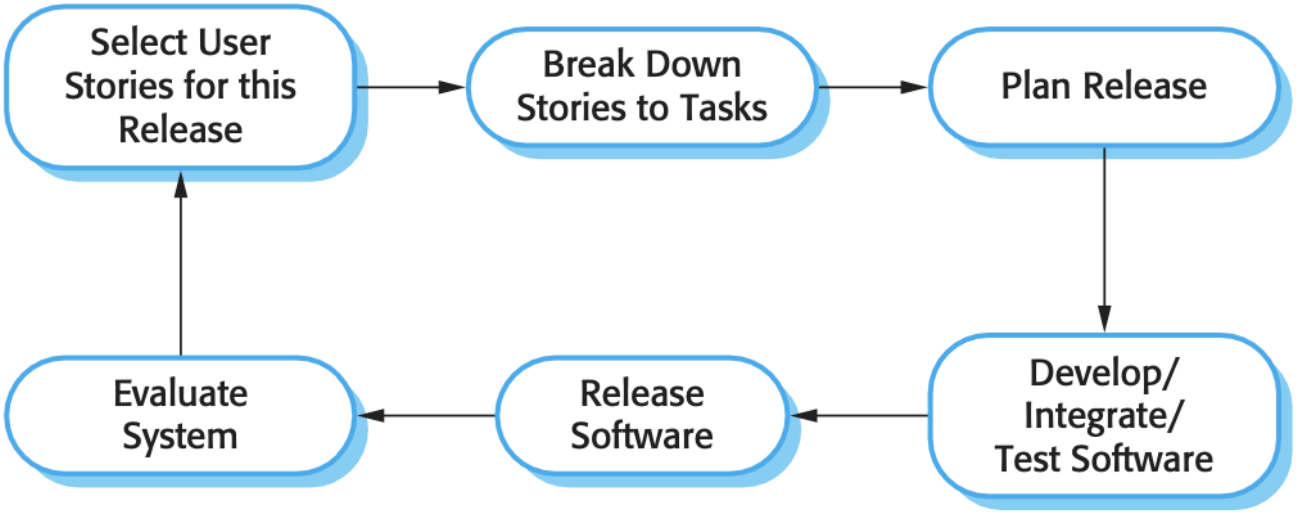

Figure 3.3 The extreme programming release cycle

3.3.1 Testing in XP

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.

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.

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.

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

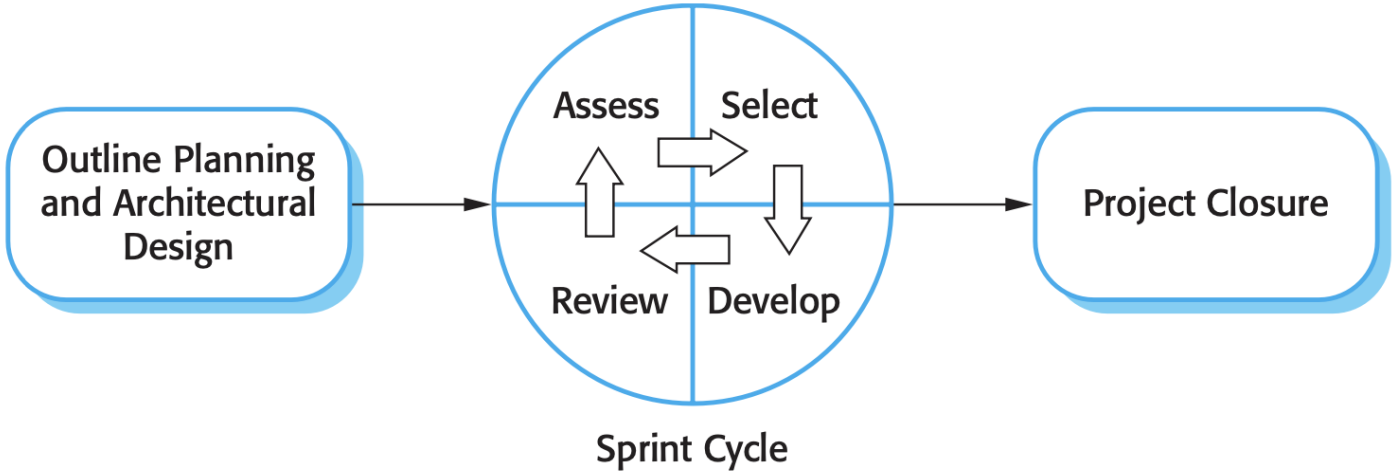

Figure 3.8 The Scrum process

3.4 Agile project management

agile development has to be managed so that the best use is made of the time and resources available to the team.

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.

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.

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.

Chapter 4

Requirements engineering

Objectives

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

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.

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.

4.1 Functional and 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 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.

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

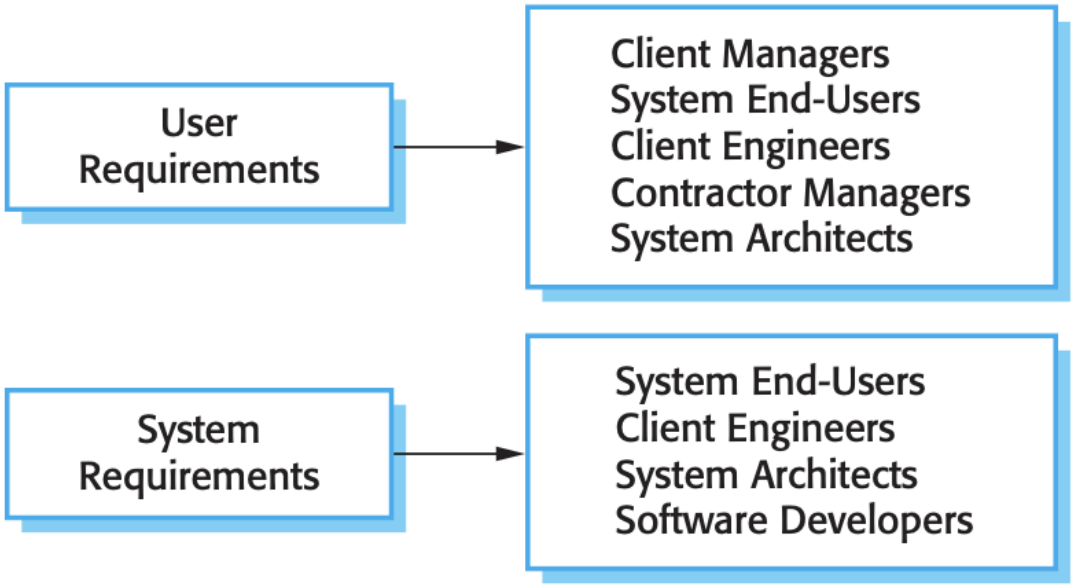

Figure 4.2 Readers of different types of requirements specification

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.

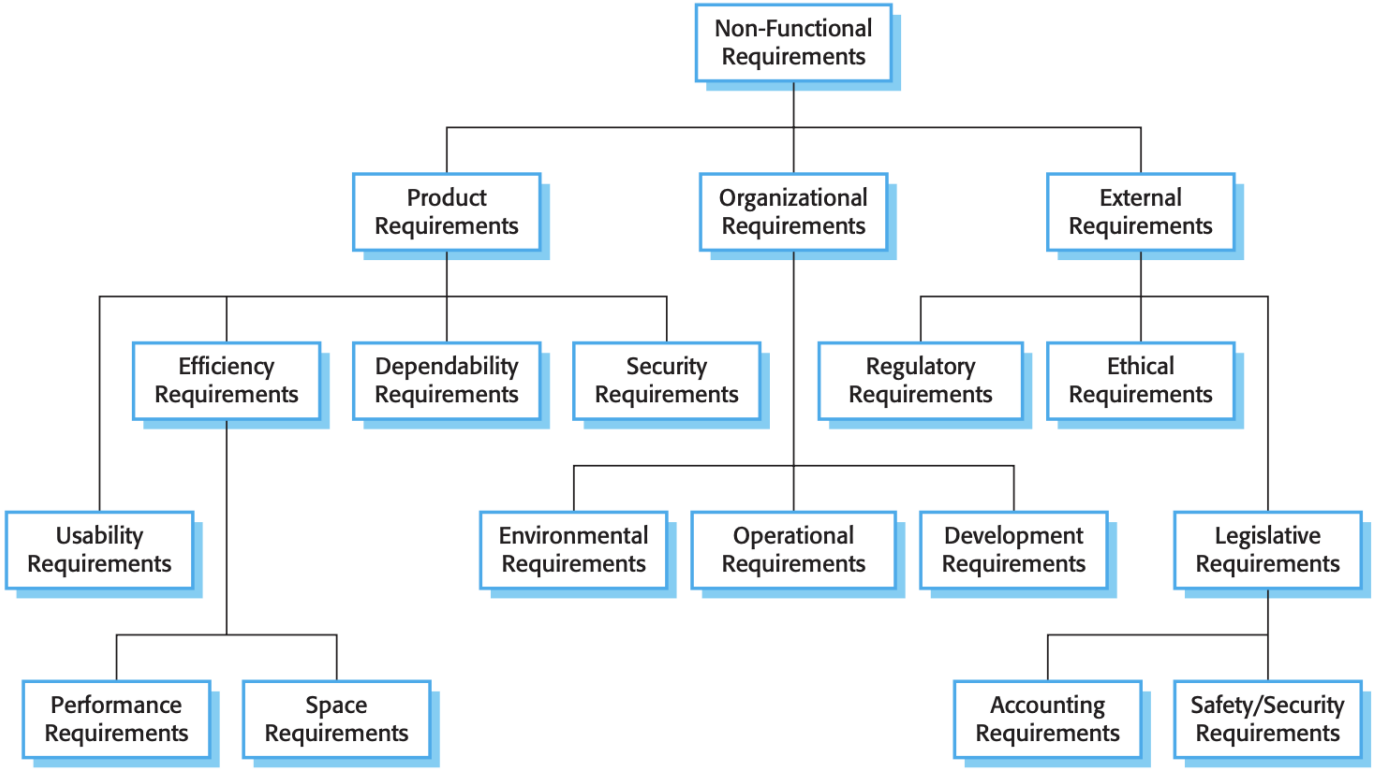

Figure 4.3 Types of non-functional requirement

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 soft-

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

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

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.

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.

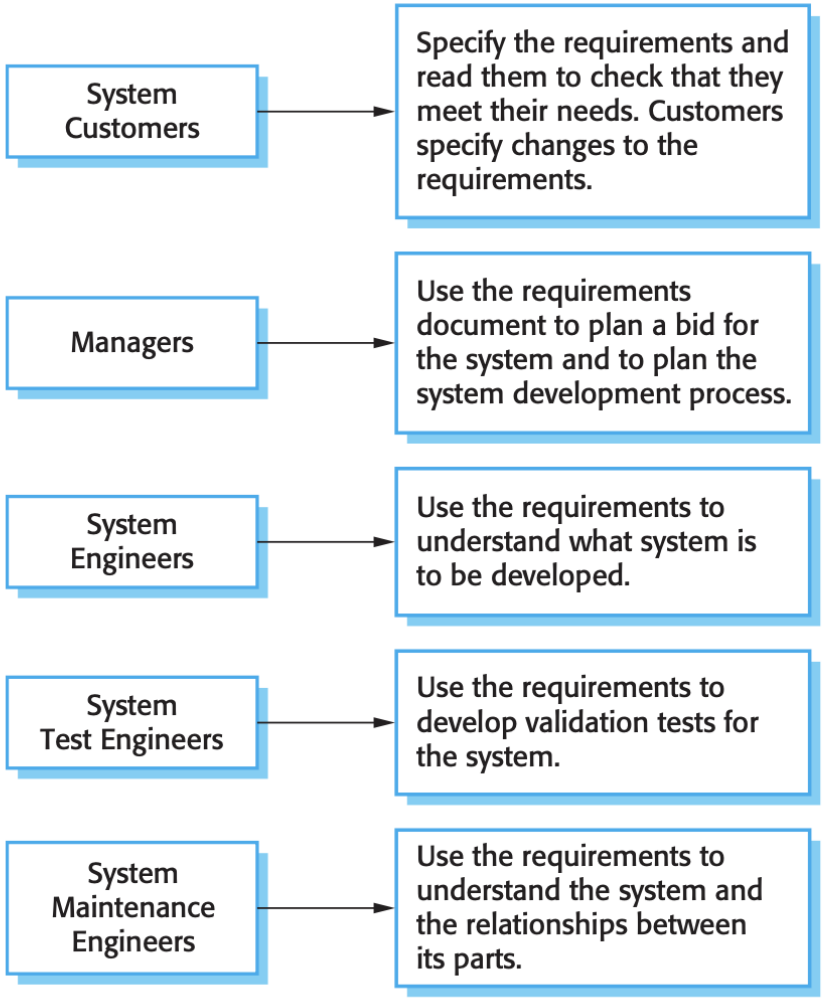

Figure 4.6 Users of a requirements document

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.

4.4 Requirements engineering processes

requirements engineering processes may include four high-level activities. These focus on assessing if the system is useful to the business (feasibility study), discovering requirements (elicitation and analysis), converting these requirements into some standard form (specification), and checking that the requirements actually define the system that the customer wants (validation).

4.5 Requirements elicitation and analysis

After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis. In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

The process activities are:

1. *Requirements discovery* This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity. There are several complementary techniques that can be used for requirements discovery, which I discuss later in this section.
2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.
3. *Requirements prioritization and negotiation* Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.
4. 4. *Requirements specification* The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced, as discussed in Section 4.3.

4.5.1 Requirements discovery

For example, system stakeholders for the mental healthcare patient information system include:

1. Patients whose information is recorded in the system.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Healthcare managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

4.5.2 Interviewing

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions. Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a predefined set of questions.
2. Open interviews, in which there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

4.5.3 Scenarios

People usually find it easier to relate to real-life examples rather than abstract descriptions. They can understand and criticize a scenario of how they might interact with a software system. Requirements engineers can use the information gained from this discussion to formulate the actual system requirements.

At its most general, a scenario may include:

1. A description of what the system and users expects when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how this is handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario finishes.

4.6 Requirements validation

Requirements validation is the process of checking that requirements actually define the system that the customer really wants. It overlaps with analysis as it is concerned with finding problems with the requirements. Requirements validation is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

1. *Validity checks* A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.
2. *Consistency checks* Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3. *Completeness checks* The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. *Realism checks* Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.
5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

4.7 Requirements management

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address ‘wicked’ problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders’ understanding of the problem is constantly changing (Figure 4.17).

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.

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Figure 4.9 Example requirements for the insulin pump software system

4.3.1 Natural language specification

Natural language has been used to write requirements for software since the beginning of software engineering. It is expressive, intuitive, and universal. It is also potentially vague, ambiguous, and its meaning depends on the background of the reader. As a

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 − r1) < (r1 − r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4) If rounded result = 0 then CompDose = MinimumDose

Figure 4.11 Tabular specification of computation for an insulin pump

When a standard form is used for specifying functional requirements, the following information should be included:

1. A description of the function or entity being specified.

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

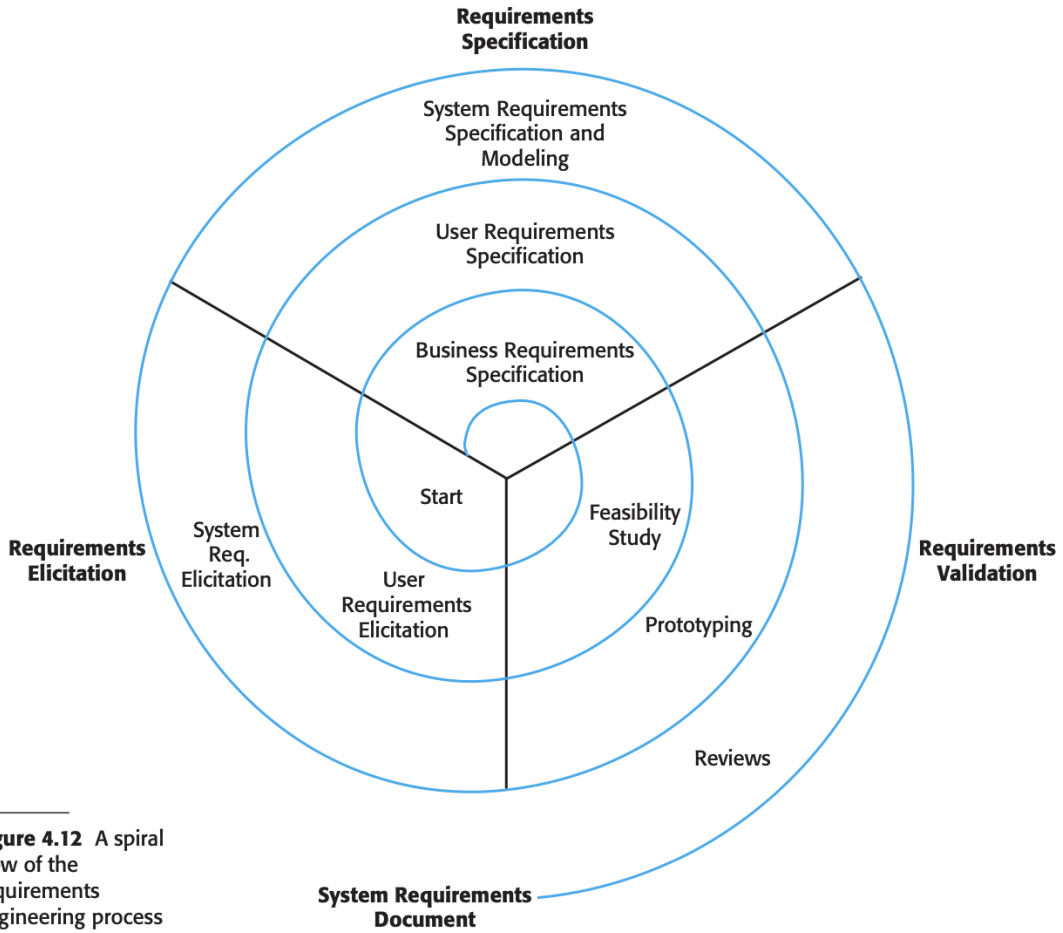

Figure 4.12 A spiral view of the requirements engineering process

Feasibility studies

A feasibility study is a short, focused study that should take place early in the RE process. It should answer three key questions: a) does the system contribute to the overall objectives of the organization? b) can the system be implemented within schedule and budget using current technology? and c) can the system be integrated with other systems that are used?

If the answer to any of these questions is no, you should probably not go ahead with the project.

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

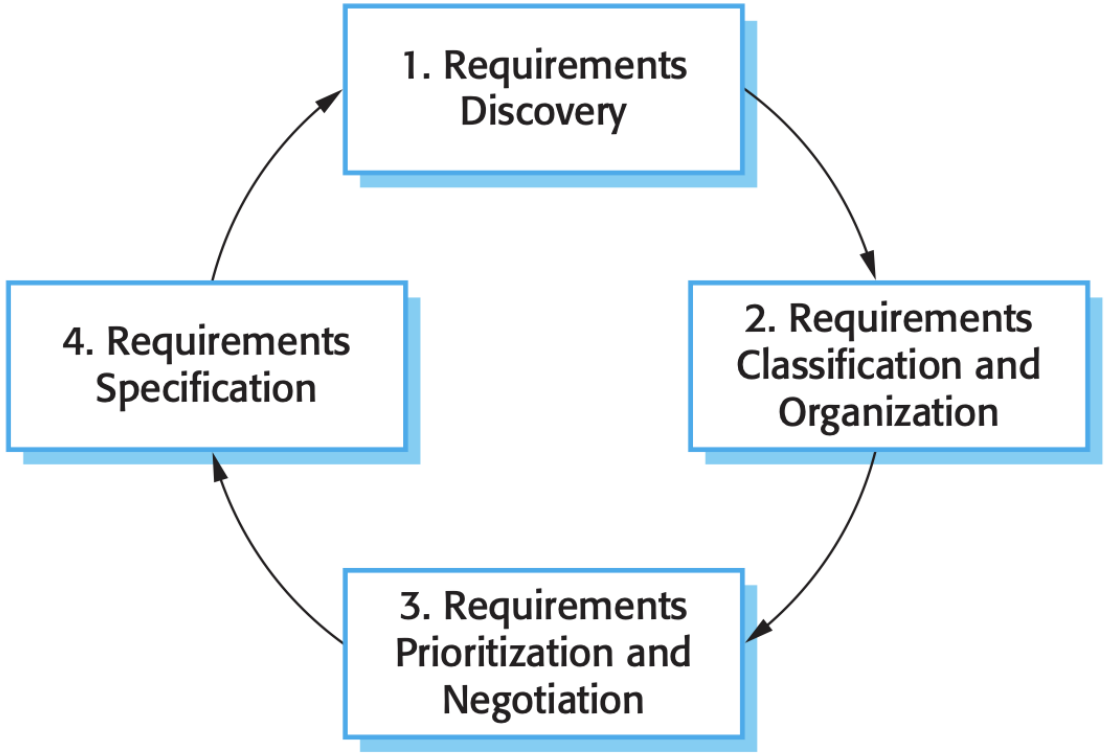

Figure 4.13 The requirements elicitation and analysis process

INITIAL ASSUMPTION:
The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

NORMAL:
The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

WHAT CAN GO WRONG:
The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

OTHER ACTIVITIES:
Record may be consulted but not edited by other staff while information is being entered.

SYSTEM STATE ON COMPLETION:
User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.

Figure 4.14 Scenario for collecting medical history in MHC-PMS

Scenario-based elicitation involves working with stakeholders to identify scenarios and to capture details to be included in these scenarios. Scenarios may be written as text, supplemented by diagrams, screen shots, etc. Alternatively, a more structured