Describing and validating functional requirements as use cases

Sanna Keskioja

Helsinki September 13, 2007
Master’s Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science
Requirements engineering is an important phase in software development where customer's needs and expectations are transformed into a software requirements specification. The requirements specification can be considered as an agreement between the customer and the developer where both parties agree on the expected system features and behaviour. However, requirements engineers must deal with a variety of issues that complicate the requirements process. The communication gap between the customer and the developers is among typical reasons for unsatisfactory requirements. In this thesis we study how the use case technique could be used in requirements engineering in bridging the communication gap between the customer and development team. We also discuss how a use case description can be developed to document functional user requirements. Furthermore, we present how use cases can be used as a basis for acceptance test cases.

ACM Computing Classification System (CCS):
D.2.1 [Requirements/Specifications],
D.2.4 [Software/Program Verification]
Contents

1 Introduction 1

2 Requirements engineering 3
   2.1 Key concepts .............................................. 4
   2.2 Requirements classification ................................. 5
       2.2.1 User requirements .................................... 6
       2.2.2 System requirements ................................. 6
       2.2.3 Requirements classification example .................... 7
   2.3 Requirements engineering process ......................... 9
       2.3.1 Process objectives .................................... 9
       2.3.2 Process phases ....................................... 10
       2.3.3 Elicitation ........................................... 12
       2.3.4 Analysis ............................................... 13
       2.3.5 Specification .......................................... 14
       2.3.6 Documentation ........................................ 15
       2.3.7 Validation ............................................ 16
   2.4 Stakeholders .............................................. 17
       2.4.1 Identifying stakeholders ................................ 17
       2.4.2 Managing stakeholders .............................. 20

3 Use-cases in requirements engineering 21
   3.1 Use case concepts ........................................ 22
   3.2 Use case pitfalls ......................................... 28
   3.3 Use case structure ....................................... 31
       3.3.1 UML use case diagram .................................. 32
       3.3.2 Dimensions of a use case description .................. 35
       3.3.3 Use case building blocks ............................ 38
4 Use cases in system testing 44

4.1 System testing .................................................. 44
4.2 Use cases and early test design ............................. 46
4.3 Basic elements of use case based testing .................. 48
4.4 Transforming use cases to test cases ..................... 50
4.5 Constructing extended use cases ............................ 53

5 Case: Dressage scorer system 56

5.1 Problem domain analysis ....................................... 56
  5.1.1 Overview to dressage .................................... 56
  5.1.2 Competition context ...................................... 57
  5.1.3 System boundary definition .............................. 60
5.2 Example use case description ............................... 63
5.3 Example extended test case ................................. 68
5.4 Reflections on the example ................................. 70

6 Conclusions 73

References 75
1 Introduction

Software engineering is often regarded as a hard discipline compared to other engineering disciplines. Many textbooks explain this by the abstract nature of the end product. Software projects, as any engineering projects, involve also people who may not possess the same level of technical expertise as those directly involved in the design and the implementation of the system.

For people outside the software engineering field it is often difficult to understand what is concretely going on in the development process. They may not understand what makes the code actually work, or be aware of the possibilities of various technologies. On the other hand, their vision of the end product can be unrealistic, and they may expect the developers to build something that is not possible with the current knowledge.

However, as software generally exists for a purpose based on real human needs, involvement of different interest groups and individuals is needed, not least for their domain knowledge. As the parties’ backgrounds often are diverse, and the software product is initially abstract, communication becomes a key issue. Developers with computing science background tend to favour precise, unambiguous expression, whereas the customers may only be used to informal natural language articulation. In the absence of touch and feel, this incoherence of communication opens up room for misunderstanding. This communication gap can retard the whole development process, or lead to software that is not compatible with the customers’ or users’ expectations or actual needs. Therefore finding a means of communication that enables the developers to interpret the customer’s often implicit needs, and turn them into clearly expressed requirements is very important. The communication method must also be suitable for conveying the requirements to system designers and programmers. The challenge is to capture the requirements in a way that they can validated to match the customer’s needs. The requirements should also be captured in such a detailed and unambiguous manner that allows them to be transferred via system design and implementation into a system that still matches the customer’s needs and wants.

In this thesis we study how the use case technique can be used in the requirements engineering process to capture system’s functional requirements. We also discuss how use cases can help in bridging the communication gap between the customers and the requirements engineers. Furthermore, we explore how use cases could be
used as basis for validating the implemented system’s compliance with the customer’s needs and expectations.

The rest of the thesis is organised as follows. Section 2 describes the requirements engineering process, and the role of requirements engineering in software development. Section 3 presents the key definitions related to use cases, and central principles for structuring use cases. Section 4 discusses using use cases as basis for system validation, followed by an example in Section 5. Section 6 concludes this thesis.
2 Requirements engineering

Every software development project is a fusion comprising of various factors re-shaping over time, such as a mix of various stakeholders and evolving business environment. The stakeholders represent the various individuals, institutions and organisations that have some kind of association with the customer or the developers.

There is also a lot of variety in the role, type and criticality of the software. The role of software refers to the purpose of the software in its environment. Software may for example assist in expanding the customer’s operations to new product or service areas. On the other hand, software may be developed to rationalise existing, potentially old fashioned business processes. The type of software refers to the fashion in which the software is implemented. Different types of software expand from real time distributed systems to an embedded system in an electronic device. The criticality of software refers to the level of importance of a system functioning as intended all times, and the need for a swift system recovery. The level of criticality varies from highly safety-critical software used for example in hospitals and air traffic control to non-critical applications, such as entertainment and games.

The new software might have to be integrated to existing databases and software already in use. On the other hand, some of the functionalities could be readily available by acquisition of commercial-off-the-shelf (COTS) software. The new application might become part of a product family and therefore would need to conform to some joint qualities. Commonly it might be a case of product evolution, or a case of building a completely new software from the scratch. All these multiple combinations of variables add complexity to the already difficult problem solving task of building software.

Requirements engineering aims to find clarity in this complexity with techniques that help in outlining the specific context and narrowing down the customers’ mass of often vague and implicit needs. In this section, we first go through some key concepts relating to requirements engineering, and especially to requirements classification, and to the scope of the project. Subsection 2.2 discusses the requirements engineering process. Stakeholders and their importance to the requirements engineering phases are described in Subsection 2.3.
2.1 Key concepts

Requirements engineering is an area that does not only apply methods from software engineering, but also utilizes knowledge and tools from such fields as cognitive science, psychology, and business management. In addition to understanding the technical aspects of software development, it is important to grasp the related socio-economic impacts and business dynamics. Part of requirements engineering is the system boundary identification. This involves identifying the problem that needs to be solved by the new system, as well as determining where the system will be placed [NuE00].

Often there are many ways to solve the identified problem in the business environment. The solution may not need to be technical, but if it is, it is up to the requirements engineers and the software developers to decide which technical solution fits best. The best fit is determined by the characteristics and constraints of the environment of use. It is not possible to define the best solution unless one understands the context of use, the users and other stakeholders, and their relationships. Important related aspects also include understanding the decision making process and people involved, the customer's current and future business goals, as well as the related economics. The goals and needs derived from the operating environment are formulated into requirements in the requirements engineering process.

The main source for requirements is the environment where the new system will be located. The extent to which the requirements for the new system are stipulated by the customer prior to engaging the development team depends on the project. Regardless of the moment when the development team's requirements engineers get involved in the requirements process, it is important that they get a thorough understanding of the environment and the conditions under which the new system is expected to operate. Even though the customers may be clear on what they want, this information must be interpreted into a detailed and precise form.

In the following subsection we discuss the requirements and their classification. We also give an overview to the the requirements process, as well as to stakeholder identification and management. Before that we define some key concepts, business domain, problem domain, and business rule, which are related to the analysis of the environment of the future system.

**Business domain.** Building a system is a decision based on some problem recognised in the customer’s business environment. This environment is denoted as
the business domain. The business domain demarcates the scope of an organisation's operations and competitive boundaries. The internal stakeholders of a business domain include the employees, business units, and departments that are part of the same entity. The business domain also includes connections to external stakeholders, such as suppliers, legislators, and customers.

Problem domain. Problem domain is that particular part of the business domain where the new system will be placed. The problem domain delimits the business domain based on the scope of the problem, and provides more detailed information than the business domain about the requirements of the new system.

Business rule. Business rules denote the operative tasks within the problem domain. Business rules describe the processes that take place in the quotidian business operations of a company. Business rules implement the higher level strategic plans, and relate, for example, to how the company handles customer invoicing, sales or manages production.

2.2 Requirements classification

A requirement is a general concept that refers to a collection of information that describes the functions and the qualities that are expected from the system under construction. The requirements are based on real-life needs derived from the organisation's business environment including the stakeholders. There are several definitions for a requirement in the literature. The word seems to refer to almost anything that somehow describes what a software is expected to do. In this thesis the word requirement is used in two ways; first, according to the abstraction level and second, according to the content. Grouping of requirements according to their abstraction level divides them into user and system requirements [Som06]. Grouping of requirements according to their content divides them into functional and non-functional requirements, and domain constraints.

The requirement classification types are used in a somewhat confusing manner in the literature. User requirements have for example been used as a term to describe the functional requirements that are based on the user goals, and elicited with help of the use cases [LeX99]. The common conception today however is that all requirements should initially be determined based on what the customer and the users want to accomplish with the use of the software [Coc02].
2.2.1 User requirements

User requirements refer to the behaviour and effects that the system is expected to have in its problem domain. User requirements are a high level description of the problem the new system is built to solve. The name ‘user requirement’ is appropriate as they describe the solution from the user’s point of view, with no reference to the new system [Bra02, Som06]. In fact, user requirements should be described on such a general level that may allow solving the problem even without software.

User requirements, along with the problem domain description, are documented in the requirements document. The requirements document is an output of the requirements analysis phase (Subsection 2.3.3). The purpose of the requirements document is to provide the requirements engineers with information about the problem domain, as well as to provide them with the related data to give them basis for constructing the more detailed system requirements.

The user requirements can be categorised to functional or non-functional requirements, depending on whether they describe expected software behaviour or quality. The functional requirements describe the inputs the system should be able to react to, and its expected behaviour in those situations [Som06]. A viable requirements technique especially for highly interactive systems are the use cases, where system responses to user inputs can be systematically captured. The non-functional requirements are more complicated to express. The non-functional requirements relate to the quality aspects of the expected system behaviour, such as performance, usability, and safety. Domain constraints are requirements that instead of representing specific user needs may set bounds to some functionalities or determine how particular output values must be computed. The categorisation of the domain constraints is not that clear. They could either be functional or non-functional in nature, or clear design constraints. Generally, we would rather group them separately as domain constraints to support traceability and modifications in case of future changes in the environment.

2.2.2 System requirements

System requirements are more detailed descriptions than the user requirements of what the system should do [Som06] detailing the system’s services and constraints in a precise way. System requirements are an output of the requirements specification phase (Subsection 2.3.5). Requirements specification is a step further from
understanding and analysing the context towards designing desired external system behaviour. System requirements attach the problem solution to the system, and represent the external behaviour the requirements engineers have invented based on the user requirements.

The important building blocks at this stage are the inputs and outputs to and from the system, and their relationships. The system requirements are however not concerned with the internal structure of the system. In the literature the word ‘specification’ is often used both in relation to the requirements specification and the internal design of the software. The difference, though, is that the requirements specification deals only with the external behaviour taking a black-box view of the system [Bra02].

2.2.3 Requirements classification example

Applying the user and system requirements, and adapting an example from the literature [Coc02], we group the requirements according to their context and vicinity to the system. This thinking is based on the notion that all software behaviour should be based on the business and user goals [Coc02].

The primary driving force for professional software development comes from the customer’s business environment. The customer has a problem that needs to be solved, or they want to improve their existing process. Table 1 presents a top-down perspective to system goal derivation. The business domain is an operating environment of an imaginary media company. Their business is to produce and publish content for a magazine, and their operations are financed by subscription fees and advertising sales. We assume here a situation where their shareholders’ expectations for increasing profits have to be addressed to. The finance department will look into streamlining the cost structure, but also the advertising sales need to be improved. The sales director analyses the sales department’s operations, and notices that there is no systematic way of contacting the customers. He imposes a new business rule to the sales people that every customer must be contacted on a regular basis, the frequency depending on their priority as a customer.

This new rule presents a problem to the sales people. The customer base is so large that it has been difficult to keep track of all the customers and the times when they have last been spoken to. Also, the smaller customers are handled by an external telesales company. That means such a customer has no assigned account manager,
Table 1: Each domain is a subset of a wider context. The goals of the various subsets are derived accordingly.

but he or she is every time time contacted by a different sales person. The sales director discusses the options with the rest of the management team, and they decide that a customer relationship management system is needed to manage the customer data, and to enable following-up the sales efforts. The new system would replace the old inconsistent system where contacting the customers has been up to the activity of each individual sales person.

The sales director’s goal brings the discussion from the general business environment to the specific problem area, where a software system could be considered as a potential solution. We call this type of a goal a transition goal, as it changes the focus from the surrounding environment to the system level. In this example the wide scope of the whole business environment has now been narrowed down to a specific subset of the sales department problem domain. The sales director and his sales people will become the source of specific user requirements based on the company’s business rules and their individual preferences. User requirements are turned into system requirements through requirements engineering activities. The system requirements are attached to a specific way of solving the media company’s problem.

We find that this type of grouping of the project’s domains and goals could be useful also in practice. It gives an idea of which level of information is interesting to each of the participants of the project. It also matches conceptually the software testing V-model where one proceeds from larger system level perspective to the more detailed code level. This type of grouping could also give an idea of when to use each level of information in the process cycle.
2.3 Requirements engineering process

The major source for requirements are the stakeholders that operate within the problem domain and its surrounding business environment. The requirements on the other hand are the primary input when determining the behaviour of the system [JBR99]. Failing to specify the requirements can lead to software that does not do what the customer wanted it to do or does it inefficiently [WLL01]. The requirements engineering process defines a structured set of activities that help the stakeholders and the developers to understand and document the potentially complex issues related to system development [Som05].

2.3.1 Process objectives

In industrial software development a process should be both effective and efficient. It should be effective in a way that the projects are successful and achieve the wanted goals, and efficient in a way that the goals are reached without wasting resources. A shared process model in the company facilitates common understanding of the subject, as well as communication between the stakeholders. The process model should be designed in a way that supports continuous process development. As technology and business environment evolve continuously, managing and adapting to change should also be incorporated into the process model. The process model should also support managing the process itself. One cannot determine process success or failure unless some metrics are applied to provide points of comparison. The process model should also include decisions on how to automatise the project management. It should also be determined how the automation tools will be managed [CKO92].

The few available empirical studies within software companies indicate that the requirements engineering processes used in companies are insufficient or missing [NSK00]. This is true especially in the companies whose main business focus is not in software development. The main deficiency was the lack of requirements specification. Later empirical studies indicate that the requirements engineering processes that companies apply have been improved. However, lack of a systematic process continues to be a major problem in software development [VeE05].

In a recent survey 48% of the problems in development processes were found to be caused by problems in requirements engineering [HBR02]. However, it was also found that many of the problems were not directly related to the requirements engineering per se, but to other issues in the company. One major issue is inadequate
communication among the developers. Problems also originated from insufficient skills and resources. Even though these issues are part of the general company culture, we feel they are vital to the success of the project, and must therefore be addressed explicitly also in the requirements engineering process. Project success factors include finding skillful project managers who are empowered and have a broad background rather than expertise in a narrow area. A good project manager also has good people and communication skills, and the ability to estimate schedules and priorities [VeE05]. Not every person is perfect in each of these aspects, but a good process model and project techniques can help the project manager to improve.

The requirements engineering process should be chosen according to the type of the application being developed. Other influencing factors are the company’s size and culture [Som05]. When new methods are introduced to improve the process, one should take into account the receptivity and experience of the personnel, as well as their skills. One should also consider the time and resources available. Advanced requirements engineering practices should not be introduced until a good understanding and application of the basic practices has been accomplished [SoR05].

2.3.2 Process phases

Core requirements engineering activities can be described as phases where requirements are elicited, analysed, specified, documented, and validated. Communicating, negotiating and agreeing upon the requirements are also important tasks in the requirements engineering process, as are managing and evolving the requirements through the product life-cycle [Bra02, NuE00, SoR05]. One should also be prepared to resolve conflicts. Conflicts can relate to inconsistent or conflicting requirements, or conflicting goals among the stakeholders [LBJ04, MCY99, NuE00]. Even though these requirements engineering activities are listed in a particular order, in practice the phases are interleaved, iterative, and may span the entire software systems development cycle [NuE00].

The model in Figure 1 illustrates the requirements engineering activities as iterative steps organised around a spiral [Som06]. The goal of the process is to create a system requirements document. In the figure the process progresses clockwise starting with problem recognition in the business environment at the customer’s end. The customer, the development team, or some consultant produces the business requirements through business analysis. In the feasibility study one analyses the advantages and disadvantages of various solution alternatives. Once the decision
has been that the solution to the problem will be achieved through development of new software, the actual requirements engineering activities, elicitation, analysis, specification, and validation, begin.

Figure 1: Requirements Engineering process [Som06].

As the requirements have such an important role in software development, it is important to identify possible problems that may occur in the requirements engineering process. Without an understanding of what can go wrong in each process phase, it is difficult to improve the performance. In the requirements process, most requirements are derived from the application domain. Communication problems encountered in the process concern for example conflicting interpretation of requirements, and inadequate interaction between developers and the customer. Also social and organisational problems in the process affect the quality of the requirements. These include lack of stability in the development team, poor user participation, lack of mutual understanding, lack of trained personnel, and different views of the system. Additional problems have arisen due to organisational politics and technical
problems [SEM99].

Problems in the process reflect in the outcome of the requirements engineering process, the requirements specification. It may be that the requirements are not documented extensively enough. This implies that the requirements could be incomplete or missing. The requirements could also be erroneous and a result of wrong interpretation of the goal or the need behind the requirement. Also data expressed in the requirement could be incorrect. In the case of missing requirements some of the information could have seemed so self-evident to the requirements engineers that they forgot to document it. Hence, this information will not be transferred to the designers and programmers.

The requirements could also be documented correctly, but still the developer makes a wrong interpretation. It is up to the process designers to minimise the probability of these problems. Especially improving the study of user tasks and early prototyping have proven to be successful requirements defect prevention techniques [LaV01]. There is a myriad of techniques available for each requirements engineering process phase, including various elicitation, specification, modelling and verification and validation techniques. Unfortunately many of the techniques have not found their way from the research community to practical use in software companies.

The following process phase descriptions follow mainly Bray's [Bra02] and Sommerville's [Som06] work. Both Bray's and Sommerville's books present the requirements engineering process in a way that is in line with the common perception in the requirements engineering literature. In addition, especially Sommerville's work has been widely cited. There are variations to the definitions in the literature, but the main tasks that need to take place in the requirements engineering process are the same.

2.3.3 Elicitation

_Elicitation_ stands for the phase in the requirements engineering process where the developers and the customers work together to obtain information about _the application domain, the services that the system should provide, and the system’s operational constraints_ [Som06]. The purpose of elicitation is to provide the requirements engineers with data needed to gain adequate understanding of the context including related business processes, stakeholders, and problems the new software is expected to solve. Finding out the goals or problems that need to be solved, facilitates iden-
tification of the system boundaries [NuE00]. In fact, requirements elicitation should be guided by business concerns and goals [SoR05]. Elicitation and analysis are often interleaved and cyclic; as the understanding of the problem domain increases, more specific data is needed. Among popular elicitation techniques are scenarios and use cases, focus groups, informal modelling [NeL03], and throw-away prototypes.

Customers express needs and requirements for the new system in various levels of detail and precision depending on their familiarity with software engineering. As the real-world needs are not that obvious in all cases, the developers must make sure that they ask the right questions and use all possible information sources to ensure that they understand the business domain and the role of the new system in it. The questions should cover information regarding the customer’s business environment and goals. The elicited information should also cover potential constraints related to the project or the future system. The client may for example have legal obligations with regard to data protection. Also technical constraints must be outlined. The new system may have to run on existing hardware, or interfaces to existing own or third party software may be required. Also needed and available software licenses must be uncovered.

The person doing the elicitation should have strong interpersonal skills. People involved in the project may have their own personal agendas. Some may be afraid of the change, whereas others may see the project as an opportunity to enhance their career and gain more power within their organisation. This means that the requirements engineer will face various attitudes that may bias the elicited information.

Output of the elicitation phase comes in various forms of notes regarding the requirements. They can be in the form of formal technical documentation, notes scribbled on a corner of a paper or recordings of actual usage situations, in practice in any form that enhances understanding issues related to the future system.

2.3.4 Analysis

There are three goals for the development team in the analysis phase: to gain thorough understanding of the problem domain and related problems; to establish the user requirements as the basis of the future system behaviour; and to produce the describing documentation [Bra02]. Typical approach into gaining understanding of an often large problem area is to break it into a set of subproblems. Depending on when in the development phase the customer decides to involve the requirements
engineers, the problem area can span from the customer’s business domain to an already defined problem domain. Modelling is an example of a technique that helps in defining and analysing the subproblems.

Generating models is an important requirements engineering activity. Models provide an abstract description of the subject of modelling, and serve for example as means for communication and validation in the intertwined requirements elicitation and analysis phases. The modelling approaches can be categorised according to the type of information one wants to model. The categories are organisation modelling, data modelling, behavioural modelling, domain modelling, and modelling of non-functional requirements [NuE00]. Organisation modelling captures the reason why a system is being developed, the types and amount of data that the system is expected to handle, the expected behaviour of the system, and the ways of using the system. Data modelling relates to the data that the system is expected to handle. Behavioural modelling relates to the expected behaviour of the new system, including the interaction with the future users. Domain modelling produces an abstract description of the environment where the new system is expected to operate. Non-functional requirements are often difficult to express in a way that is measurable, and there is still ongoing research on how they should be modelled. There are several modelling techniques available, and one should choose the one that fits best the process model, analysis method, and type of application in question. The suitability of a technique for the task at hand is not always obvious. One can base the decision on the personal experience gained in previous projects, or on experiences reported by other requirements engineers within the company or research community.

Discussing analysis methods is an extensive subject. Methods span from structured analysis to object-oriented analysis. There are also goal-oriented approaches that are based on the objectives of the business and the stakeholders. The object-oriented techniques for example work well when comprising models of the future system’s operational environment including the stakeholders, work processes and related software systems. The object-oriented techniques have been perceived not to address so well the non-functional requirements relating to customer’s objectives, and a goal-oriented approach is suggested instead [MCY99].

2.3.5 Specification

Specification refers to the act of inventing system requirements that match the user requirements described in the requirements document. The output of this phase
is the requirements specification document. It contains a description of the system behaviour that will solve the problems defined in the analysis phase. System behaviour consists of the inputs to the system, the outputs from the system, and the relationships between them [Bra02]. Anything relating to how the behaviour is implemented is left to the internal design phase later on in the development process.

The specification can be documented by using three levels of formality: informal, semi-formal, and formal. An informal specification is simply written by using a natural language. This kind of specification is easy to write. A downside is that natural language specifications often miss structure and may be highly ambiguous. Formal languages are notations and languages which have their basis in mathematics. They are precise, but can be difficult to grasp for a layperson. Semi-formal techniques, such as use cases and task analysis, offer some structure, but still leave room for individual expression [Coc97]. Semi-formal techniques may also be easier for a layperson to learn than formal techniques.

Specification is performed by the developers based on the knowledge gained in the elicitation and analysis phases. The purpose is to inform the designers, programmers, and the testers about what they need to accomplish. On the other hand, the specification is targeted at the clients and the users. They must be able to validate that the requirements engineers have understood the problem correctly, so that the developers can proceed with the project.

2.3.6 Documentation

Documentation is one of the key activities in the requirements engineering process, and it requires writing the requirements down in such a way that they are understandable to the software developers and other stakeholders [Som05]. At the customer's side, the possible stakeholders in a software development project include the management and the people in finance and operative user level. From the software company's perspective, important stakeholders are project management, user interface designers, testers, and programmers. They all need different types of information of the project to be able to perform their own project responsibilities, and different techniques are needed to describe all needed views. The business people in the customer's end may only have need for a general business perspective of the project. They may only want to know the functions of the new system, and the effects it is going to take in the business environment, and have no interest in the details of the system behaviour.
Understanding of what needs to be described in the requirements documentation is gained through analysis of the elicited information. Documentation is laborious and time consuming, albeit necessary. One should prepare in advance for the kind of information that will be needed later in the development process. The form of documentation may be chosen to support for example traceability of the requirements throughout the software development project, reusability of requirements, or test case derivation.

The knowledge about the problem domain and the requirements for the future system is learnt through several forms of communication, such as personal meetings and documentation. Deciding what to document should be a conscious decision, and not every piece of information need to be documented. The documented information transfer is aided by models and written descriptions. The abstraction level must be chosen to meet the needs of the audience and the status of the development phase. The terminology should stem from the problem domain to enhance the involvement of the domain experts who often represent stakeholders from the customer side.

2.3.7 Validation

One difficulty in the requirements engineering is that it is very much a cognitive process. The outcome of the requirements engineering efforts materialises only later in the process when the completed code is executed. That implies that most of the discovery and invention taking place in requirements engineering has no concrete manifestation. Furthermore, understanding the transferred information is very much subject to individual traits and abilities. The purpose of validation is to decrease this ambiguity by applying various techniques throughout the requirements engineering process. The goal is to ensure that the problem domain, the requirements, and the derived functionalities have been described in an adequate level of detail and abstraction, and are understandable and meaningful to the target audience.

The techniques for requirements validation are the same as in any software related validation. They include, but are not limited to, checks, reviews, logical analysis, use of prototypes, and use cases. The most important aspect is to maintain an open conversation between the stakeholders and apply techniques for information transfer that are powerful enough in expression and understandable to the relevant parties.
2.4 Stakeholders

Stakeholders are the key source of user requirements, and they have a varying degree of influence and interest in the undertaking of building a new system. Missing some group of stakeholders will lead to incomplete or defective requirements. Identifying the stakeholders helps to recognise the interfaces between the system and the surrounding environment. The more critical the system, the more important is active stakeholder participation [Bra02].

Experiences in software projects have given evidence that a common concern for the developers is finding the right stakeholders to discover, specify, and test the requirements [AIR04]. It is important for the requirements engineer to understand how people interact with each other, how they perceive and understand the surrounding world, and how their actions are affected by the surrounding environment [NuE00].

A major question is which of the stakeholders should be engaged in the project and when. It is also important to recognise which of the stakeholders are the sources and which are the users of information, and in which project phases their involvement is needed [AIR04]. Requirements engineers are very much involved in many stakeholder related aspects. Those aspects include stakeholder recognition, involvement and commitment; managing stakeholders’ expectations and resolving conflicts; and requirements communication and negotiation.

2.4.1 Identifying stakeholders

Stakeholder identification starts before the project has officially been commissioned. First steps are taken in the initial conversations between the client and the developing organisation. This negotiation of cooperation and decision to cooperate on a software project narrows down the number of potential stakeholders that might be worthy of attention. The potential stakeholders to the system are recognised through analysis of the business domain. If a group or an individual has no involvement with the business domain, logically they cannot be a stakeholder to the problem domain, which is a subset of the business domain. Further analysis of the problem domain further delimits the number of stakeholders that are relevant to this particular project. In system development, we can distinguish between those stakeholders who have financial or managerial responsibility in the system procurement process and those stakeholders who have knowledge related to the actual operative usage of the system [SFG99]. This distinction is important, as the two groups
represent different types of influence applicable in different development phases.

The stakeholder theory has its origins in the area of strategic management, and it expands to all aspects of an organisation. In this thesis we focus only on the stakeholders within the problem domain, regardless of the unquestionable influence that the stakeholders outside the problem domain have on the existence and the success of the project.

In stakeholder identification, one should decide upon two things: First, the criteria which determine why a certain group should be considered a stakeholder and second, under which conditions these groups are regarded as stakeholders [MAW97]. Additionally it should be decided how to prioritise the stakeholders and the related requirements or constraints. Stakeholder theory is not an exact science, and there is no model that would be applicable in all situations. However, there are some guidelines that support stakeholder identification. It is advisable to observe existing practices, or to study designed future work processes for any new stakeholder roles [SFG99]. Iteration between the requirement elicitation and analysis phases should provide the information to allow discovery of these roles. Analysts should be able to categorise stakeholders into groups that are directly involved in the development project, or are future users of the new system. They should also identify and group those stakeholders that are part of the larger business context: the decision makers, and those with financial or legislative power.

One should also avoid making assumptions, but study a whole cycle of business rule activities to gain thorough understanding of the context and related issues. For each development phase one should plan ahead which stakeholders are needed. As the software development cycle can be long, even years, one must be prepared that the individuals, various groups, and their priorities will change over the time. Stakeholders are not only on the client side; just as important are the roles within the development team. It is a duty of the project manager to manage both.

Figure 2 depicts how stakeholders can be grouped according to their immediate vicinity to the project. In this onion model [AIR04], each layer acts as context to the inner layer. The closer the stakeholder is to the kit, the more likely they are a source for requirements and use cases. The kit refers to the system under construction, and the socio-technical system refers to the problem domain. The dotted lines refer to some service or information that the stakeholder provides or receives. The stakeholders in the problem domain are potential use case actors.
Figure 2: Grouping stakeholders according to their vicinity to the system under development [AIR04].

Now we have established that the stakeholders can be categorised according to their vicinity to the new system. Stakeholders can also be prioritised according to their subjective weight to the project. They could be clustered as primary and secondary stakeholders according to their influence power and importance in getting the product right. Influence can be social and material power, which means the ability to achieve the desired outcome. More weight could also be gained through legitimacy. That means that the stakeholder in question has something at risk, such as money or ownership, or has moral or legal claims. Higher priority at a time could also be gained by urgency, which means time-sensitivity of the relationship or the claim [MAW97]. Criticality and prioritisation are subjective matters, and dependent on the situation where applied.
2.4.2 Managing stakeholders

Managing stakeholders is an area that requires good interpersonal skills from the requirements engineer. Skilled stakeholders are not only difficult to discover, but once found, keeping them committed is another challenge. If the individuals do not have any personal gain in the project, let it be financial or social, it may be hard to keep them interested and involved. It is really the art of a skillful project manager to ensure there is an adequate communication strategy to keep people informed and involved. This is not an easy task. It helps when one understands what motivates people and how they want to win. The project manager should establish reasonable expectations, manage them, and allocate such tasks to each stakeholder that matches their win-conditions [BoR89]. A win-situation simply is a state in a person’s life when he or she feels satisfied with the outcome; feels like a winner [BoR89].

Considering the potentially large number of stakeholders, it is likely that differences of opinion will rise. Various groups have different winning conditions. Also, in every organisation, there is a lot of politics involved in decision making. Everyone may not be satisfied with what they get. The optimal result should however be a balance between the stakeholders’ win-conditions, to create an overall win-win-situation. Communication is a key issue in avoiding and solving stakeholder conflicts. The communication gaps between the client and the developer should be avoided by using unambiguous techniques.

The success of the development process is often determined by how well the needs, the goals and the constraints absorbed from the stakeholders have been described and formulated into requirements. They should be documented in such a way that they can be formalised and transformed into a working program [JBR99]. Since the customers and the end users have such an important influence on the success of the system, the chosen specification technique should support communication and capture the requirements in a way that can be validated by the users.
3 Use-cases in requirements engineering

*Use case* is the name of a technique used for eliciting information about future system behaviour and interaction between the user and the system. Use cases can also be used for specification of functional requirements. Well formulated use cases can also be used for requirements validation throughout the iterative and incremental requirements engineering process. Use cases consist of alternative courses of action that can take place while the user attempts to reach his or her goal by using the system. The courses of action are generally called scenarios, and are included in the use case description. In the simplest version of a use case, there is only a short natural language narrative describing what the user wants to do with the computer system. There are also templates available that offer more structured and systematic ways for describing use cases. Use cases can be accompanied by various diagrams that provide the reader with more information of the scope and contents of the use case. Such diagrams include for example UML use case and activity diagrams. Alternatively, use cases can be described by using some diagram that is suitable for presenting control-flow complemented with textual descriptions of the concepts.

Use cases were introduced systematically in the literature in 1992 by Ivar Jacobson in his book *Object Oriented Software Engineering* [Jac92]. Jacobson’s idea was to use the use cases as a tool in object-oriented software engineering. In his methodology, use cases describe the system functionality as object classes that could be instantiated and that could interact with various user groups, or actors as they are called in the use case context. In his vision, use cases can be applied throughout the whole development cycle linking the requirements, the implementation and the user’s acceptance [Jac92, JBR99].

Use case modelling in requirements engineering increased in popularity in the late nineties, and it has become a very popular tool in performing software requirements analysis and specification [Fir99, NeL03]. In addition to having been relatively well adopted in the industry, use cases are also perceived as a relatively simple tool that can also be acquired by a non-technical person.

Use cases have many advantages that support their use in requirements engineering. However, they also have well-known problems that should be taken into consideration in the development process. One should even be prepared to acknowledge that in some cases use cases may be found completely useless and therefore should not be applied at all.
3.1 Use case concepts

According to the UML 2.0 standard, use cases are a means for specifying required usages of a system [Omg06b]. Alternatively expressed, use cases describe what happens when actors interact with the system [Coc06]. The UML use case diagram enables modelling the system’s actors and their relationships, and specifying required usages of a system. The use case diagram is one of the three types of diagrams included in the UML 2.0 standard that can be used to model behaviour. The other two are the activity diagram and the state machine diagram [Omg06a]. The use case diagram is however not enough for capturing the requirements of a system. In the standard, the term 'use case' in fact refers to a use case type, and the instance of a use case refers to behaviour that matches the use case type. In principal, one could use any format best suitable for conveying the use case’s functionality, details, and flow of control to the reader [PiP05]. Typical formats are more or less structured natural language, activity diagrams and state machines.

The core idea of the use case diagram is still based on Jacobson’s initial work on use cases. The use case diagram uses the original notion of actors and use cases. The actors are pictured as ‘stick figures’ and denote what exists outside the system. Use cases are pictured as ellipses and denote what should be performed by the system [Jac92]. Initially, the actors were perceived only as humans interacting with the system, but today the notion has extended also to other systems [Coc97].

Even though the contents of the use case diagram are defined in the UML standard, there is no similar clarity in the industry of the use case description format. As we first began the process of writing this thesis, our intention was to present how the use case technique is applied in practice and compare different templates and their suitability in the requirements engineering process. We expected to find articles that present experiences of projects where the use case technique had been applied, and compare the results. Unfortunately, we ran into trouble quite early in the process, and instead of discussing and evaluating the forms use cases take in practice, we had to settle for envisioning what the factual content of a use case should be, and what kind of aspects should be considered when choosing the presentation format.

There are two main reasons why we needed to change our direction in this thesis. First, there is no standard use case description format available. The UML use case applies only to the UML element and its name. Several approaches that are called ‘use cases’ have sprouted due to the lack of a precise and commonly accepted definition [CoF98]. Furthermore, there is not even a clear consensus among the
authors of what use cases are versus scenarios. Articles that discuss development
methods like 'use case driven development', 'scenario-based development' or 'goal-
based development' all seem to refer to the type of development process which
is based on the needs and objectives derived from the environment where the new
system will be placed. However, often in the articles the words scenario and use case
are used interchangeably. Especially the word scenario is burdened with different
definitions varying from a description of a course of events that can take place
in a system to a short description of the usage of a system. For example Alistair
Cockburn has found 18 different definitions of a use case varying by purpose, content,
plurality, or structure [Coc97]. In another article, scenarios where found to vary
according to their form, purpose, content, and life cycle [WPJ09]. Even though
the latter article discusses scenarios, not use cases, according to our interpretation
the scenarios in the article are very close to what we perceive as belonging to the
description of a use case, even though not explicitly stated.

Semantics of use case concepts is not the only problem we run into. Another prob-
lem is that there is not that much information available about what use cases used
in projects are like. One of the few articles that we found, reports with examples
how use cases and scenarios were developed in a major software development project
[MaR05]. Generally, the surveys on the industry requirements engineering practices
that we have come across, merely state that use cases are a popular technique. For
example, according to one survey over 50 percent of the companies used use cases
in the requirements phase [NeL03]. However, the companies that had participated
in the surveys had not specified what their definition of a use case was. In the same
survey, 51 percent reported use of informal representation. Furthermore, when an-
other survey reported of the rarity of a standard application of various requirements
specification practises, and only about one third of the companies used natural lan-
guage in a company wide standard manner [NSK00], we draw the conclusion that
indeed there is no general consensus as of how to describe use cases.

We found support for this assumption as we recently had an opportunity to look at
some requirements specification documents in a large information technology com-
pany's database. The company had acknowledged the importance of requirements
engineering practises, and apparently had encouraged their employees to apply use
cases in the requirements specification documentation. The variability in the scope
and structure of the documentation was however huge. If use case templates were
available, they were not used consistently. The quality of the documentation seemed
to be directly dependent on the skills and enthusiasm of the author. Even though
the number of viewed documents was too small to give any conclusive evidence of
the company's prevailing requirements practices, we feel this experience supported
the common view that there is still a lot of room for improvement both in terms of
developing customised use case templates and of improving the skills of the people
using them.

We believe that one reason for the lack of industry reports on success of use cases
in development process is due to the fact that especially larger companies take ad-

vantage of some commercial use case driven development method, which is either
licensed or purchased from the developer as part of a larger development and service
contract. The companies owning rights to the method may not wish to publish their
models in too much detail for obvious reasons. It is even less likely that any such
company would wish to publish cases that have failed. The authors that do publish
their experiences, are in our view active members of the requirements engineering
research community who often work in projects through their own company as in-
dependent consultants. They are in a position where they want to contribute to the
development of requirements engineering research, and are also free to publish their
findings.

Our conceptions of use cases are therefore based on articles presenting work of
prominent researchers and industry practitioners. The authors’ recommendations
of how and when use cases should be used are mainly based on their experiences
in software engineering projects, but also on their observations on the requirements
engineering practices in various companies. Considering the scarcity of empirical
research in use cases, we feel that following these guidelines is the best option.

The definitions and thoughts presented in this section are mainly reflections on the
UML 2.0 standard [Omg06b] and Alistair Cockburn’s work on use cases [Coc97,
CoF98, Coc01, Coc02, Coc06]. We have chosen to base our thinking on his use case
description, as we have found that his work has been widely accepted and referred
to in the literature, for example in [CoL00, KPR03, SeS00, FoS01, RoB08]. His work
on the other hand follows closely Jacobson’s initial work.

In the following subsections we will look into the UML use case diagram and our
views on describing use cases. Before that we elaborate on the key use case elements
actors and scenarios, as well as define the concept of goal, which in our view is an
important ingredient of a use case description.

**Actor.** According to the UML 2.0 standard an actor *specifies a role played by a
user or any other system that interacts with the subject, the subject denoting
the system under construction, or a class or a component [Omg06b]. In this
context we take the 'black-box' view and consider the subject as the system.
All action in the system that is triggered from outside the system should be
described as an actor activity [PiP05].

In Jacobson’s initial definition of a use case, an actor referred to a human
individual or groups of users of the system [Jac92]. Today the concept of an
actor expands from humans to external hardware and software systems. The
actors can also be grouped as primary or secondary actors depending on their
relationship to the system. Primary actors are the system users who require
a service from the system. Secondary actors are the actors who provide the
system with some service [Coc97]. Since a use case diagram can also be used
in determining the scope and assignment of responsibilities when designing a
system, subsystem, or component [PiP05], the actors can also be grouped as
internal and external actors. In this context where we only look at the external
behaviour of the system, we are only interested in external actors.

Actors are considered to execute the use cases. However, there are differences
in what people show as actors in their diagrams. Some people like to show
every external system or human actor. Some only show the initiator of the use
case [FoS01], and some both the primary and the secondary actors [KSW01].
We feel that focusing only on the primary actors keeps the use case diagrams
much more simple. It is then clear that the focus is on services that the new
system is about to provide the users with.

Scenario. A scenario is a description of a sequence of actions that can take place
while the user interacts with the system. The scenarios that share a common
goal are the contents of a use case [FoS01]. A scenario describes the system’s
responses to the actor’s inputs. The scenarios are started by a triggering event
[Coc97]. The main scenario describes the intended successful course of actions,
which is completed when the actor’s goal is reached. The alternative scenarios
describe routes that deviate from the main scenario. These alternative routes
can lead to error states of the system, and hence failure in reaching the actor’s
goal, or offer an alternative way for reaching the goal.

The basic idea of scenarios is that they help the requirements engineer in con-
sidering the range of possible courses of events that can take place within a
use case. The scenario is the core of the use case, and the advantage of using
scenarios is that they help to deal with what can vary in the course of actions directed towards reaching the goal [Coc97]. Additionally, scenario-based thinking can help the requirements engineer to expand his or her imagination to be more creative about how to solve the customer's problem. Perhaps looking at the solution from a wider perspective can improve usability or even performance. Scenarios are widely used in the requirements engineering, but there is not much information available about specifying scenarios [MaR05]. Therefore, we find it appropriate to look for analogies in the field of business strategy for further justification and discussion of why scenario-based requirements capturing is beneficial.

In business management scenario planning has been a tool for deriving business strategies since the early 1970's [Sch95]. In the economics and the business strategy the scenarios capture possible courses on a macrolevel or company level, but some of the basic guidelines could well be adopted for drafting the initial setting for use case scenarios. A process for developing scenarios should include defining the scope [Sch95]. In the scenarios related to use cases the scope would be the goal of the use case, for example 'withdraw money', 'buy goods'. Furthermore, the scenarios provide context and boundaries for the requirements elicitation, analysis and specification discussions, so that people do not stray away from the area of interest [Coc06].

Another important step in outlining a scenario is the stakeholder identification [Sch95]. The stakeholders help to define the breadth of the system [CoF98]. The stakeholders in the use case context are the primary and secondary actors. For requirements prioritisation purposes, also other stakeholders should be identified, such as the decision makers who have particular interest in the goals related to the use cases. One should also identify the basic trends and the rules of interaction [Sch95]. In software development, this could mean for example identifying how people want to use their phones, music players, or desktop applications. In some other application areas, it could mean preparing for the future change in the legislation that will enforce new ways of using the system. In practise, these steps are interleaved in the requirements elicitation and analysis phases. Both the actors and their goals need to be defined, but which comes first is not that clear. One must acquire information about both to be able to map future users or their job descriptions to goals [CoF98].

**Goal** Goals are objectives and targets that the system should achieve [Lam01,
ACD01], or the desired end-state of the system [CoL00]. If the software helps
the user to accomplish his or her goals, it creates the greatest business value
[Coc02], assuming that the system level goals have been correctly derived from
the business goals. The abstraction level of a goal may span from high-level
strategic goals to low-level system goals.

The use of goals in requirements engineering provides benefits for both the cus-
tomer and the developer. The advantages extend from improving the commun-
ication and requirements identification to providing a criterion to complete-
ness of the requirements specification [Lam01]. Communication is improved
when the terminology used in requirements engineering is derived from the real
world [ZaJ97]. Goals are derived from the customer’s business environment,
and hence expressed in concepts that are familiar to the stakeholders. An em-
pirical study in the requirements engineering process improvement has given
evidence that the definition of business goals motivates the business manage-
ment and enhances their participation in the project [SoR05]. Approaching
problems from the business perspective shows the relevance in a concrete way,
and people become more engaged compared to when presenting things solely
from the technical perspective.

Furthermore, the goals provide reasoning why we want to do something [ACD01],
keeping things understandable for the less technical stakeholders. Providing
the rationale for the requirements goals is also a good starting point for actor
identification. As the development goes further, the goals provide a criterion
for evaluating completeness of the requirements specification. In the end of
the development, the implementation is validated against the specification,
and the final acceptance criteria could be based on identifying whether the
goals have been reached.

However, there are problems related to the use of goals. First of all, the
concept of goal is fuzzy. Furthermore, it is not always easy to identify goals.
On organisational level goal derivation requires expertise in business analysis
and strategy. The goals derived from the business environment may end up
not reflecting the actual situation, but instead represent an idealistic view.
False goal recognition may lead to ineffective requirements. Additionally, it
is not clear how goals could be formally tracked, or associated to scenarios
[RSB98].
3.2 Use case pitfalls

Use cases are considered to be a simple technique which can be used in projects as the entire system requirements specification, as part of the system requirements, as an analysis technique to elicit user requirements, and as software subsystem-level requirements [Lil99]. However, the simplicity of use cases is somewhat misleading. Many of the use case related problems stem from the lack of skill and experience of the people applying the technique. When a task seems very simple, it is human to bungle, and that has been observed to happen in many cases when development teams have applied use cases. Typical problems associated with use cases are undefined or inconsistent system boundary, complexity of the use case model, and writing use cases that are hard to understand or never completed [Lil99].

Definition of the system boundary should get attention early in the requirements elicitation and analysis phases. The system boundary definition includes defining what one means by the 'system'. The boundary definition does not only mean being clear about the abstraction level, but also deciding who the system users are. For example, for a banking system the online customer is a user, but a client who is contacting customer service through a telephone is not. The use cases should be written from the user’s perspective, and they should be named from the perspective of the actor’s goal. Inability to analyse the domain and its business environment may lead to misinterpreted goals. Wrong goals, on the other, hand are likely to lead to system behaviour that may not meet the customer’s needs.

The ‘user’s perspective’ is pointed out in several occasions in the literature. The reason why one should describe the system functionality from the user’s perspective is that one does not want to get involved in internal system design. One wants to focus on what the system is expected to do for the user to reach his or her goal, not how the system works internally to provide the users with service. One should also be careful with the actors and their names. The actor represents a role that a user has when using the system. A role should occur only once and be presented by one actor only.

Complexity of the use case model implies to a number of problems in the use case specification or a description of a single use case. Pitfalls include creating too many use cases, not attaching the use cases to correct goals, and authoring use cases that are not consistent in contents and size. Additionally, there may be too many relationships between the actors and the use cases: An actor may have interaction with each use case, or a use case may relate all actors. For reduction of the complexity,
one should be clear on the actor goals and their definition, remove the use cases that
describe the system's internal functions and include only those that describe inter-
action between the user and the system. The size of the use cases can be limited by
splitting them or by using include and exclude relationships, and by checking that
the definition of an actor is appropriate.

*Use cases* can be *hard to understand* if they lack context. The use case descriptions
have also been observed to be hard to understand if they resemble the flow of a
computer program. If there is a need to describe complex branching and looping,
one should consider other techniques, such as decision trees and pseudo code. If the
use case descriptions are complex and inconsistent, they are difficult for the customer
to understand. This is a serious problem in the process, as it is the customer's duty
to validate the requirements. The requirements engineers are however responsible
for ensuring that the customers are trained to understand the technique used, and
that the customers are involved in the requirements process from the elicitation to
the specification of the user requirements, or even system requirements.

*Use cases might not get completed* for two reasons. First, the author of the use case
does not know when to stop authoring a use case description, and second, use case
descriptions need constant changes because they are linked to system design or user
interface design. The notion of a goal is a good delimiter for determining when one
has completed authoring a use case. Once one has described the actions leading to
completion or failure of the actor's goals, the use case description is complete. If
use cases do not only describe key interaction, but are coupled to internal design
or elements in the user interface, every time the user interface or system design
changes, one must also change the use case. This not only adds to the workload
of the requirements engineers, but also reduces the reusability of the requirements
specification in other projects or in potential future re-design of the user interface.
While attending a software development project module we personally experienced
that one gets easily carried away in describing requirements by linking them to
the elements of the user interface design. We believe linking requirements to the
user interface is a typical error beginners make while taking their first steps in
describing requirements. One should however be aware that the use cases may not
get completed and requirements frozen unless one manages to avoid the linkage.

*Localisation* and *information hiding* are aspects one needs to be aware of when using
use cases for system requirements specification [Ber95]. These aspects relate to
using system requirements as support for recognising potential object classes, and
for establishing the interaction and relation between the classes in object-oriented system development. *Localisation* means that an item’s location is limited to a particular area, suggesting that object classes could be found within the boundaries of individual use cases. As use cases are generally functional in nature, developing object classes directly from the use case imposes a threat of functional decomposition. Additionally, as objects are not functions, and functions are not objects [Ber95], use cases do not automatically provide the object classes for the internal designer. Deriving object classes directly from the use cases may lead to overlapping class structure either in design or implementation.

*Information hiding* refers to the concept of encapsulation where certain elements of class information are made inaccessible to other classes. The problem with use cases arises if the information hiding aspect is omitted, and the author of a use case does not stay within the black-box view, but gets involved in designing the internal structure. In addition to the fact that the design decisions are likely to change over time, there is a possibility that use cases get overloaded with details, which complicates the requirements management.

According to an optimistic view use cases are all one needs for specifying the requirements. It would be wonderful if they were, but unfortunately there appears to be no one single technique that is enough for describing all qualities related to functionality, behaviour, structure, dynamics, and control [Got02a]. Using several views allows presenting the context in a much more versatile way. The categories for modelling organisation, data, behaviour, domain, and non-functional requirements presented in Subsection 2.3.4 could for example be used as a guideline for determining the types of information that need to be described or modelled, and choosing the most suitable technique accordingly. The different models and use case descriptions should though be somehow linked. For example the business rules should be presented separately, and then referenced by name and location of the description when used in a use case description. Likewise the inputs and outputs, and their value ranges and types could be described somewhere else, and then referenced in the use case description [Coc98b].

The choice of modelling or description technique should be based on the needs of the application area. Use cases are a suitable requirements specification technique for highly interactive systems in business domains like operations, administration, billing, and ordering. On the other hand, for business domains, such as data query and analysis, where structure and attributes of the data play a significant role, or for
business domain such as logistics, contract negotiation, where dynamics, including the allowable states of the process need to be specified, one should consider other techniques [Got02a].

3.3 Use case structure

The core of a use case description in principal is about answering to simple questions who, what, when, and how, and being able to communicate them to the relevant stakeholders. The who refers to one or more actors that interact with the system. The what describes the actor’s goal, and how its accomplishment will be determined. The when refers to the timing; what are the prerequisites for the use case to take place, and how one knows when the use case has been completed. Finally, the how describes the steps of a scenario that need to be taken to accomplish the goal.

At the first stage of constructing use cases, the actors are very important. The actors define the scope of the system, and are the source for the goals that need to be accommodated. Once the goals have been identified, one needs to map them to the business rules and the people performing the related operative tasks [CoF98]. How one chooses to describe the scenarios and other related information is up to the project’s needs, the skills and preferences of the involved people as well as the organisation’s decision.

Revisiting the initial problem setting we have two aspects to the way the requirements should be described. On the one side are the potentially non-technical customers who need to be involved in the process, as they are the owners of invaluable domain expertise. Furthermore, ultimately they are the most important authority in deciding whether the new system meets the goals. On the other side are the developers who need complete and accurate information of the requirements. As the developing organisation also wants to be profitable, they want to use methods that allow them to re-use and automate as much of the work as possible. In most cases formal expression will be too complicated for the customer side, and not necessarily mastered by all of the developers either. A totally free form on the other hand is not optimal either. Especially in large projects there is a need for formats that can be easily managed. This is a quality not supported by completely informal natural language narratives.
3.3.1 UML use case diagram

The use case diagram presents a general view of the relationships between the use cases. The use case diagram also gives an overview to the extent of the system, and could potentially be used for estimating the amount of work needed for the functional requirements specification.

The UML use case diagram notation is fully available in the UML standard [Omg06b]. We present here the core elements only to the extent that is needed for understanding some issues related to constructing use case descriptions. The core elements, as presented in Figure 3, are the actors that interact with the system, a list of use cases connected to the relevant actors, and the relationships between the use cases.

![UML Use Case Diagram](image)

Figure 3: UML use case diagram

The standard defines the scope of the use case diagram of the system, but allows the notation to be used for a subject which is defined as the system under construction to which the use cases apply. The actors are entities outside the subject [Omg06b]. As such, one could consider the use case diagram as a kind of a table of contents of the use case descriptions in the specification.

There are four kinds of use cases: The concrete use cases which can be instantiated, the generalisation use cases which support reusing use cases, the extension use cases, which add behaviour to the concrete use cases without changing them, and the inclusion use cases which add behaviour to use cases by changing them [Coc02]. The UML use case diagram offers three types of relationships, extend, include, and generalisation, for presenting the relationships between use cases in the use case
diagram. These relationships denote the one-to-one or one-to-many connections that use cases can have.

«extends»

The extending use case is a use case that adds behaviour, possibly conditionally, to another use case, the extended use case at an extension point [Omg06b]. The extend relationship can be used when describing a variation on normal behaviour. The varying behaviour could be described in a scenario, but describing the varying behaviour as a separate extending use case enables its use in other base use cases. The fork to the extending use cases is signalled with an extension point.

The extending use case may add additional behaviour only at extension points. A use case may have many extension points, and an extending use case may extend one or more of these extension points. The extension points can be indicated on the line between the use cases on the diagram [FoS01]. Additional functionality is extended to a base use case if specified conditions are met [PiP05].

In the UML standard a dashed arrow with an open arrowhead shows an extend relationship. The arrow is directed from the extending use case to the extended one. In the example in Figure 4 the use case Open Account is extended with the extending use case Add joint member. The actual use case description of this extending use case would be a scenario where in the conjunction of opening a bank account there is an option for adding more members to the account. Where this adding joint members would actually take place in the base use case would be marked with extension points in the extended use case.

«includes»

In many applications there are functionalities that are repeated in more than one use case. The repeatable behaviour can be described in a separate use case which can be referred to in other use cases. An include relationship in the use case diagram defines that part of the use case’s behaviour is actually defined in another use case. Include relationship helps to avoid repetition in the documentation. One does not need to copy the same behaviour to several places. It also facilitates future document updates.

In the UML standard a dashed arrow with an open arrowhead shows an include relationship. The arrow is directed from the including use case to the use case that defines the referred behaviour. In the example in Figure 5 there are two example use cases from an online shopping service. Two important use cases Order item and Track packages require that the user must be logged in. As there is no point
of repeating the logging-in process in the two use cases, an include relationship is made from them to a separate Login use case. Again this facilitates producing and maintaining the documentation. It also helps to keep the use cases focused.

Figure 5: «include» relationship [PiP05].

«generalisation»
The generalisation relationship is also applicable in other UML modelling concepts. In the context of use cases generalisation can be used when one wants to express some type of system behaviour without going into detail [Omg06b]. For example, in Figure 6 the Verify identity use case denotes the general high level case. The Check fingerprints and the Check RFID use cases denote two specialised use cases. The generalisation relationship is presented as a directed solid line with an empty arrowhead pointed from the specialised use case to the generalisation. The
generalisation could also be applied to the actors. For example the general case of an actor could be a Staff Member with the specialised cases of a Manager and a Team Member.

The difference between the generalisation and the extend relationships is somewhat difficult to grasp. Just as the extend relationship, generalisation adds behaviour to the base use case. The difference is that in the generalisation it is a type of inheritance where the attributes of the general base case apply also in the supplementing specialised use case. Generalisation is basically another way of capturing alternative scenarios [FoS01]. One should be aware though that the specialisation should still discuss functionality, not implementation [PiP05].

![Diagram](image)

Figure 6: «Generalisation» relationship [PiP05].

The use cases and the diagram listing them will also evolve as the requirements elicitation and analysis progress. The use cases that start as skeletons, get more meat around their bones, and maybe get divided as the process goes on. As the work proceeds the requirements engineers may choose to use relationships. The great benefit that the relationships offer is that generalisation and extend allow a use case to be split up, and include helps to avoid duplicate work. All relationships support reuse of the specification. Furthermore, splitting a use case makes managing it less complicated [FoS01].

### 3.3.2 Dimensions of a use case description

In the absence of one single use case description template that is applicable throughout the requirements engineering process, development teams should be prepared to use different templates in different requirements engineering process phases [Coc97].
Each template should convey the essential information on that particular level of abstraction [Li99], and be understandable to the relevant stakeholders.

Table 2 lists possible techniques that can be used to describe or model use cases. The techniques include interaction diagrams, activity diagrams, state charts, and textual descriptions [KSW01]. All techniques have advantages and disadvantages when used for use case description. Activity and state diagrams show the flow of events well. Their downside is that they cannot capture associations between actors or use case relationships. The interaction diagram’s downside is that it cannot express all scenarios of a use case, but only one scenario per diagram. Textual descriptions are easy to produce, but their downside is that in practice they lack structure, exacerbating requirements management.

None of these techniques is generally applicable in all cases. Rather they could be used to support each other. We however believe that use of natural language is necessary at least when use cases are first developed. At least for a layperson, it is easier to conceive the idea of the use case when expressed in natural language. Later on in the requirements engineering process one should move towards more structured and formal expression.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interaction diagram</td>
<td>Can only express one scenario, not the whole use case</td>
</tr>
<tr>
<td>Activity diagram</td>
<td>Cannot capture association between actors or use case relationships</td>
</tr>
<tr>
<td>State chart</td>
<td>Cannot capture association between actors or use case relationships</td>
</tr>
<tr>
<td>Textual description</td>
<td>Often lack structure to facilitate management and automation</td>
</tr>
</tbody>
</table>

Table 2: Techniques that can be used to describe use cases [KSW01].

Since the main source for the requirements is the environment where the new system will be located, the form that use cases take in the documentation must convey facts and needs representing that environment. In projects use cases have been observed to vary by purpose, contents, structure or form [Coc97, WPJ09], plurality [Coc97], and life-cycle [WPJ09].

*Purpose* refers to what one wants to achieve by using a use case. Use cases could be utilised merely for elicitation and analysis to gather user stories, or they could
be used to build and specify functional requirements. We have not found enough proper examples of use cases to be fully convinced of their suitability for system requirements specification, but at least for user requirements specification they appear to work well.

*Content* refers to the substance of a use case and the way it is presented. Content includes ensuring that use cases are authored in a consistent way and they deal with the same level of abstraction. The content of a use case also refers to whether one describes a normal or an exceptional case of behaviour.

*Structure* of a use case can be unstructured, semi-formal, or formal. Unstructured, or informal use case, is composed with natural language, has no internal structure, and lacks mathematical basis. Additionally, use cases that lack structure can be ambiguous and prone to errors of logic [CPS01]. Formal description methods are based on mathematical notation and languages, and allow precise expression. There are numerous variations available of formal methods, but their downside is that they are difficult for a layperson to understand. Use of a semi-formal language in a semi-formal structure is the third alternative. Semi-formal structure leaves space for people to express themselves with natural language, but provides structure to enable utilisation of some requirements management tools [Coc02]. The form of a use case refers to the type of presentation format. The form can be either static, animated, or interactive [WPJ09].

*Plurality* refers to the number of scenarios within a use case. A use case could be just another name for a single scenario, or there could be multiple scenarios within a use case.

*Life-cycle* represents the use case’s existence and development over time. Life-cycle includes aspects like technical handling, evolution, traceability issues, and use of a use case as basis for test cases. Our opinion is that in professional software development, it is of utmost importance to consider the life-cycle of requirements, and be prepared to author use cases that can be re-used in other projects or later on in the project. The more need there is for reuse, the more rigorously one should pay attention to the preciseness of the details and form, as well as the traceability and management of the use cases. A typical problem in requirements engineering however is that the requirements keep changing even towards the end of the development process. This imposes a challenge to requirements management. Updating requirements and related models and diagrams is laborious as the requirements evolve and change. One needs a really good management tool to keep track of the effects of
change in a use case on other use cases, and in the documentation created for other abstraction levels. We believe that linking the requirements, and in this case the use case, to business goals may help managing them.

3.3.3 Use case building blocks

We start constructing a use case by using the use case dimensions as our guideline. We first consider the purpose of the use case. In a real project we first develop understanding of the environment and the stakeholders by using techniques suitable for modelling context. Once we have developed basic understanding of the problem we would start using use cases and scenarios. We work together with the relevant stakeholders to develop the first initial short descriptions to elicit more requirements and keep the discussions focused. As the process goes on we use use cases to facilitate identification of further requirements, and elaborate use cases to a full specification with needed attachments. This approach is described in [MaR05], and in the authors’ experience the approach enables full participation of the non-technical domain experts in the project. In our opinion the greatest benefit of this approach is that the requirements and scenarios get continuously validated both by the customer and the developer. Furthermore, continuous dialogue promotes mutual understanding of the problem domain and the requirements. In this thesis we however omit the process of actually creating scenarios and focus on the elements that a use case description used for documenting user requirements should contain.

The use case description we present here is adapted from Cockburn’s use case template and samplers [Coc98a, Coc98b]. The template is not his latest one, nor are we claiming that it is the best template ever published. However, in our view it presents important aspects consistently, and aids in considering the use case contents from a rather wide perspective. The use case dimensions we apply in the use case description template are listed in Table 3.

We look at the use case template from the perspective of using it for the purpose of specifying user requirements. The content of the use case includes describing both main scenarios and scenarios of alternative flows of events. The structure of the use case will be semi-structured, contents presented in consistent prose in a static form. An alternative to consistent prose would be to utilise some formal grammar to define the semantic structure and the grammatical functions of the language [RoB98]. Use of grammars no doubt enhances the coherence of use cases and allows use of automated checks of consistency. We however believe that without
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Specify user requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>Consistent prose</td>
</tr>
<tr>
<td>Structure, Form</td>
<td>Semi-formal, static</td>
</tr>
<tr>
<td>Plurality</td>
<td>Multiple scenarios</td>
</tr>
<tr>
<td>Life-cycle</td>
<td>First version</td>
</tr>
</tbody>
</table>

Table 3: Our choice of use case dimensions for the use case description template.

an authoring tool that guides the author along the way, use of grammar is not that easy to set up in an organisation. Additionally there the training and setup overhead may be turn out to be high.

As for the plurality of the use case template, we assume that a use case consists of multiple scenarios. The use case scenarios consist of the main scenario and possible alternative scenarios. The description follows a predetermined semi-formal structure which is the same for all use cases. The possible values of the life-cycle dimension are not clearly defined in [WPJ09]. We however take the approach that this is the first version of the full use case description.

We also assume here that the actors, the system boundary, and the goals have been identified and determined in the interleaving elicitation and analysis phases. We also assume that there are initial scenarios of usage situations available, and focus only on what a use case specification could be like. We only present a 'normal' use case that describes what a system should do. Misuse cases are an interesting addition to use cases, that could be worth considering. Misuse cases are use cases that describe behaviour that is a threat to normal use cases [PaX05]. Earlier it has been advised that use cases should only describe what the system will do, not what the system will not do [RoB98]. Nowadays misuse cases have however become a topic for research especially in safety-critical application areas [PaX05], and could in our opinion well be incorporated in the use case specification.

The use case description consists all together of six sections, divided into three main sections and three optional sections. Table 4 presents the three main sections and their respective elements: Characteristics information, main success scenario, and alternative scenarios. The three optional sections, related information, open issues, and general issues, and their respective elements are presented in Table 5.

Main sections of a use case description template
The characteristics information consists of elements that support identification of
the use case, portray the context and level of abstraction, and the conditions needed for the events of the use case to take place. The identification number of the use case facilitates use case traceability. In our opinion the identification number should not be based on the structure of the document, such as subsection numbering, but be an individual reference code that stays the same even when the document is changed. Descriptive use case names enhance readability and support browsing the document, and it is recommendable to use short active verb phrases that crystallise the described action [RoB98].

The goal in context -element provides a place for a longer statement of the actor's goal. The scope refers to the abstraction level of the use case; the extent of the system being considered as the black-box under design. The scope can vary from a high level organisational view to the system’s role in its environment to a low level view presenting the interaction between the user and the system. Level refers to the scope of the use case description, whether it is providing a higher level summary view to behaviour, or describing a primary task of a use case or its subfunction. The precondition of a use case is a constraint that must be true before the events described in a scenario of a use case can be instantiated [Win05]. The precondition refers to what we expect the system state to be before the events in a use case can take place. Alternatively in a use case on a higher abstraction level, the precondition can refer to readiness of the user to engage himself or herself with the system to start interaction. The postcondition is a constraint that must be true when a use case has ended [Win05]. The postcondition defines the expected system state after the use case has been successfully completed, or after the goal has been abandoned [Coc97]. On a higher abstraction level, the postcondition defines the state of affairs in the problem domain or business environment after the completion of the use case. The primary actor refers to the role of the user who has the goal and triggers the use case. The trigger is the event that starts the use case. In interactive systems, the triggering event is something that the actor does via the user interface to launch the events of the scenario.

The use case description should cover all steps from the triggering event to the satisfaction or failure of the goal. The basic flow of the main success scenario describes the most common and non-erroneous way for the actor to reach his or her goal. The alternative flow section contains all scenarios that describe deviances from the main scenario, including alternative flows of events and error scenarios [Win05]. Cockburn names this section as extensions in [Coc98a], but in our opinion that name is somewhat confusing, as it is not clear whether it refers to the UML


extends-relationship or the alternative scenario.

All scenarios should be written in customer’s terminology from the user’s perspective. Direct word order is recommendable. UML relationships includes and extends will be used for some use cases, and the extension points should be added where applicable. One should also find a way, for example by use of hyperlinks, to refer to subfunctions described in other use cases and link the nouns representing system inputs and outputs to the glossary, and to the data model. Without linkage it is difficult to know which other documents or use cases need to be updated when a use case changes.

<table>
<thead>
<tr>
<th>USE CASE DESCRIPTION - main sections</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Characteristics information</strong></td>
</tr>
<tr>
<td>Use case id and name</td>
</tr>
<tr>
<td>Goal in context</td>
</tr>
<tr>
<td>Scope</td>
</tr>
<tr>
<td>Level</td>
</tr>
<tr>
<td>Precondition</td>
</tr>
<tr>
<td>Postcondition</td>
</tr>
<tr>
<td>Primary actor</td>
</tr>
<tr>
<td>Trigger</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>2. Main success scenario</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic flow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>3. Alternative scenarios</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative flow</td>
</tr>
</tbody>
</table>

Table 4: Main sections of a use case description. Adapted from [Coc98a].

Optional sections of a use case description template

The related information section covers a collection of elements spanning from the importance of the use case to its interconnectedness to other use cases on abstraction levels one above and one below. The priority of the use case refers to the criticality of the use case to the system. The more essential the described behaviour is to
the system, the higher priority should be granted to the use case. The performance target refers to the quality aspect of a use case quantifying the amount of the time the completion of the goal should take. Also other related non-functional requirements could be attached to the use case in a similar way. The frequency element expands the view of the importance of the use case over to considering how often the use case is expected to occur. In the original template the related information section also includes notions of superordinate and subordinate use case. The superordinate use case is a use case one abstraction level higher which includes the use case in question. The subordinate use case refers to a use case that describes some sub-behaviour related to this use case. We are not fully aware of how Cockburn intends these elements to be applied in practice, but we see them as an opportunity to link the use case to the abstraction levels one above and one below. In the secondary actor element one should list other systems that are needed to accomplish the use case. The modelling and definition of secondary actors should not be included in the use case description.

The open issues section presented in Table 5 provides a slot in the use case description for the author to make note of the questions yet to be answered, and decisions about the use case yet to be made. We find it advantageous that there is a designated space for the author to express potential concerns about the use case. It not
only helps other readers to notice what the remaining open issues of a seemingly completed use case are, but also document the open issues within the context where they belong.

The final section, the general issues section assembles a mixed collection of information related to the use case. Both the content and name of the section diverge from Cockburn’s template, where the section is named 'schedule' [Coc98a]. In Cockburn’s template the schedule element relates to due dates or releases of the deployment, staffing issues, and so on. In our opinion there are however elements that are not included in any of the sections, such as the version history containing information about the use case authors and update dates, and maybe even a short description of the changes made. In addition one should keep track of the related attachments and the use cases that are linked to the use case, and make a note whether they have been updated as well. One option would be to keep the schedule element as it is, and add another optional section. We however feel that in documentation schedule could well be bundled together with version history and other possible issues related to the use case.

The use case description templates have many advantages that support their use in the requirements engineering. First of all, templates promote systematic elicitation of the information needed in the use cases. The sections and elements in the template point out explicitly what information is wanted in the requirements specification. Use of the use case templates also provides support for authoring coherent and homogeneous use case scenarios.

It is important to deploy a requirements management system that supports traceability of the use cases. An optimal system includes a help function which provides assistance for novice use case authors for example by presenting typical scenario patterns for different types of interaction between the actors and the system.

When designing use case templates one must consider the specific needs of the application area, and adapt the template when needed. Additionally, distinct templates should be designed for different abstraction levels. One must also decide how the use cases on different abstraction levels are linked in the requirements management system. Furthermore, it is important to decide how the information in the use case descriptions is communicated to the stakeholders. A requirements management system that provides distinct views to the requirements, such as a business view or a user view, would be a definite asset for the requirements engineers in improving communication with different stakeholders.
4 Use cases in system testing

The outcome of the requirements engineering process is the requirements specification. The behaviours and quality aspects described in the requirements are transferred through the system design to the implemented system. If the requirements engineers have been successful in their work, the requirements specification describes the expected system behaviour, the related qualities, features, and the data in a such way that matches the problem domain and the needs of the customer.

The validation techniques that help to examine the match of the requirements to the factors in the problem domain should take place throughout the iterative and incremental requirements engineering process. The communication gap that may exist between the domain experts and the developers can in our opinion be diminished, if one uses the scenario-based use case description technique described in Section 3. Close cooperation between the domain experts and the developers promotes mutual understanding of the requirements. Additionally, the requirements get validated as they are elicited and analysed.

Let us assume that the requirements specification has been validated in a manner that is satisfactory both to the customer and the developers, and the customer has accepted it. Depending on the process model the customer now waits for the completion of system design and implementation. There is however another potential communication gap which can take place within the developing organisation: If system designers do not understand the requirements as the requirements engineers intended, the success of the development project may be compromised. Converting the requirements into elements in the system architecture is an unsolved problem in the requirements engineering, and until the system, or parts of it, has been completed, one cannot really validate that the implementation matches the specification. Well-authored use cases however provide a clear view to the behaviour of the system, and as such may help in bridging this gap. As far as we know, at the moment the only way to test exhaustively that use cases are realised in the implementation, is to run tests on a completed and integrated system.

4.1 System testing

The realisation of the requirements in the implementation can be confirmed in two ways, by static means or by validating the system by means of dynamic verification. The static means include code inspections and reviews. Dynamic verification means
testing the system by executing it. Generally testing refers to activity where errors and defects are found by executing the software, and then removed. Requirements-based testing’s purpose is however rather to validate that the system implements the requirements than to find defects [Som06].

The specification-based testing methodology for a completed system is called functional testing, or black-box testing [TVK01]. The testing phases that use a black-box view to examine the system are system testing and the post-development acceptance testing [Bin99]. The target of testing for both system and acceptance testing is the system in its entirety. System testing is performed by members of the development team. The purpose of the acceptance test is for the customer to determine whether to accept or reject the system [HKS97]. The acceptance test is eventually conducted by the customer, and it focuses on major functional requirements, interaction between the actors and the system, and the system’s external interfaces [HKS97]. The acceptance testing can be grouped into three test phases: alpha, beta, and acceptance tests [Bin99]. The alpha tests are performed by the development team, and their purpose is to evaluate the system’s overall readiness by simulating real-world usage. The purpose of beta testing is the same as the purpose of alpha testing, but the difference is that the customer and selected users participate in testing. Acceptance testing should take place at the customer’s final environment of use and be performed by the customer’s representatives.

The type of system testing we have in mind, is in fact a mix of system testing and acceptance testing. In our opinion use case based system testing has three goals. The first goal is to validate that the completed system complies with the requirements specification [HKS97]. The second goal is to simulate system use as it would be done by the customer, a goal of alpha testing [Bin99]. The third goal is to demonstrate that the contract between the customer and the developers, as described in the requirements specification, has been fully delivered by demonstrating that the user can reach his or her goal of using the system.

The requirements specification including the use case descriptions is where the correct behaviour of the system is determined, and incorrect behaviour can be identified [Whi00]. If use cases are used to describe the system’s functional requirements, including the expected external behaviour and interaction with actors, it is quite natural to use them as a basis for system test case design. The test case is a set of data which is used as an input when executing the system under testing. The test case is supplemented with information about the pre-test state of the system and
the environment, execution conditions, and expected result of a use case scenario [Bin99, Heu01].

The precondition for using use cases as a basis for test cases is that each use case has a full set of scenarios. The most important parts of a use case for generating test cases are the main and alternative scenarios describing the flow of events. The goal of the use case provides an observable measurement for determining whether the use case has provided the actor with the requested service or not. There must be at least one test case for each scenario including the conditions under which it will be executed. Equivalence classes for the input data values must be identified for each test case. Test cases should be so numerous that all scenarios with possible equivalence classes get tested [Heu01].

Software testing involves executing the software with test data. The tester simulates the interface that the system uses [Whi00]. In case of testing behaviour described in use cases, the tester simulates the role of a primary actor by inputting values to the system according to the predetermined list of inputs. The tester compares the outputs and behaviour of the system to the expected outputs and behaviour described in the use case descriptions. The actor's inputs and the system's expected responses are described in a test case.

4.2 Use cases and early test design

Use of use cases as the basis of test cases is not a new idea, but there are not many articles published that actually discuss the connection. Use cases as such are not test plans or test cases. The input values needed for test cases are not included in the use case descriptions. Use cases however describe the system behaviour and interaction with the actor under different conditions [Coc02], hence making a good basis for the testers to design and generate test cases. The advantage of use cases is that when used through the requirements engineering process, they capture key functionality early [Heu01]. In projects one should utilise a systematic approach to convert the use cases into test cases. One should also be prepared to alter the use case based test cases during the process, as the initial use cases scenarios may be outdated at the time of system testing [WPJ09].

There are arguments in support of early test design that suggest that test planning activities do not need to be put on hold until there is executable code available, but should take place as soon as the use cases have been captured [JBR99]. As
suggested in the adapted V-model of testing in Figure 7, the test design for system and acceptance testing could start as early as in the requirements analysis phase. Additionally, involving testers in the requirements engineering may mitigate defects and deficiencies in requirements [Gra02].

![Figure 7: The V-model for early test design [Gra02].](image)

Testers’ participation in the requirements analysis and specification may help to detect problems and inconsistencies in the requirements that otherwise might have been detected much later causing changes in the requirements. Additionally, testers have the expertise to point out if the requirements descriptions are not measurable, or do not support testability [Gra02]. A requirement is testable if *an objective and feasible test can be designed to determine whether the requirement is met by the software* [Bin94]. Furthermore, the testers learn directly from the customer what the system is expected to do. A thorough understanding of the system’s expected effects in its business environment helps to design better system and acceptance tests. The test cases are an instantiation of the requirements, and the requirements should provide enough detail to enable design of such test cases that portray a real usage situation. If the requirement description does not support testability, the tester can point that out. Furthermore, as specifying testable non-functional requirements is difficult, testers may be able to help by coming up with some measurable scale.
4.3 Basic elements of use case based testing

The preparation for system level testing consists of creating the test cases, designing the test procedures for the test cases, creating test scripts that implement the test procedures, and running the tests. The conditions which will be implemented during the test, and which are used to verify the success and acceptability of the implementation of the requirements are communicated in a test case [Heu01].

Important part of creating test cases is determining how the system's environment is represented in the tests [Whi00]. In case of testing behaviour described in use cases, key representatives of the environment are the primary actors. Thus, when executing the tests, the testers simulate the inputs given by the actors to the system. In case of use cases, the interest is often on actor's inputs via a user interface. Typical graphical user interface input mechanisms are mouse-clicks, keyboard events, and input from other devices. Additionally, all possible file formats, communication protocols, interfaces, and secondary actors should be considered as input sources. For test design the tester must examine all use cases and related requirements, and consider how their realisation can be tested in the implemented user interface. The testers should also understand how the users behave outside the system boundary.

Understanding business rules and user behaviour helps to outline the range of typical errors related to the system's functionality, and to understand what kind of activity can potentially crash the system. The tester must choose the input value ranges that will be tested, and decide upon the sequences and combinations of the inputs. The sequence generation problem relates to developing a model of the the physical inputs and abstract events, for example a graph or state diagram, or a regular expression. The model presents how the inputs and action symbols are combined to form syntactically correct test case execution paths [Whi00]. The inputs used in a test are selected by the tester. Even though the purpose of system testing is to check the match between the requirements specification and the system, by applying heuristics like equivalence partitioning and boundary value analysis in selecting the test input values, a tester can also catch errors in the system.

Another part of creating test cases is selecting the test scenarios. Alternative use case scenarios have been developed to describe expected behaviour also in conditional and error cases. In consequence, the use case scenarios provide a good source for test scenarios. In a test scenario one must take into account the possible source through a use case, including the branching structure marked with extension points. However, a disadvantage of use cases is that at least in the description format we
have presented, the dependencies between the use cases are not clear. One should clarify whether the use cases can be executed in a random order, or if the use cases must be executed in a specific sequential order.

The superordinate and subordinate use case elements in the use case description format target towards linking use cases. However, the linkage is vertical, rather than horizontal among the use cases in the same abstraction level. The dependencies between use cases would be easier to understand if one could somehow link the use cases. Use cases could for example be formed into a graph, where the nodes are the pre- and post-conditions, and the edges are the triggering events.

The next step after completion of test cases and test scenarios includes converting tests to code, and automating the application of inputs. The tests will be run according to the test plan's testing metrics. The test plan includes decisions of the number of inputs, the percentage of the use case scenarios to be executed, and the use case combinations one aims to cover in testing. Furthermore, in a test plan a tester should also decide how many times each use case will be triggered, and the number of times scenarios leading either to success or failure of the goal will be executed [Whi00]. The ideal coverage for testing is 100 percent, but it is unlikely possible or feasible to test all use case combinations. Therefore it would be important to know the use case frequencies, and the probabilities of use case combinations. For a totally new system this knowledge may be hard to acquire, but it is easier when there are existing systems and work processes at place. Additionally, one must ensure that the test cases are designed with intention and kind of inputs that are likely to produce errors and find faults in the operational logic of the system. Unless one systematically attempts to find errors also in the system’s external behaviour, the validation result is not exhaustive, and the match between the requirements and implementation is not validated [TVK01].

The tester's job is to adjudicate between what a system does and what it should do. The test oracle is the requirements specification. The outcomes of running the system should match the expected outcomes in the requirements specification [Gra02]. The difficulty in validating the requirements specification is that not all requirements are directly associated with a specific use case. Hence, it is difficult for the testers to trace which individual requirements are tested by specific test cases [TVK01]. For example it is difficult to come up with a straightforward test measuring 'usability’. Unless requirements are initially attached to some testable measure, it is not possible to validate exhaustively their realisation in the implementation.
Instead of requirements, a tester could use 'expert' users, the old system, user manuals, or the tester's own opinion as a test oracle. However, the specification represents the contract between the user and the system, and without the specification one cannot confirm that the implementation conforms to the contract. Additionally, ad hoc experimenting with a system is rather exploring than testing [Gra02]. Without a systematic approach to testing, one cannot establish convincingly that the goals of testing have been met.

4.4 Transforming use cases to test cases

The requirements specification provides guidelines on system design and implementation. Despite the requirements engineer’s efforts to describe the requirements in an unambiguous way, the designers and programmers may miss some of the requirements, or misinterpret them. Typical faults for use case implementations include wrong interpretations of the conditional behaviour, resulting in logic faults where the system behaves as if and logic was specified instead of or. Additionally the dependencies between two or more use cases may have been misconceived, or the dependency is missing in the implementation. Additionally, there may be generic system-scope faults in the use case implementations. The programmers may have implemented functions that were not specified, or the output may be incorrect [Bin99]. Use cases are a good source for deriving system test cases, because they describe the behaviour and services that the user expects from the system. Test cases derived from use cases can help in detecting if the implemented functionalities deviate from the functionalities agreed with the customer.

Use case descriptions provide the information needed regarding pre- and post-conditions of the use cases, and the paths through scenarios [Bin99]. However, use cases lack the information regarding the types and values of the operational variables. Operational variables can be identified from the input and output values visible in the user interface, constraints and relationships among variables, key characteristics of the user or the environment, and elements of the system state [Bin99]. Use cases may include extend or include relationships that must be considered when planning execution sequences for the test cases [BrL02]. A test model should include a complete inventory of operational variables, a complete specification of domain constraints for each operational variable, dependencies between use cases, and the relative frequency of each use case [Bin99].

The use case description model we presented in Section 3 does not include all in-
formation needed in test cases. Typically most use cases need to be reshaped or appended before they are suitable as test cases [Bin94]. There are two basic approaches, presented in Figure 8, for converting use cases into test cases, transforming the use cases and extending the use cases [RRW00]. Model transformation is based on the idea that the usage of the system is modelled in a different way in requirements engineering than it is modelled in verification and validation. In model transformation the information in a use case is transformed into another model, appended with test-specific information. Model extension is based on the assumption that the use case has been modelled or described in such a way which can be directly used in testing.

![Diagram](image)

**Figure 8:** Two ways of integrating usage modelling to testing [RRW00]

Many of the techniques for converting use cases into test cases take an approach where test cases are automatically derived from formally modelled use cases. However, initially specifications are never formal. Before the properties of a use case can be stated precisely and formally, they must be formulated in cooperation with the customer in natural language which all parties understand [Lam00]. Once the initial scenarios have been established it is possible to apply formal methods in modelling the expected system behaviour.

One formal approach to test case derivation is a method where use case modelling is combined with statistical usage testing [RRW00]. The approach is based on the idea that the use cases model usage of the system. However, use cases do not include information about the frequency of the use cases, and therefore statistical estimates of the frequency of usage are integrated with use cases.

Another technique that takes a formal approach to constructing use cases, is a technique where the use cases are initially developed with testability in mind by using abstract state machine language (ASML) for describing use cases [GLS01]. In this technique, tests are generated automatically with the help of a test generation algorithm that computes all paths to the states which can be reached using one
of the use case actions. Even though the tests can to some extent be generated automatically, in this method the testers must assign priorities to test sequences, and choose the actions that will be executed.

Natural language use case scenarios have also been formalised with statecharts [RyG99]. The statecharts are then annotated with pre- and post-conditions, data ranges and data values, and performance requirements. Test cases can be derived from the statecharts by traversing paths in the statecharts. Path traversal generates only tests for valid sequences of events. Therefore one must also include test sequences that result in a failure. There has been critique against use of statecharts, based on a view that statecharts are used to model state-dependent classes or small class clusters, and therefore, should not be applied to system testing [BrL02].

The techniques that convert use cases into test cases must be able to deal with the alternative scenarios in the use cases. One approach for deriving test cases from use cases that contain variabilities, expressed with UML use case relationships include and exclude, is a technique where the use cases are expressed as sequence diagrams. For example the include relationships can be handled by fragmenting the use case. If the test case derived from the use case is not affected by the include relationship, only one test case is derived. If the include relationship affects the course of the test case, the use case is split into two fragments, the base use case and the include use case. The two fragments will be concatenated in the test case [KPR03].

In the context of object-oriented development use cases are typically expressed with some UML diagram, such as sequence or collaboration diagrams [BrL02]. A natural corollary is to derive system test requirements from those models. Sequential dependencies between use cases are depicted in an activity diagram for each actor in the system. The presentation format helps the domain experts to identify and visualise the dependencies between the use cases. In such a diagram vertices are the use cases and edges are sequential dependencies between the use cases. The vertices are in fact extended use cases. The use cases are the source for parameters defining system behaviour, and extended use cases formalise the use case parameters by adding type and defining whether the parameter is an input or an output. Once the test case sequences that will be executed and tested are defined, the test oracles for each tested sequence must be derived. The main source for deriving a test oracle is the post-condition of operations in a sequence, which in the UML-based technique are defined using object constraint language (OCL).
4.5 Constructing extended use cases

Extended use cases are a test design pattern developed by Binder [Bin99]. Extended
use cases can be applied whenever a functionality has been described as a use case.
The technique facilitates mapping what needs to be tested, and deciding upon the
extent of the testing. The mapping is based on the use case’s constraints and
relationships [Bin99].

Our impression is that extended use case test design pattern is a widely accepted
technique. However, literature search linked us mainly to text book references.
Despite the lack of articles published on extended use cases, the number of references
to [Bin99] implies that extended use cases are considered a mature, and working
technique. In consequence, the content of this subsection is mainly derived from
[Bin99]. However, in our view the aspects that the extended use case test design
pattern covers, should always be addressed when deriving test cases from use cases.
Therefore, we see that in this context it is useful to cover the extended use cases,
even though presented from a unilateral perspective.

In the case of extended use case the word extended does not refer to a UML rela-
tionship extends, but to a concept of adding information on a use case description,
and transforming it into a form, that systematically aggregates the data needed in
test cases. The goal of the extended use case technique is to ensure that all funda-
mental use case relationships get exercised in testing. A use case describes the
inputs given to the system by the actor, and the system’s responses or changes of
state. An extended use case represents these relationships in a decision table. Addi-
tionally, possible faults in the implementation can be modelled as decision table
faults.

The first step of creating an extended use case is to identify all operational variables
for a use case. The use case’s operational variables include the explicit inputs and
outputs, environmental conditions which affect the actor behaviour, and abstractions
of the state of the system. The operational variables are used in creating a specific
test case.

The second step is to define the domains of operational variables. The domain
compasses all valid and invalid values for a variable. The domains are needed in
determining the inputs that will be used when executing the test cases. The valid
values for a variable are within the acceptable value range determined by the fac-
tors in the problem domain, and the invalid values for a variable are outside the
acceptable value range.

The third step is to model the relationships among the operational variables. The variables that cause similar system responses or changes in the system state, are grouped together into a decision table. Each row in a decision table is called a variant. Variants must be mutually exclusive: For a given input only one row at a time can be true. A variant can correspond to a scenario in a use case, but the testers can also add variants to the decision table, if they detect environmental factors that are not represented in the requirements specification. Table 6 presents an example of operational relations for a use case Log-in describing logging-in to an online banking service. In the example we use customer identification customer id as the operational variable input by the customer. In reality the customer id consists of a combination of multiple variables, such as customer number, password, key number provided by the bank, and the matching security code input by the customer. In this example, we have however bundled them together for simplicity.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Operational variables</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Customer id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Entered customer id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Customer bank response</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Customer account status</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
<td>Input correct customer id</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Valid</td>
<td>Contact your bank</td>
</tr>
<tr>
<td></td>
<td>Matches account</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bank acknowledges</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Closed</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Valid</td>
<td>Display main menu</td>
</tr>
<tr>
<td></td>
<td>Matches account</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bank acknowledges</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Open</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Operational relations for the use case Log-in (Adapted from [Bin99]).

The first row of Table 6 describes what happens when a customer attempts to log-in with an invalid customer id. The system rejects the customer’s attempt, and presents a message where the customer is advised to input a correct customer id. The second row indicates a situation where the customers id is valid, but the account is not valid. The system displays a message where the customer is advised to contact his or her bank. The third row describes also a situation where the customer’s identifier is valid. In this scenario the customer’s account is open, and the system establishes a session, and presents the main menu to the customer.
The fourth step is developing the test cases. Each variant will be designed at least one test case that makes the variant true, and at least one test case is designed that makes the variant false. A test case is true, if the set of values in a test case satisfies all conditions in a variant. A test case is false, if an input or a state value causes at least one condition in a variant to become false. If the operational variable used in a test case has a bounded domain, the domain of the variables is used to derive additional tests.

Intuitively development of such use cases that can be extended into test cases seems like the most advantageous approach. Adding detail into an existing structure should be an easy and straightforward task. However, the easiest way to communicate with the customer, is to use natural language. The use cases described with natural language lack the formalism needed in automating the generation of test cases. Additionally, the use cases tend to change during the development process, and there is a high cost related to maintaining two versions of the use case, the natural language and the formal version.

The extended use case technique is relatively straightforward method for transforming a use case description into a test case. The technique can be used independent of the use case description format. Our impression is that the use case scenarios depicted in a diagram can just as easily be converted into a decision table as a semi-structured scenario described in a natural language.

The disadvantage of the basic extended use case technique is that it does not contain the element of frequency. One should attempt to map the current usage patterns, and estimate the frequencies of different use cases. Identification of the most common use cases enhances prioritisation of the use cases and the test cases derived by using the extended use cases. Prioritisation supports the management of resources for example in a case of a delay in the project schedule.
5 Case: Dressage scorer system

Earlier in this thesis we stated that use cases are a convenient technique for specifying functional requirements. In this section we carry out an exercise where we apply the use case technique we have acquired from the requirements engineering literature. The application area is a scoring system for dressage competitions. The goal of this exercise is to get an idea of the feasibility of using use cases for describing requirements for an interactive system. Additional objective of this exercise is to investigate how well suited the use case description template is for deriving extended use cases.

The information used in the example is collected from the web sites of the Equestrian Federation of Finland (EFF) [Eff07] and the Fédération Equestre Internationale (FEI) [Fei06]. As an additional source, we use our own experiences in organising dressage competitions.

In this application there are many domain constraints that restrict introducing technically more developed approaches to scoring, and would require much more attention when designing a real system. In our opinion the two main constraints are the practically non-existent technical infrastructure in most dressage arenas, and the limited financial resources of organising clubs. Nonetheless, the problem domain is simple enough to provide a reasonable amount of information for demonstration purposes of this example.

In the following subsections we analyse the problem domain to identify the use case actors and their goals, and to determine the system boundary and the services the system must provide to the actors. We also demonstrate how a use case description template could be applied, and how it could be converted into an extended use case.

5.1 Problem domain analysis

5.1.1 Overview to dressage

Dressage is an equestrian discipline, one of the three competed in the Olympic Games. Dressage is often considered the art of equestrian sport, and its object is to develop the horse’s physique and ability in such a way that it seems to be performing the movements on its own accord, and responds to the rider’s commands immediately and even intuitively. The most advanced levels of dressage are competed in the Olympic Games and on international Grand Prix level, but competitions are
arranged to riders and horses with varying skills starting from the novice level.

In dressage competition the routine the horse and the rider perform is called a test. A test comprises of a series of movements, called figures, and is performed in a flat arena, measuring 60x20 or 40x20 metres. The arena is surrounded by a low rail along which 12 letters are placed symmetrically to provide reference marks for certain manoeuvres. Two to five judges assess the competitor’s performance, and award a mark from 1 to 10 for each of the figures and the transitions in the test. The judges also award penalty points if the competitor makes an error of the course during the test, such as omits a movement or takes a wrong turn. At the end of the test the rider and the horse leave the arena, and the judges award four collective marks for various aspects of the horse and rider’s overall performance throughout the test. Each collective mark is from 0 to 10 points. These marks produce a total score, which consists of the total points and the percentage score. The rider or team with the highest result is the winner [Fei06].

In this example we examine the problem domain from the perspective of a riding club organising regional dressage competitions. In the international Grand Prix or the Olympic Games similar tasks need to be performed to keep score of the competitors’ results, but the environment supports more advanced solutions.

5.1.2 Competition context

The organisation that promotes horse sports in Finland is EFF. EFF is one of the 126 sport organisations in Finland that are members of the Finnish Sports Federation (FSF). As presented in the organisational chart in Figure 9, the members of EFF are categorised as organisational members and actual members. The organisational members are typically riding schools, livery stables, and companies providing horse-related services or products. The actual members are individuals who engage themselves in horse sports professionally or as a hobby. The actual members are associated with EFF via membership to a riding club. The riding clubs and their individual members are of particular interest in this example, as they organise the regional dressage competitions. Internationally EFF liaises with FEI, the governing body and sole controlling authority for all international equestrian events. The rules and regulations of FEI are reflected also on the national level in Finland.

The role of EFF is to regulate and set rules for dressage competitions in Finland. EFF also designs the competition tests, authorises judges, and organises training in
various aspects of organising a competition. EFF also publishes the competition invi-
tations and the results on their monthly periodical and on their website. The tests are in line with the international rules and regulations for dressage. EFF provides the settings for organising competitions, but the practicalities of the actual event are the riding clubs’ and their members’ responsibility. The regional dressage competi-
tions are organised in a similar fashion with similar types of responsibilities for the organisers. The involved actors, EFF and their database system, the organisers representing the riding club, and the competitors in a competition are presented in the context diagram in Figure 10.

The competitor’s task on a competition day is to present on their turn the test required in the competition class. The test within a competition class is the same for each competitor, with the exception of Freestyle test where each competitor presents the required movements according to their own choreography. Any rider can participate in a regional competition provided that there is a suitable class, and that they have membership in a riding club which is a member of the EFF. The rider must also hold a licence to compete. In addition the horse must have a certificate of vaccination and an annual competition fee must have been paid. A rider can ride one or more horses in one or more classes. In order to participate and to become a competitor, the rider must enrol into the competition via EFF’s online enrolment service.

The judges’ role is to evaluate the competitors during their performance. The evaluation is both numeric and verbal. The secretary writes down the marks and the
verbal evaluation to the judges’ sheet at the judge’s dictation. At the end of the test the rider and the horse leave the arena, and the judges the collective marks. After the judges’ sheets have been completed by all judges, a messenger takes them to the scorers. The scorers first check all sheets for completeness, and then input the numeric marks into the scoring system. In the current operations model there are two scorers who perform the task together. One scorer reads the marks out loud one by one, and the other scorer uses a keyboard to feed the marks into the system. To avoid mistakes in the scoring the scorers double check that the numbers are correct by comparing the marks in the sheet to the input numbers appearing on the screen. Possible missing or unclear marks are checked with the judge at the earliest convenience, preferably between the next two competitors.

When the marks have been input, the system calculates the competitor’s total score. Once the results are ready, they are given to the announcer. A copy of the judges’ sheet is forwarded to EFF. The original sheet is given to the rider. At appropriate intervals the judges print out the results so far. The results are stored in the EFF’s database, and are available for the public via EFF’s web-site and periodical. Often in dressage arenas there is no online connection available. Therefore the scorers may not be able to transfer the results to EFF’s database until the end of the day from somewhere else, for example home.
5.1.3 System boundary definition

In system boundary definition three important aspects to consider are what one means by the 'system', who the system's users are, and what the goals of the system are. In this example we look at the system from two abstraction levels, competition and scorer system levels. The context diagram in Figure 10 presents the high level view of the actors involved in organising dressage competitions. The goal of scoring in a competition is very simple. The purpose of scoring is to compute the results and to rank the competitors to determine the winner and the placed riders for each class in the competition. The goal of the scoring system is to assist the scorers in this task. We will now elaborate on the knowledge we have so far in order to identify the scorers' responsibilities, and to define the boundary of the scorer system.

We first want to summarise the knowledge we have so far about the stakeholders and the system boundary. We do that by applying the stakeholder onion presented in Subsection 2.4. We first place the two governing organisations, FEI and FSF to the most outer circle of the applied stakeholder onion presented in Figure 11. Following the organisational structure, we set EFF on the next circle. In the same circle with EFF we place EFF's existing system. The existing system comprises of the enrolment system for the competitions, the result database, and the database for the competition classes. In the competition context this system is in fact a secondary actor. It is a source for the scoring templates, and also the destination of the computed results. It is clear already at this point that an interface will be needed between the existing EFF database and the scoring system.

In the same circle with EFF we also place a functional beneficiary to the system. This beneficiary is the EFF employee who is responsible for operating the interfacing system. All competition results maintained in the results database are input into the database either manually or via scoring system. Instead of the scorer inputting the results into the system on the competition day, the employee could just as well type the same information into the system at some later moment. In the same circle is another beneficiary group consisting of the competitors. In this case we expand the competitor group with the trainers, competitors' family members and friends - anyone who is interested in individual or overall competition results.

Moving closer towards the centre of the onion, we have the voluntary staff from the riding club organising the competition, and the hired judges qualified by EFF. The stakeholders that are involved in scoring are the judges, their secretaries, the messenger, and the scorers. They are all potential candidates as actors of the scorer
system. The judges are responsible for giving marks to the competitors for their performance in the competition. In principal the judges’ secretaries could input the scores directly into the system, but in practice the riding arenas where the local competitions take place do not have the facilities to support that. Also the performance of the test flows in such a quick pace that the secretary must be fully focused on the dictation of the judge. The messengers are not responsible for scoring because they cannot be delivering the judge’s sheets and computing the results at the same time. Thus the system’s primary actor is quite naturally a scorer. In principal any person participating in organising the competition could be a scorer, but in practice they are people who have been trained to use the system, and who know the scoring rules of a competition.

In addition to the scorer we have also identified other user groups illustrated in the onion, the help desk and the maintenance operator employed by EFF. However, they are typically not present in small competitions. Therefore there is a need for the system to be robust and operate so reliably the roles of the maintenance or help
desk are not needed on a competition day.

The scorer’s responsibilities prior, during and after the competition are depicted in the UML use case diagram presented in Figure 12. The responsibilities based on the prevailing scoring custom can be grouped into four use cases, to Download classes, Manage start list, Manage scores, and Communicate results. All but one of the four use cases include use cases related to more specific goals of using the system.

![UML use case diagram](image)

Figure 12: UML use case diagram presenting scorers’ use cases.

Downloading of the classes refers to ensuring that the scoring templates matching the selected competition classes are available at the time of the competition. Managing the start list includes the use case Organise start list, which describes the activity of organising the enrolled competitors into an order according to which they start their performance. The order may be drawn, or accommodated by the
organisers if some of the riders have special needs. Managing the start list also includes the use case Acknowledge changes, which refers to the scorers taking into account possible cancellations or changes in the participation. A rider may for example enter the competition with a different horse. During the competition day the starting order is communicated to the audience and staff in hand-outs and on notice-boards, so the start list must also be available in print, hence the use case Print. The use case Manage scores covers all activities that are directed towards the goal of the competition, ranking the riders per competition class. Managing the scores includes the use cases Input scores; inputting into the system the scores given by the judges, Compute total score; computing individual total points and percentages for each individual competitor, and Compute ranking; ranking the competitors. The use case Communicate results includes informing the competitors, spectators, and general public of the results of the competition classes. On a competition day the results are communicated in hand-outs and on notice boards, hence the use case Print.

5.2 Example use case description

The process of constructing use cases should be iterative and incremental. In a real project the use case drafts and related requirements should be adapted and iterated according to the increasing knowledge of the problem domain, and more importantly, according to the feedback received from the customer and domain experts. In this example we have no stakeholder feedback, and cannot present actual use case versions as they would be in different development phases. Instead we wish to demonstrate through two example use case descriptions how we would go about constructing a use case description.

In the previous subsections we have identified the key elements of a use case description, the actor and the high level system goal. In a dressage competition the scorers are responsible for carrying out the tasks that are needed to determine the ranking of the competitors. Thus we choose them as the main primary actor of the system. The people responsible for system maintenance and help functions are other actor candidates, but we ignore them in this example. The key goal of the system is to help the scorers in computing the results. This high level goal can be broken further down into subgoals that are met by the use cases listed in Figure 12.

In this example we apply the use case description template presented in Subsection 3.3.3. We take one of the scorers' duties depicted in the UML use case model in
Figure 12 under closer examination. We choose the key functionality of computing the individual competitor's total score. Computing the competitor's total score is a critical use case, as the winning order of the competition is based on that computation.

We start off constructing a use case description by structuring a higher level use case Manage score, which describes the current flow of the process of computing the competitor’s total score. The intent is to continue defining the system boundary and the system’s expected functionalities. The abstraction level of the use case describing the current flow of events presented in Tables 7 and 8 is high. The use case description so far only structures on a general level the current method of scoring as described in Subsection 5.1. We however feel that when there are established computation methods and practices in place, it is important to structure them in a way that provides a clear basis for analysing what elements of the current process cannot be changed and where there is room for design of new solutions.

<table>
<thead>
<tr>
<th>1. Characteristics information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case id and name</td>
<td>1 Compute total score</td>
</tr>
<tr>
<td>Goal in context</td>
<td>The judges mark the movements and the transitions in the test performed by the competitor. The scorer calculates the total score which comprises of the total points and the percentage, and is an average of each judge’s total points.</td>
</tr>
<tr>
<td>Scope</td>
<td>Competition</td>
</tr>
<tr>
<td>Level</td>
<td>Summary</td>
</tr>
<tr>
<td>Precondition</td>
<td>A judge's sheet and a template for scoring the class are available,</td>
</tr>
<tr>
<td>Postcondition</td>
<td>The total score for each individual competitor is computed</td>
</tr>
<tr>
<td>Primary actor</td>
<td>Scorer</td>
</tr>
<tr>
<td>Trigger</td>
<td>Receiving judge’s sheet</td>
</tr>
</tbody>
</table>

Table 7: Characteristics information for use case Compute total score.

Table 7 presents the characteristics information of our high level use case for Compute total score. In this example the use case is our first use case, thus the identification number of '1'. The name of the use case, Compute total score, is a short, active phrase, and states the goal of the use case. The goal of the use case reflects a high level user requirement of the system’s expected function of assisting the scorers in
computing the total score. The primary actor as identified during the stakeholder analysis, is a scorer.

The abstraction level is high, and the use case summarises a function on a competition level. The precondition for a computation to take place is that a marked judge’s sheet is available. The postcondition of the use case is that the total score for the competitor’s performance has been computed. The use case is triggered on the receipt of judge’s sheet.

2. Main success scenario

<table>
<thead>
<tr>
<th>Basic flow:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Scorer checks the judge’s sheet is satisfactorily completed. &lt;Variation: 1&gt;</td>
</tr>
<tr>
<td>2. Scorer acknowledges the rider’s marks from the judge’s sheet</td>
</tr>
<tr>
<td>3. Scorer calculates the total points by adding up the marks multiplied by the corresponding coefficients where applicable.</td>
</tr>
<tr>
<td>4. Scorer calculates the rider’s percentage by dividing the rider’s total points by maximum total points.</td>
</tr>
<tr>
<td>5. Scorer sets the intermediate individual classification of the rider based on the order of the percentages of riders performed so far.</td>
</tr>
</tbody>
</table>

3. Alternative scenarios

| 1. A score missing in the judge’s sheet |
| 1. Scorer returns the sheet to the secretary |

Table 8: Main scenario for use case **Compute total score**.

Table 8 presents the main scenario of the Compute total score use case. As the use case describes actor’s tasks on a high abstraction level, the scenario lacks descriptions of the system’s responses to the user’s actions. In addition to the main scenario, there is one alternative scenario, which will take place if the received judge’s sheet is not completed. We will however not elaborate on that, as the solution for an incomplete or erroneous judge’s sheet is always outside the system scope. Any unclarities related to the evaluation of the competitors must be personally consulted with the judge.

From the initial main scenario we can see that there are many details that must be clarified. Candidates for system requirements that add detail to the use case regard the numeric values of the marks, coefficients for collective points and penalty points. One also needs to clarify the computation equations for the total points and percentage.

The optional elements of the use case description Compute total score are pre-
sented in Table 9. We have set the priority of the use case as critical, as scoring is needed to find out the winners and placed riders of the day. Additionally, total score is an indication of the competitor’s personal success of the day’s performance. The goal of scoring is to have the competitor’s result ready preferably during the performance of the next rider or at the latest the one after. The reason is that the audience and the competitor are interested in hearing the points as soon as possible after the performance. The computation in this application is so simple that the performance target is easy to accomplish on a system level. The final order of the competition class is completed after all riders have performed the test.

<table>
<thead>
<tr>
<th>USE CASE DESCRIPTION - optional elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Related information (optional)</td>
</tr>
<tr>
<td>Priority</td>
</tr>
<tr>
<td>Performance target</td>
</tr>
<tr>
<td>Frequency</td>
</tr>
<tr>
<td>Superordinate use case</td>
</tr>
<tr>
<td>Subordinate use case</td>
</tr>
<tr>
<td>Secondary actors</td>
</tr>
<tr>
<td>5. Open issues (optional)</td>
</tr>
<tr>
<td>Open issues</td>
</tr>
<tr>
<td>6. General issues)</td>
</tr>
</tbody>
</table>

Table 9: Optional sections of the use case Compute total score.

We will next look at the subordinate use case, also named as Compute total score, which describes what kind of behaviour is expected from a system that should assist the scorers in accomplishing their objective of computing the total scores for all riders in the competition. There are many same elements as in the superordinate use case, but now the scenario includes system’s responds to user’s inputs. The description on Tables 10 and 11 is on a level of functional user requirements.

The scenario describes on a high level the expected exchange of actor’s inputs and system’s responses which are directed towards getting the scores computed. The main scenario describes how the system will behave in response to the actor’s inputs. On a requirements classification scale, the description illustrates functional user
requirements. Table 11 presents only the main flow of events. From a scorer’s point of view the main reasons for an alternative flow of events are an incomplete or ambiguous judge’s sheet and associating marks for a wrong competitor. For system an alternative flow arises if the value of the mark is not an integer between 0 and 10.

<table>
<thead>
<tr>
<th></th>
<th>Characteristics information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case id and name</td>
<td>1 Compute total score</td>
</tr>
<tr>
<td>Goal in context</td>
<td>The judge marks the movements and the transitions in the test performed by the competitor. The system computes the total score.</td>
</tr>
<tr>
<td>Scope</td>
<td>System</td>
</tr>
<tr>
<td>Level</td>
<td>Use case</td>
</tr>
<tr>
<td>Precondition</td>
<td>Competitor’s judge’s sheet and a template for scoring the class are available,</td>
</tr>
<tr>
<td>Postcondition</td>
<td>The total score for the competitor is computed</td>
</tr>
<tr>
<td>Primary actor</td>
<td>Scorer</td>
</tr>
<tr>
<td>Trigger</td>
<td>Scorer requests access to the competitor whose performance is to be scored.</td>
</tr>
</tbody>
</table>

Table 10: Characteristics information of the use case Compute total score.

### 2. Main success scenario

#### Basic flow:

1. System presents the competitors
2. Scorer chooses the rider whose total score is to be computed
3. System presents competitor’s scoring template
4. Scorer inputs the marks from the judge’s sheet into the system
5. System computes the total score
6. System presents the results for the scorer

#### Alternative scenarios

Not applied

Table 11: Main success scenario of the use case Compute total score.
5.3 Example extended test case

Development of test cases can start as soon as there are use cases available. In this example we apply the extended use case test case technique, and use the use case description Compute total score depicted in Section 5.2 as basis. We start the development of the extended use case by identifying the operational variables.

The use case description in the example is on a high abstraction level, and lacks much of the information needed in the extended use case. First we search the explicit inputs and outputs from the main scenario of the use case description. The first operational variable in the description is competitor. The next operational variable identified in the use case description is the scoring template. The scoring template consists of slots for the marks for the movements and to the overall performance in the test. The scoring template must also allow separate input of the marks granted by each judge. Another operational variable could be the judge's sheet. The judge's sheet describes and lists the movements and transitions of the test in a sequential order, and has slots for the numeric marks and the commentary. Judge's sheet also has space for collective marks. The judge's sheet's identifiers are the rider's and horse's names, and the competition class. However, the judge's sheets are rather sources for the information that need to be input to the system than operational variables. However, the judge's sheet defines the contents of the scoring template. The marks granted by the judges are another operational variable group, and a crucial element of computing the total score. There is one more operational variable in the use case description, the total score. Total score is the outcome of the use case, and an important system output. The operational variables represent only inputs and outputs. The prerequisite for all use cases is that the system is on. There are no other significant system states that would affect the execution of use cases.

The next step in creating an extended use case from the use case Compute total score, is to define the domains of the operational variables. We have identified four operational variables: Competitor, scoring template, mark, and total score. Two of the operational variables, marks and total score, have numeric values. Defining the domain of values for the mark is straightforward. According to the dressage rules, the marks must be between 0 and 10. Invalid values are integers beyond that range, or non-numeric values. The domain of the total score would require further investigation, as the total score comprises of two data items, the total points of the rider, and the percentage achieved of the maximum points. The maximum points are the highest possible points one can achieve in a test.
The operational variables with non-numeric domain of values are the competitor and the scoring template. We have not yet defined the composition of the competitor as a data item. The competitor might include properties such as competitor number, first name, family name, name of the horse. The decision what data will be included into the system is part of the requirements specification phase. How the information should be presented to the user is a combination of requirements specification and user interface design. The scoring template is another operational variable that must be broken down into sub elements. The data items included in the scoring template are competitor, competition class, movement and transition descriptions, and slots for marks and commentary.

The third step is developing the operational relation for the use case. As presented in Table 12, the two possible cases for the marks are that the mark input into the system is either valid or invalid. If the judge’s sheets were scanned to the system, other factors would include the identification of the character in the judge’s sheet, and placing the characters in correct slots. The first row indicates that when an invalid mark is input to the system, the system rejects it. In the second row the mark is valid, and the system accepts it and adds to the sum of points so far. The system indicates to the user that it is waiting for the user’s next input.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Operational variables</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mark</td>
<td>Entered mark</td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Valid</td>
<td>Within score domain</td>
</tr>
</tbody>
</table>

Table 12: Partial operational relations for the use case Compute total score

The fourth step is developing test cases. We look at the operational relations in the Table 12, and design a number of test cases for each variant. The number of the tests depends on the decisions made for the test plan. We decide that every variant should be made true once, and false once. A partial test suite for the use case Compute total score is presented in Table 13. The first variant is true when the input mark is invalid. The mark is invalid, if the input is a number outside the range of the domain, or if input is not a non-numeric character. The second
variant is true with all values that are within the range of the domain. Even though the range is small, it would be worthwhile to design additional tests to show how the system reacts to numeric inputs that are outside, or on the border of the value range. One could for example test with negative values, or values that are greater than the highest acceptable value of 10. Additionally one could test the system’s reaction if the input mark has a decimal point, as in ‘10.0’. It has been advised, that the false-case should not be another variant’s true-case [Bin99]. However, in our small example we were not able to come up with a false-case that did not repeat the true-case in variant 2 and vice versa.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Operational variables</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mark</td>
<td>Entered mark</td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
<td>-</td>
</tr>
<tr>
<td>1T</td>
<td>#</td>
<td>-</td>
</tr>
<tr>
<td>1F</td>
<td>True test in variant 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Valid</td>
<td>Within score</td>
</tr>
<tr>
<td>2T</td>
<td>3</td>
<td>Within score</td>
</tr>
<tr>
<td>2F</td>
<td>False test in variant 1</td>
<td></td>
</tr>
</tbody>
</table>

Table 13: Partial test suite for the use case Compute total score.

5.4 Reflections on the example

Our opinion in the light of the example is that use cases are a great technique for eliciting requirements and describing functional user requirements regarding expected interaction between the user and the system. However, the structure of the use case, as we have applied it, does not allow attaching the kind of detailed data directly to a use case description that would be required to transform user requirements into system requirements, or test cases.
For example if the formulae that are needed to compute the result, and data types and value ranges of the inputs are attached to the use case, the actual behaviour does not stand out from the description as clearly as it should. Additionally, many times the same definitions and details are needed in many use cases. If the rules and definitions change, one must update every use case that is affected by the change. In our opinion the best choice would be to maintain a separate data directory where the data relating to the input and output is stored. What is missing in our use case description is the method of referring to the data directory. One should mark or link the nouns in the description referring to inputs and outputs in a way that clearly associates them to the definitions in the data directory. Also the concepts used in use cases must be defined, but stored and maintained separately from the use cases.

One definite advantage of using use cases is that when the scenarios are well structured, it is easy for the reader to see the flow of events in the use case. The textual format also makes them relatively easy to author as long as the author bears in mind the possible pitfalls. Additionally, should one have a need for formal modelling, the structured text is already a step forward from natural language narrative. The use of use case description templates definitely enhances consistency of the requirements specification. Templates assist the author to pay attention to the abstraction level, which has appeared to be a big problem in use case descriptions. We also feel that in a user requirements specification the use cases anchor and give context to other functional, and maybe even non-functional requirements. Naturally there will be requirements that cannot be attached to any of the use cases, but in our opinion it enhances understanding of the requirement if there is a context one can attach it to.

The application domain in the example is not complicated, but we already experienced how difficult it is to avoid attaching the flow of a scenario to user interface elements. A good example of our mind’s diversion towards interface design is in the beginning of the use case Compute total score presented in Table 11 where one must select the competitor whose result is to be computed. The initial idea was to describe in the sequence that ‘the scorer selects the competitor from the list’. However, ‘selecting from a list’ is a reflection of the user interface implementations we have seen previously. The user requirement we want to express is that the scorer can be sure that the total score is calculated for the right competitor.

A system like this would probably benefit from active use of interface prototypes starting at the early phases of requirements elicitation and analysis. There are not that many functions in the system, and the users who necessarily do not have much
experience of computers benefit from a system that is simple and intuitive to use. 'Simple' and 'intuitive' are of course non-functional requirements that cannot be expressed as such in a use case description. However, in our opinion use cases can influence on those qualities indirectly. If the descriptions are authored with care, and they include only the most straightforward behaviour that is needed for the actor to reach the goal, they should promote simplicity also in the design and implementation. It is only our assumption, but based on our learning through the process of writing this thesis, we believe that well-structured and explicit requirements also promote well-structured design.

Our experience is that the extended use cases work well with the use cases. Once the use case scenarios have been developed, applying the extended use case technique is quite straightforward. Despite the small size of the example, we feel that the operational variables table helped to look for the details that were missing in the use case description. The operational variables table summarises the operational variables in a way that supports systematic test case derivation. Additionally it provides a good overview to the complete set of the operational variables which otherwise remain distributed in individual use cases. We felt that the extended use case technique could even be advantageous in supporting the requirements analysis in pointing out what details are still missing in the functional requirements.

Even though extended use cases cover the missing details in the use case descriptions, they only handle functional requirements, and domain constraints up to some extent. Other test case derivation methods must be considered for non-functional requirements. Despite the simplicity of the extended use case technique, there is still a fair amount of skill and experience needed in correct and extensive identification of the operational variables. Testing is exhaustive only if all significant operational variables have been covered.
6 Conclusions

Requirements engineering is an important field in the software engineering. Missing, incomplete or erroneous requirements can lead to software that does not fully comply with the customer’s needs, or performs its duties so badly that the system is in practice useless. One reason why requirements engineers may fail in their task to produce a consistent and exhaustive requirements specification is the communication gap between the customer and the developers. Often the non-technical customers do not understand the language and techniques used by software developers. On the other hand the developers may lack the domain expertise that is needed to get a full understanding of the problem domain.

In this thesis we have explored how the use case technique can help in bridging the communication gap, and how it can be used in the requirements engineering process for specifying functional requirements. We also investigated whether use cases could be used as basis for validating whether the implemented system matches the customers needs and expectations.

Our opinion is that use cases are a well suited technique for requirements engineering. The use case technique can be relatively easily taught to non-technical people. Using one technique from the early requirements elicitation and analysis phases to requirements specification provides continuity to the process. We believe that in communication with the domain experts it is easier to hold on to one technique, and rather use it to iterate and increment knowledge, than to confuse stakeholders with constant introduction of new techniques for each iteration. It is the customer’s responsibility to approve the requirements specification, and having participated in the use case development, it is easier for them to understand the requirements specification in its entirety.

Use cases are not a technique to be applied in all application areas, but for specifying expected behaviour in interactive systems, they appear to work well. The key in using the use case technique to develop scenarios and elicit other requirements is that the requirements engineers work closely with the domain experts. Unless all parties are motivated, empowered and enabled to participate in the development process, the end result will not be optimal.

The points where the success of the project are to be determined are system testing, and finally acceptance testing where the customer validates that the system matches the requirements specification. When the functional requirements have been spec-
ified as use cases, it is worthwhile to use use cases as basis for test cases. When the customer and the domain experts have participated in developing use cases, it should be easy for them to detect discrepancies between the implementation and use case based test results.

There are many techniques for converting use cases into test cases. In our opinion the extended use case test pattern provides the most straightforward method for analysing and adding details to use cases for test case generation. Even though the use cases would not be formally modelled, extended use cases can be applied. In addition, the extended use case could be used as a tool in requirements analysis for mapping the requirements and their details.

There are however aspects in requirements engineering that use cases provide no answer to. Use cases do not for example enhance the touch-and-feel experience of the customer in the requirements engineering phase. Neither can use cases be used to specify all types of requirements, but have to be used in combination with other techniques. Additionally, use cases are not suitable for all projects, but work best when designing interactive systems.

The use cases technique as such is well explored and formulated. The biggest challenge remains with the requirements engineers applying the technique. Unless one is trained to avoid the typical use case pitfalls, use cases may turn out to be not so advantageous as one would expect them to be. The usefulness of the use case technique is very much dependent on the abilities of the person applying it.
References


BoB01 Boehm, B. and Basili, V., Software defect reduction top 10 list. Software Management, 34,1(2001), 135–137.


Got02b  Gottesdiener, E., Top ten ways project teams misuse use cases - and how to correct them, part II. http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/jul02/TopTenWaysJul02.pdf [26.4.2007]


KSW01  Kösters, G., Six, H.-W. and Winter, M., Coupling use cases and class models as a means for validation and verification of requirements specification. *Requirements Engineering, 6, 1 (2001),* 3–17.


MaR05  Maiden, N. and Robertson, S., Developing use cases and scenarios in the requirements process. In *Proceedings of the 27th International Conference on Software Engineering (ICSE’05)*, St. Louis, MO, USA, 2005, 561–570.


SoR05  Sommerville, I. and Ransom, J., An empirical study of industrial requirements engineering process assessment and improvement. ACM Transactions on Software Engineering and Methodology, 14,1(2005), 85–117.


