Electronic version of an article published in **Rumpe, B.; Hesse, W. (Hrsg.): Modellierung 2004, Proceedings zur Tagung in Marburg; GI-Edition - Lecture Notes in Informatics (LNI), P-45, pp. 59-74**

An Integrated Quality Assurance Approach for Use Case Based Requirements

Christian Denger
Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6
67661 Kaiserslautern
Germany
denger@iese.fhg.de

Barbara Paech
University of Heidelberg
Institute for Computer Science
Im Neuenheimer Feld 348
69120 Heidelberg
Germany
paech@informatik.uni-heidelberg.de

Abstract: Since their introduction, use cases (UCs) have become increasingly important for the specification of software requirements. Model driven development approaches like the Rational Unified Process base the whole software life cycle on UCs. Therefore, high quality UCs are a prerequisite for project success. Despite the high importance of their quality, UC driven approaches often lack systematic and integrated quality assurance techniques. Only ad-hoc recommendations, creation guidelines, and a few checklists for inspection are available in the literature. If at all, these techniques are developed and used seperately, so that one class of defects is addressed by several techniques and other classes are not addressed at all. In this paper, we present an integrated approach that combines UC creation guidelines, UC inspections, and simulation in a systematic way. The techniques are combined based on a classification of defects in UC development. Each technique is focused on a different set of defect classes. The integrated approach is systematically developed and thus more efficient in its application.

1 Introduction

Since the introduction of the unified modeling language (UML) [BGJ99] as a de-facto standard in industry software development, use cases (UCs) have become one of the most important techniques for specifying software requirements. As in the Rational Unified Process (RUP), UCs often drive the whole software development life cycle; that is, all development steps are based on UCs. A UC driven development approach encompasses several advantages. According to [Kru99], it is most important that the UCs link the requirements to other software artifacts such as design, implementation, and test cases. Thus, they help to synchronize the content of the various models. UCs provide a common basis for communication between the different stakeholders (users, customers, management, designers, and testers), which is fundamental for understanding the system and building it right. Moreover, UCs provide a means for project planning purposes such as iteration planning and effort estimation. There are several representations for UCs: text or sequence diagrams, or

activity diagrams. In this paper, we concentrate mainly on textual UCs, as they are most common in industrial practice.

However, specifying system requirements with UCs is not as easy as it might look. Common challenges are that the UCs do not represent the system behavior required by the customer (incorrect, not complete or inconsistent), that the UCs are incomprehensible to some stakeholders, that too much effort is spent on the UC specification (overspecification, unnecessary information), or that the set of UCs overlaps in the described behavior, i.e., different UCs are not clearly separated with respect to the described system functionality. If these challenges are not addressed, poor quality of the UCs threatens the whole software development process: In case that defects remain undetected in the UCs, they can propagate via analysis and design into the code and the final system. This causes undesired and incorrect behavior and results in costly rework activities. The cost of a defect (depending on its class) increases by a factor of 3 – 10 [Boe81], [BB01] per development phase. Thus, the detection and correction of defects in the UCs is one of the most cost-efficient quality assurance techniques [Kru99].

Despite the importance of high-quality UCs, there are only few approaches that focus on quality assurance for UCs. Most common are creation guidelines for UCs that should ensure high quality in a constructive way [Coc01], [AM01], [BS03]. In addition, recommendations can be found in the literature on how to avoid certain quality flaws in the UCs [Lil99]. Also, some inspection approaches define checklists that can be used to detect defects in UCs [AS02], [Pet02]. All of these approaches address some, but not all defects. Furthermore, they are developed as if they were applied as the only quality assurance technique. In addition, developers are left alone with the question of when to use which technique.

To overcome these problems, we developed an integrated quality assurance approach for UC-based requirements specifications. The main idea of our approach is that we analyzed common defects in, and challenges of UCs modeling. Based on this analysis, we developed different quality assurance techniques that address certain defects and challenges. We combined constructive UC creation guidelines with UC inspections and simulation in such a way that they form an integrated quality assurance approach. Integrated means that the different techniques focus on different quality aspects. The idea of our approach is that one technique is more efficient than the other in detecting certain classes of defects. For example, constructive guidelines are more appropriate to ensure structural aspects of the UC (e.g., naming conventions and use of active voice in the UC scenarios). On the other hand, inspections are appropriate for identifying subtle, logical defects in the UCs, which is almost impossible with creation guidelines. Applying simulation allows efficient identification of defects in the dynamic behavior, which would be extremely time-consuming in an inspection. Focusing the techniques on different defect classes increases the coverage and reduces the overlap (i.e., one quality aspect is addressed by one and only one technique). This increases the effectiveness and efficiency of the overall quality assurance approach; that is, more (major) defects can be found with less effort.

The remainder of this paper is structured as follows. In Section 2, common defect classes in UC modeling are summarized. Section 3 describes the integrated quality assurance approach we developed and how this approach addresses the identified

defects. The approach is discussed with respect to related work in this area. In Section 4, some results of the evaluation of different aspects of our approach are presented. Section 5 gives a conclusion and briefly describes future work.

2 UCs and Quality Challenges

The focus of this section is the definition of quality criteria and common defects that affect UC quality and therefore should be addressed by a quality assurance approach. We define the most important quality criteria for requirements documents in general. Based on this definition, we present a defect classification scheme that is tailored to UCs. This schema summarizes the most common defects that have an impact on the defined quality criteria. In addition, we present challenges and common pitfalls in UCs modeling that prevent a requirements engineer from creating high quality UCs. We relate these to the defect classification. The defect classification then forms the baseline for the definition of our integrated quality assurance approach.

2.1 Quality Criteria for UC Models

The IEEE standard for requirements specification [IEEE] lists a general set of quality criteria for specifications, namely: *consistency, completeness, correctness, unambiguity, verifiability (testability), changeability, traceability, prioritsation.* The first four are general criteria for documents, the last four address specific concerns of developers using the specification: verifiability is important for the testers, changeability is important for maintenance, traceability for maintenance and project management, and prioritisation for project management. To cover all stakeholder concerns, we have extended this general scheme with *comprehensibility* (easy to read for all stakeholders) and *feasibility* (necessary for designers) as well as *adequate level of detail* (avoiding overspecification). Based on these quality criteria, we developed a defect classification scheme for UCs. Table 1 shows in the first column the defect class, which is a negation of the quality criteria. In the second column, a definition for the defect class is given that is specific for UCs, and the third column provides an example for a defect of the defect class. Note that incomprehensibility typically affects all other quality criteria. Also, we address traceability under comprehensibility, as both relate to structuring means.

Defect Class	Description	Example
In-correct-ness	The UC does not match the expected or intended behavior; that is, the information presented in the UC is wrong and does not represent the user requirements.	The flow of a UC does not represent the flow of activities expected by the user.
Incom-plete-ness	The UC does not contain all necessary scenarios. The UC set does not contain all necessary UCs. Information that is	An important exception is not specified, a certain actor is not considered.

Defect Class	Description	Example
	required for subsequent activities is not present.	
Inconsistency	A piece of information of a single UC or of different UCs is described in at least two different, incompatible ways so that there is a contradiction between them.	The quality constraints of a UC contradict the event flow. One user action in two different UCs requires contradictory system behavior.
Ambiguity	Elements of the UC can be interpreted in two or more ways. Thus, it is not clear which of the interpretations are true.	A condition containing "and" and "or" does not explicitly state the required bracketing.
Incomprehensibility /intraceability	The UC is difficult to understand and comprehend. The UC is not specified according to a template.	The event flow described in the UC is too complex due to many "include" relationships. The template is not adhered to.
Intestability	The behavior described in the UC cannot be validated by means of test cases due to logical or physical constraints. That means there is no way to check whether the system fulfills the UC.	It is impossible to derive the system response to a certain user input.
Inchangeability	The UC is prone to change or difficult to change.	Details of the user interface are mixed with essential behavior.
Infeasibility	The behavior described in the UC cannot be implemented.	It is not possible to derive an initial design of the system from the UCs.
Overspecification	The information given in the UC is irrelevant or too detailed in the sense that it prescribes an implementation.	Details of internal system behavior are described in the UC. An actor not necessary for the system behavior is described in the UC.

Table 1: UC Defect Classification

This detailed defect classification gives an initial overview of the potential defects that can occur in UC modeling (diagram and textual description). This overview is the starting point for a profound planning of quality assurance activities; that is, based on the defect classification, quality assurance activities most suitable to address a certain defect class can be identified.

In the literature on UCs, pitfalls typical for UCs [Lil99], [Fir] are mentioned. In the following we map these pitfalls to the defect classes explained in Table 1. [Lil99] and [Fir] state the following examples:

- For incompleteness: System boundary is not defined. Associations between UCs and actors do not fully describe who can do what with the system (e.g. only focus on objects or on user interface). UC modeling is stopped too early (difficult to determine when UC modeling is finished).
- For inconsistency: System boundary varies for different UCs (that means UCs are on different abstraction levels). Actors are named inconsistently. UCs interfere with each other (as they have been developed focusing on single flows).
- For incomprehensibility: UCs are written from the system point of view, not the actors point of view; e.g., UC names describe system reactions, not actor goals. UC model looks like a dataflow or process model due to the use of 'extends' and 'uses' relationships. Too many UCs because the actor goals are too fine-grained. Too many relationships between actors and UCs because the actor roles are too coarse-grained. Text too long because UC covers too many instances. UC contains too many if-branches and loops. UCs lack contexts. UC terminology is not adequate for users.
- For inchangeability: UCs are associated with user interface structure.
- For overspecification: Steps of the UC describe internal functionality rather than interaction.

2.2 Existing Quality Assurance Approaches for UCs

The literature also gives hints on how to cope with these quality problems. It provides templates [Coc01], [Lil99], [Fir], guidelines for creating UCs [BS03], and checklists for inspecting UCs [AS02], [Pet02]. The recommendations are typical for guidelines and checklists. However, they do not cover all possible defects that can be dealt with through guidelines and checklists. For instance, guidelines or checklists that give advice on how to use natural language in an unambiguous way could address ambiguity defects. Also, the described techniques were developed independently from each other. Therefore, they often address similar or the same defects resulting in an overlap of the addressed defect classes. In order to improve the efficiency of quality assurance activities, this overlap should be reduced; that is, the different techniques should address different defects. Finally, the recommendations and guidelines at hand do not provide help for the prevention or detection of the really difficult defects like infeasibility, intestability, and serious inconsistency defects resulting from interference between UCs. Most of the guidelines and checklists focus on pure structural and syntactical defects. However, the real expensive defects are on a more subtle (logical) level. Thus, additional quality assurance techniques are required that address such defects. We show how to combine approaches that address structural defects with those that focus on more subtle defects in the next section.

3 An Integrated Quality Assurance Approach

The focus of this section is the description of specific UC quality assurance techniques and their integration. For each technique (creation guidelines, inspections,

simulation) we briefly describe the basic concepts and show how the technique contributes to the quality of UCs with respect to the criteria described in the last section. In addition, we show how the quality assurance techniques are combined into an integrated approach.

3.1 Guidelines for Creating UCs

UC creation guidelines can mainly deal with the document and with structuring-related quality criteria such as *comprehensibility*, *unambiguity*, and *completeness*. Our guidelines focus on UCs as part of the requirements specification. They are used as input for deriving a more refined system specification. In general, we do not recommend including all details of the system specification into the UCs, since they will get too long. In any case, one should make sure that the system details are separated from the main UC description. We have collected the guidelines from literature, e.g., [Coc01], [RA98], and from our experiences regarding requirements engineering projects. Guidelines that are reported in the literature are referenced. Due to space limitations, we can only sketch the guidelines for creating UCs; the full approach can be found in [DPB03]. Our guidelines comprise 4 main steps, which are briefly described in the following paragraphs based on the example of a door control unit for a car. The door control unit (DCU) allows several actors (driver, co-driver) to position their windows and their seats. Moreover, the DCU allows the driver to position the outside mirrors, and it controls the central locking system. Passengers in the back can also position their windows. Note that we use an embedded system example. This shows that our guidelines are not only valid for business systems, which is often the case for UC guidelines.

Step 1: Identify Actors and their Tasks
Identify the most important actors of the system. Actors are roles not persons. Identify the tasks of these actors. Tasks are characterized through goals that actors want to achieve. In order to capture the user's point of view, it is important to abstract as much as possible from technical solutions. Tasks are visualized in UC diagrams. In contrast to ordinary UC diagrams, we distinguish two kinds of task visualizations: Those tasks that are mainly influenced by the user are visualized as bubbles crossing the border between system and environment. Tasks that mainly concern system reaction are shown inside the border. In this step, only the former are elicited. The UC diagram connects the tasks and the actors.

Step 2: Identify the Input and Output of the System (i.e., its Context)
Distinguish monitored and controlled variables. Controlled variables describe the system parts controlled in the UCs as well as system data created. Monitored variables capture the different possibilities actors have to trigger the system reaction as well as other system data needed in the UCs. Create a list of monitored and controlled variables, which captures the name and the description. Do not separate inputs that are needed to trigger one task (that is, both inputs are needed to trigger the same task). "Internal identification input", for example, includes the selection of the "Position seat" function as well as the "Identification" given by the actor. Abstract

from user interface details [Coc01], e.g., do not use “seat_position_button” unless it is required that this is a button (e.g., instead of a touch screen). These inputs and outputs help to delineate the system boundary [Lil99], but do not fix the details of the man – machine interface. It is important to keep these details separate from the UC description, because this often changes over time and between different releases. Thus, abstraction supports *changeability* of the UC description.

Step 3: Refine the Tasks According to Variations
Give special considerations to variations of the tasks. Variations are often due to slightly different handling of input and output. If the variation is quite likely and results in significantly different behavior of actor or system, then define new UCs for the different variations. These new UCs should be included in the general UC. If the variation is quite likely, but can easily be described as a case distinction, include this distinction in the UCs. If the variation is not likely, include it as an exception in the UCs. Avoid too many UCs in order to support *comprehensibility* of the UC model.

Step 4: Fill in the UC Template
We provide a template for the textual UC description to ensure their *completeness*. Figure 1 shows an excerpt of this template. In addition, the name of the UC, actors involved in the UC, the goal of the actor, and quality requirements related to the UC are collected. Name and actor can directly be taken from the UC diagram. Then, the name is elaborated with the actor’s goal. This goal is further detailed with the precondition and the postcondition. Preconditions capture conditions needed for successful execution of the UC and are typically established by other UCs. Postconditions define the system state after the UC has executed successfully; that is what is achieved when the UC scenario is performed without exceptions. Next, the monitored and controlled variables relevant for the UC are collected. They can be taken from the lists created in step 2. The explicit collection of monitored and controlled variables supports *traceability* between UCs that overlap on variables.

Description	1. Actor totally moves the window into a certain direction 2. System reacts accordingly [Exception 3.1.: Actor moves the window partially] [Exception 3.2.: Technical problem] [Exception 3.3.: Safety Opening]
Exceptions	• 3.1 Partial movement: → use case “Partial Movement” • 3.2 Technical problem: System does not react completely • 3.3. Safety Opening: System moves the window into its lower end position
Rules	The system activates the “Safety opening” in the case that the actor moves the window upwards but no change of the window position is recognized.
Mon. Variables	Window_Position, Actor_Input: movement type (patially/total) and movement direction (up, down)
Cont. Variables	Window_Position
Precondition	-
Postcondition	Window has new position

Figure 1: Excerpt of example of filled in UC Template

The main step is to describe the normal course of interaction between actor and system in the description facet. Here, we use the essential UCs from [CL99]. To achieve *completeness*, we focus the requirements engineer on four types of exceptions resulting from:

1. actor inputs outside of the UC (e.g., partial movement in Figure 1),
2. boundary values of controlled variables such as limit positions
3. system behavior outside of the UC, but visible to it (e.g., safety opening in Figure 1), or
4. problems in carrying out the system reaction (e.g., technical problem in Figure 1)

To support *comprehensibility* of the main flow, details of the system reaction are captured in the rules facet. The rules facet gives additional information to specific aspects in the main flow, for example, in which case the system will activate the safety opening of the window control (see **Figure 1**).
The separation of the requirements into different facets is an important prerequisite for the efficient derivation of the system specification.

3.2 UC Inspection

Inspections are one of the most efficient quality assurance techniques. Moreover, inspections in the early life cycle are highly effective in adding new views on a software artifact by different stakeholders. Therefore, we use inspections as a second technique to ensure the quality of the UC. We use the perspective-based reading (PBR) approach [BGL96], [Lai00]. The idea of this inspection approach is that the UCs are inspected from the perspective of the most important stakeholders of the UC. Typical stakeholders of the UCs are:

- test engineers who use them as input for test case creation,
- designers who derive high level design diagrams from the UCs,
- customers who take the UC as the main document to check whether all their requirements on the system are captured,
- maintainers who have to perform changes on the UCs

We chose these perspectives in order to complement the creation guidelines described in the last section. The perspectives focus on those defects that are difficult to address in a constructive way. The customer perspective addresses *completeness* and *correctness* in the sense that all requirements are captured in the UCs, as intended by the customers. We support *testability* with the tester perspective, as this perspective focuses on the possibility to derive test cases from the UCs. In the same way, we support *feasibility and adequate specification level* of the UCs with the designer perspective. This perspective ensures that it is possible to derive a reasonably high level design from the UCs. Finally, our maintainer perspective supports *changeability* of the UCs, as this perspective considers how potential changes of the requirements could be realized in the UCs. Other perspectives might also be useful, but in combination with the creation guidelines the above mentioned perspectives are the most valuable ones, since they address defects not considered in the guidelines. In

each project, the used perspectives need to be tailored to the specific project context, and in each project, the set of used perspectives needs to be checked. For example, dependent on the context, different customer perspectives are possible (like marketing department, users, management). In such cases, the different stakeholder needs must be considered in the PBR approach.

In order to support the inspectors in detecting defects, the PBR approach provides reading scenarios that are tailored to the needs of the stakeholders [Lai00]. These scenarios give a step-by-step description of the activities an inspector should perform during the defect detection. A reading scenario consists of three main parts: **introduction, instructions, and questions.**

In the *introduction*, the goal of the scenario is described and the quality aspects that are most important in the particular scenario are defined. Thus, the focus of the inspector is set; that is, it is clarified **what** should be inspected (on which aspects should the inspector focus).

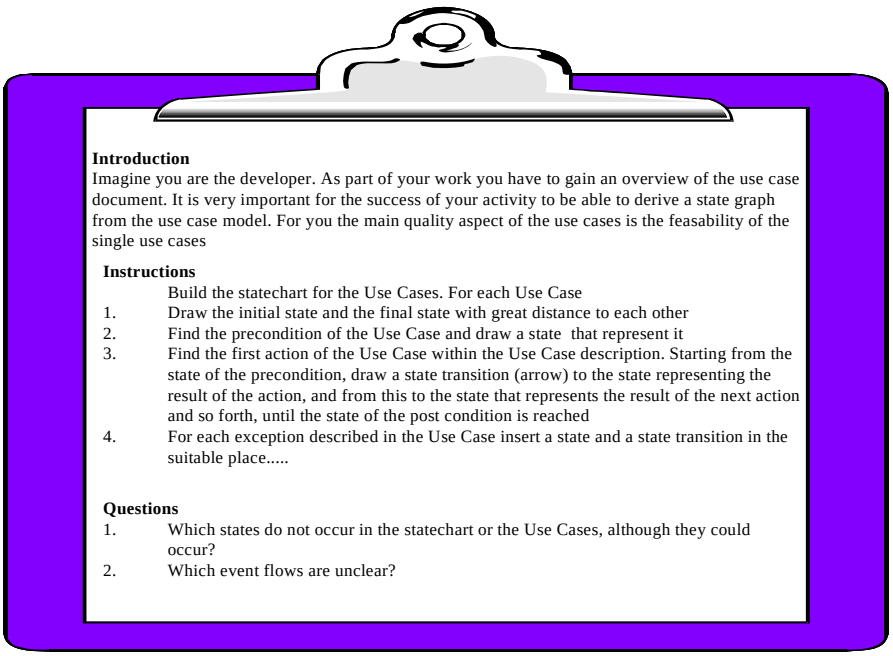
In the *instruction*, an inspector gets concrete guidance on how to perform the inspection in order to detect defects. The instructions define exactly which documents an inspector should use during the inspection, how to read them, and **how** to extract information from them. While identifying, reading, and extracting information, inspectors may already detect defects. One important aspect of the PBR approach is that the inspectors extract information in a way that can be reused in later activities. Therefore, the scenarios give guidance on how to perform typical activities of the assumed perspective. The result of these activities are then real work products of that stakeholder. For example, the PBR approach defines that an inspector who assumes the designer perspective should derive high level statecharts from the UCs. Then, the result of the inspection is not only the detected defects, but also a certain set of artifacts (in this example, some initial design diagrams) that can be used as a starting point later in the process when they need to be derived anyway. Another example is the tester perspective, which asks the inspectors to derive test cases from the UCs. Again,, these test cases can then be used as an initial set of test cases during the acceptance test phase. Thus, no time is wasted in the inspection. As in many cases the inspections cannot cover all elements in the UCs (especially when the system is too complex), the derived artifacts must be viewed as an initial basis for later activities. The inspections cannot substitute these activities.

The motivation for providing guidance for inspectors is three-fold [BL02]: First, the instructions help an inspector to gain a focused understanding of the UCs. Understanding involves the assignment of meaning to information in a particular set of document parts and is a necessary prerequisite for detecting more subtle defects. Second, the instructions require an inspector to actively work with the UCs rather than passively scanning them, which is a prerequisite for a profound understanding and which results in the creation of real artifacts that can be reused. Third, the attention of an inspector is focused on the information most interesting for a particular stakeholder. Thus, the inspector is not swamped with details irrelevant to his or her perspective.

The third element of the reading scenarios are the *questions*. With the profound understanding gained during the performance of the reading scenario, the questions support the inspectors in judging whether the document fulfils the required quality

properties. The inspectors should answer the questions while working with the document.
In Figure 2, an excerpt of the reading scenario of the designer perspective is shown.

Applying the PBR approach results in several benefits: Detecting subtle defects is the most important aspect of an inspection, as subtle defects are the most costly defects in a software system. By focusing the inspectors on specific aspects that are relevant for a perspective, the inspection becomes more efficient, as not all inspectors are searching for the same issues (as in checklist-based inspection). Also, the inspectors concentrate on those aspects that are most important for the perspective. Compared to other reading techniques, the inspectors produce documents during the defect detection. This allows a quality assurance responsible to judge whether an inspection was performed thoroughly or not, as it is possible to look at the produced documents and judge their quality. An additional advantage of this inspection technique (beside the early detection of defects) is that it supports the communication of different stakeholders of the UCs. Thus, we avoid *incomprehensibility*, by bringing together the views of the important stakeholders.



Introduction
Imagine you are the developer. As part of your work you have to gain an overview of the use case document. It is very important for the success of your activity to be able to derive a state graph from the use case model. For you the main quality aspect of the use cases is the feasability of the single use cases

Instructions

Build the statechart for the Use Cases. For each Use Case

1. Draw the initial state and the final state with great distance to each other
2. Find the precondition of the Use Case and draw a state that represent it
3. Find the first action of the Use Case within the Use Case description. Starting from the state of the precondition, draw a state transition (arrow) to the state representing the result of the action, and from this to the state that represents the result of the next action and so forth, until the state of the post condition is reached
4. For each exception described in the Use Case insert a state and a state transition in the suitable place.....

Questions

1. Which states do not occur in the statechart or the Use Cases, although they could occur?
2. Which event flows are unclear?

Figure 2: Excerpt of the Reading Scenario from the Perspective of a designer (in our case the creator of the high-level statecharts in the system specification)

3.3 Simulation of Requirements

The most difficult defects are *inconsistency* and interference issues. They could be detected by experienced inspectors (but only with some effort), but no detailed guidance can be given in the reading scenarios. Therefore, we integrate simulation as a further means of quality assurance that addresses such defects more efficiently. To apply simulation, textual UCs must first be transformed into a formal model. Here one can use sequence diagrams. However, this only allows the simulation of individual UCs. The typical formal model that allows for simulation of the complete system behavior are statecharts (SCs) [Har98]. Typical approaches for transforming UCs into SCs require already formal UCs in terms of sequence diagrams. To avoid using formal sequence diagrams as an intermediate between textual UCs and formal statecharts we have defined guidelines for deriving SCs directly from textual UCs. The SCENT-approach [RG99] also gives guidelines to derive Scs from textual UCs. However, the SCs are quite different from ours. The states of the SCs represent actions of an actor or the system. They do not describe the internal behavior of the system and the main focus is to check the completeness of one UC-description by integrating all scenarios belonging to one UC. In contrast our guidelines help to derive an integrated set of SCs representing the system behavior necessary to realize all UCs. The full guidelines are published in [DKK03]. Here we only sketch the main idea.

To keep the guidelines simple we preserve the structure of the UCs in the SCs. So we map input and output as well as UCs to different classes and define SCs for each of these classes. The SCs for the variables are quite straightforward and reflect the major states of these variables (e.g., door locked, door open). The SCs for the UC start in an idle state and react to the trigger of the UCs. Exceptions and preconditions are reflected in this reaction. The main behavior reflects the different steps of the UCs. Exceptions lead back to the idle state and may trigger other UCs.

On the formal SCs one can use simulation tools like Rhapsody [Rha] to validate the dynamic behavior of the system. On the one hand, this helps to detect *incorrectness* problems, as the user can directly see how the specified system behaves. Moreover, the transformation and simulation allows to check for *inconsistencies* in the SCs and thus in the UCs regarding the dynamic behavior. Inconsistency is often detected during transformation, as it is not possible to define a clear system reaction. Further inconsistencies are detected through simulation when the system does not react as expected. In particular, *interference* defects can also be detected, since with simulations it is easily possible to simulate several UCs concurrently by means of simulating their corresponding SCs.

3.4 Summarizing the Integrated Approach

By combining constructive and analytic quality assurance techniques, we are able to address all the quality criteria of the IEEE standard. In Table 2, we summarize how each of our techniques contributes to the fulfillment of a high quality requirements specification:

	Creation Guidelines	Inspections	Simulation
Incompleteness	X	(X)	
Incorrectness		X	X
Inconsistency		(X)	X
Ambiguity	X	(X)	(X)
Incomprehensibility	X	(X)	
Intestability		X	(X)
Inchangeability	(X)	X	
Infeasability		X	
Overspecification	X	X	

Table 2: Quality Assurance Technique - Defect Matrix

An "X" in the table indicates that defects related to a certain quality aspect (as described in Table 1) are addressed with a quality assurance technique. An "(X)" indicates that defects are indirectly addressed with the quality assurance technique. One example here is the use of inspections. When searching for defects limiting feasibility or testability, the inspector will also identify defects related to other classes (e.g., completeness, ambiguity).

Our integrated approach combines the techniques, so that the UC creation guidelines focus on structural aspects related to completeness (e.g., all important exceptions are considered, all template elements are filled in). Moreover, the guidelines address aspects related to the use of natural language (understandability of sentences, use of unambiguous terminology). The inspection focuses on those aspects that are difficult to address in creation guidelines, such as feasibility, testability, and changeability analysis. It also focuses on more subtle (logical) defects that are not necessarily related to structural aspects. Moreover, our inspection approach helps to involve all the important stakeholders of the UCs through tailored perspectives and, therefore, supports communication about and a common understanding of the requirements. Simulation is integrated so that serious correctness and consistency defects and especially defects resulting from interrelationships between UCs (interference aspects) are addressed. Such defects are extremely difficult to identify in an inspection. The combination of the different techniques in such a way is a promising approach to reduce quality assurance effort and achieve higher efficiency.

Our approach for high-quality requirements also bears some risks for the development process. The development of the formal model requires some effort. This should only be spent if the risk of inconsistencies is high. Keeping the structure of the UCs and the SCs very simple is better suited for requirements engineering than a structure that focuses on design optimization. This results in redundancies, in particular in the SC models and the class diagram. Thus, it is necessary to restructure the class diagram and the SCs during system design. The reading scenarios of our inspection approach need to be tailored to the concrete development context and the specific stakeholders. It might happen that additional perspectives need to be considered or that the focus of the described perspectives needs to be changed. If stakeholders are not considered or the existing perspectives are not sufficient, the inspection would be inefficient and not effective. However, tailoring the perspective-based inspection approach is easily

possible by carefully identifying the stakeholders of the UCs and analysing their quality needs. Another risk stems from a development environment in which tool support is not possible. In such a case, all defects that are addressed in our simulation approach should be captured in appropriate reading scenarios for inspection.

4 Evaluation of the Approach

We evaluated several parts of our quality assurance approach in case studies with students. So far, we have focused on the value of the single techniques; i.e., the guidelines, the inspection approach, and the simulation. We have not yet evaluated the integration of the different techniques into a combined approach. In the following, we present the results of our evaluation.

We validated the UC creation guidelines and the UC inspection approach in a case study at the Technical University of Kaiserslautern. Both techniques were used in the practical course "Software Engineering 1" in the summer of 2003. In this course, the students had to develop a building automation system that regulates the temperature and the light in the rooms and floors of a university building. In the working description of the practical course, the system was separated into three sub-systems, namely, "Temperature Control" (Temp), "Light Control" (Ligh), and "User Interface" (UI). 12 students were involved in the case study, with a group of 4 people being responsible for each sub-system. The Temp system comprised 21 UCs and textual scenarios, the Light system 15 UCs and scenarios and the GUI system 34 UCs and scenarios. We did not design an experiment with a control group who did not use our approach, as there is no other established approach ready for teaching. Thus a the control group would have used an ad-hoc approach. This would have contradicted the teaching goals of the practical course. In addition, this would have reduced the number of people giving feedback to our approach. Clearly, with a number of 12 students we cannot provide any statistically significant evaluations. However, making students apply the approach and collecting their feedback is, in our view, an important step towards a more thorough evaluation.
Based on the problem description, each group had to develop UCs for its sub-system with our UC creation guidelines. To evaluate usefulness, we used a questionnaire that was given to the subjects after they completed the UC creation. It was designed following the model recommended by Davis [Dav89]. The model evaluates the general usefulness of a certain technique by means of three basic categories: **Perceived usefulness** "the degree to which a person believes that using a particular technique would enhance his or her job performance"; **Perceived ease of use (applicability)** "the degree to which a person believes that using a particular technique would be free of effort"; **Self-predicted future use** "the degree to which a person would use a particular technique again in the future". For each category, the students had to state their degree of agreement to certain statements (e.g., "the guidelines accelerate the UC creation or the guidelines improve the effectiveness of the UC creation") on a scale from 1 (total disagreement) to 6 (total agreement). Thus, it is possible to evaluate the median for each category.

Regarding applicability, three statements had to be rated. Thus, the maximum value (most positive case) is 18. We measured a median of 12. Therefore, the students perceive the guidelines as applicable, but there is still improvement potential, as we did not reach the maximum value. The second element of the evaluation model is the perceived usefulness of the guidelines. The summarized results again show a positive perception of the usefulness of the guidelines. Five statements had to be rated in this category (maximum value 30). We measured a median of 23 and therefore conclude that the subjects agree that the guidelines are useful for performing their task. Regarding the self-predicted future use, the subjects had to agree with one statement. 10 out of 12 subjects (83.3%) agreed that they would use the guidelines again in a future project. To summarize, the overall impression of the guidelines is positive. The evaluation indicates that the guidelines are useful and applicable to create the UCs. Most of the subjects would use the guidelines again in a future project. However, the results also indicate that the guidelines can still be improved A more detailed presentation of the results can be found in [DPB03].
After the students created the UCs, they had to perform inspections on the UCs. In this task the students used our perspective-based inspection approach. We evaluated the impact of the detailed descriptions provided by our inspection approach (usefulness of the reading scenarios). In detail, we analyzed the following hypothesis:

Hypothesis H_1—Team Effectiveness: Inspection teams find more defects with the help of the reading scenarios than with a comparable checklist

Hypothesis H_2—Team Efficiency: Inspection teams find more defects per time unit with the help of the reading scenarios than with a comparable checklist

In addition, we analyzed the subjects' perception regarding the support provided by the PBR approach, again using a questionnaire.
The results of the study provide weak tendencies that our inspection approach results in more effective inspections and is perceived as very helpful to support individual defect detection. In detail, the study showed that: PBR finds between 23% and 40% more defects than a checklist approach for the Temp and GUI subsystems. For the subsystem Light, the checklist approach finds about 32% more defects than PBR. The results are statisticaly significant for the GUI system ($p=0.08$), but not for the other sub-systems. Regarding the efficiency the study showed that the checklist-based approach is more efficient for the Temp and the Light subsystems. One possible interpretation for this is that the effort spend in the PBR approach, which focuses on a profound understanding of the document under inspection, pays of only in more complex systems. As the GUI system was the most complex system (it contained the most UCs), the PBR approach performed best for this subsystem. In the two other subsystems the effort of the PBR approach was to high as they were not that complex (i.e., the checklist was sufficient to gain the understanding of the system). However, this is an hypothesis an needs to be validated in future research activities.

The evaluation of the questionnaire showed that the PBR approach is perceived to be as applicable as the checklist-based approach but harder to understand. The subjects' stated that the PBR approach is more useful compared to a checklist approach and that the reading scenarios, especially, are perceived as highly valuable. Eight out of eleven students (72.2%) agree that the reading scenarios are helpful in performing the

defect detection. Even though we could not prove our hypothesis, the results indicate that our inspection approach is a valuable means for improving the quality of the UCs and that it is perceived more useful than traditional approaches. Further results of the inspection case studies can be found in [DCL04].

In addition to this study, we evaluated the usefulness of the simulation approach in a seminar at the Technical University of Kaiserslautern. 4 students used the simulation to detect defects in UCs of the door control unit of a car. We got positive feedback regarding the simulation. The detection of more subtle defects and defects regarding the dynamic interaction of UCs are perceived as the main advantages.

The presented results are (with one exception) not statistically significant. They can only serve as initial results that are a first step towards more formal empirical evaluations. In addition, the results are based on case studies with students, which limits their generalizability even more. However, in our project, the student case study was the only possibility to initially analyze our results, and we perceive this evaluation as a necessary step. In future activities the integrated approach has to be compared to an approach that does not combine the different techniques in a systematic way, and the techniques need to be analyzed in controlled experiments.

5 Conclusion

In this paper we have described an integrated quality assurance approach for UCs. The core aspect of our approach is the combination of constructive techniques (UC creation guidelines) with analytic quality assurance techniques (inspections and simulation) for UCs. The combination is based on a defect classification for UC models. This classification enables systematic combination such that guidelines, inspections, and simulation address different kinds of defect classes. We showed that guidelines are valuable for the prevention of structural and syntactic defects, and inspections are suitable for detecting subtle logical defects. Simulation is integrated so that serious consistency defects resulting from the interference between UCs can be efficiently detected. With this approach we hope to improve the overall efficiency and effectiveness of quality assurance. The evaluation of our approach gives first evidence that each part contributes to the overall quality improvement of the UCs. However, a detailed analysis of our integration approach needs to be performed in future empirical studies, particularly in industry. The aim of this study would be to show that the overlap of detected defects can be reduced with the combined use of the different techniques. The results of our evaluation motivate us to continue working in that direction.

References

[AM01] Amour, F.; Miller, G.; Advanced UC Modelling; Addison Wesley, 2001

[AS02] Anda, B.; Sjøberg, D. I. K.; Towards an inspection technique for UC models; In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), pages 127 – 134, Italy, 2002

[BB01] Boehm, B.; Basili V. R.; Software Defect Reduction Top 10 List; IEEE Computer, Vol. 34, No. 1, January 2001

[BGJ99] Booch, G.; Rumbaugh, J.; Jacobson, I.; The Unified Modelling Language User Guide; Addison-Wesley, 1999

[BGL96] Basili, V.R.; Green, S.; Laitenberger, O.; Lanubile, F.; Shull, F.; Sorumgard, S.; Zelkowitz M.; The Empirical Investigation of Perspective-based Reading; Empirical Software Engineering 1, pages 133–164, 1996

[BL02] Bunse, C.; Laitenberger, O.; Improving Component Quality Through the Systematic Combination of Construction and Analysis; In: Proceedings of Software Quality Week Europe, Belgium, 2002

[Boe81] Boehm B. W.; Software Engineering Economics; Advances in Computer Science and Technology; Prentice Hall, 1981

[BS03] Bittner, K.; Spence, I.; UC Modeling; Addison Wesley, 2003

[CL99] Constantine, L.; Lockwood,L.: Software for Use, Addison Wesley, 1999

[Coc01] Cockburn, A.; Writing Effective UCs; Addison Wesley, 2001

[Dav89] Davis, F. D.; Perceived usefulness, perceived ease of use, and user acceptance of Information technology; MIS Quarterly, pages 319–340, 1989

[DCL04] Denger, C; Ciolkowski M; Lanubile, F: Does Active Guidance Improve Software Inspections? A Preliminary Empirical Study; Accepted for publication in the proceedings of the IASTED conference 2004, Innsbruck, Austria

[DKK03] Denger, C.; Kerkow, D.; Knethen, von A.; Paech B.; A Comprehensive Approach for Creating High-Quality Requirements and Specifications in Automotive Projects, 16th International Conference "Software & Systems Engineering and their Applications" Paris - December 2, 3 & 4, 2003

[DPB03] Denger, C.; Paech, B.; Benz, S.; Guidelines – Creating UCs for Embedded Systems; IESE Report No. 078.03/E, 2003

[Fir] Firesmith, G.; UCs: the Pros and Cons; http://www.ksc.com/article7.html

[Har98] Harel, D.; Modeling Reactive Systems with Statecharts; McGraw-Hill; 1998

[IEEE] IEEE Recommended Practice for Software Requirements Specification, Standard 830-1998, 1998

[Kru99] Kruchten P.; The Rational Unified Process, An Introduction; Addison Wesley; 1999

[Lai00] Laitenberger, O.; Cost-effective Detection of Software Defects through Perspective-based Inspections; PhD Thesis in Experimental Software Engineering; Fraunhofer IRB Verlag, 2000

[Lil99] Lilly, S.; UC Pitfalls: Top 10 Problems from Real Projects Using UCs; Proceedings Technology of object-oriented languages and systems (TOOLS), pages 174-183, 1999

[Pet02] Pettit, R.; Establishing Inspection Criteria for UML Models; tutorial at the 5th Conference of the Unified Modelling Language (UML 2002); Germany 2002

[RA98] Rolland, C.; Achour, C. B.; Guiding the Construction of Textual UC Specifications, Data & Knowledge Engineering Journal, Vol 25, N°1-2, pages 125-160, North Holland, Elsevier Science Publishers, March 1998

[RG99] J. Ryser, M. Glinz: A Practical Approach to Validating and Testing Software Systems Using Scenarios, Proceedings Quality Week Europe, 1999

[Rha] http://www.ilogix.com/products/rhapsody/index.cfm