Electronic version of an article published as **Informatik - Forschung und Entwicklung, Vol. 20, Issue 1-2, 2005, pp. 11-23**
[doi: 10.1007/s00450-005-0198-4]

Achieving High Quality of Use-Case-Based Requirements

Christian Denger
Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6
67661 Kaiserslautern
Germany
denger@iese.fhg.de
+49 6301 707 196

Barbara Paech
University of Heidelberg
Institute for Computer Science
Im Neuenheimer Feld 348
69120 Heidelberg
Germany
paech@informatik.uni-heidelberg.de
+49 6221 54 58 10

Bernd Freimut
Fraunhofer Institute for Experimental Software
Sauerwiesen 6
67661 Kaiserlautern
Germany
freimut@iese.fhg.de
+49 6301 707 253

Zusammenfassung: Seit der Einführung von Use Cases hat deren Bedeutung zur Spezifikation von Anforderungen stetig zugenommen. Die Qualität der Use Cases ist ein entscheidender Faktor für den Erfolg des Entwicklungsprozesses, da die meisten Entwicklungsschritte auf den Use Cases aufbauen. Trotz der extremen Wichtigkeit der Qualität der Use Cases stellen die meisten use-case-basierten Entwicklungsansätze keine oder nur unzureichende integrierte qualitätssichernde Maßnahmen bereit (z.B. ad-hoc Empfehlungen, Erstellungsrichtlinien, einige Checklisten zur Inspektion von Use Cases). Diese Techniken werden in den meisten Fällen unabhängig voneinander eingesetzt, so dass bestimmte Fehlerklassen in den Use Cases durch mehrere Techniken, andere Fehlerklassen überhaupt nicht adressiert werden. In diesem Artikel wird ein integrierter Ansatz vorgestellt, in dem Use Case Erstellungsrichtlinien, Inspektionen und Simulation in systematischer Weise miteinander verknüpft werden. Der Ansatz basiert auf einer Fehlerklassifikation für Use Cases, die als Grundlage dient, die verschiedenen Techniken auf bestimmte Fehlerarten zu fokussieren.

Keywords: Anforderungen, Use Cases, Richtlinien, Qualitätssicherung, Inspektionen, Perspektiven-basiertes Lesen Fehlerklassifikation, Qualität

Abstract: Since their introduction, use cases (UCs) have become increasingly important for the specification of software requirements. 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 separately, 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. We base our combined approach on a defect classification for use cases. This classification enables the requirements engineer to focus the different techniques on different types of defects.

Keywords: Requirements, Use Cases, Guidelines, Quality Assurance, Inspections, Perspective-based Reading, Defect Classification, Quality

1 Introduction

Since the introduction of the unified modeling language (UML) [BSJ99] as a de-facto standard in industrial software development, use cases (UCs) have become one of the most important techniques for specifying software requirements. As recommended by 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], 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.

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, that the UCs are incomprehensible to some stakeholders, that too much effort is spent on the UC specification, that different UCs are not clearly separated with respect to the described system functionality or that some UCs are infeasible (i.e. not implementable). 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 thus cause undesired and incorrect behavior resulting in costly rework activities. The cost of a defect increases by a factor of 3 – 10 [Boe81][BB01] per development phase, depending on the type of the defect. 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 potential defects types. Furthermore, they are developed as if they were applied as the only quality assurance technique. This can lead to redundant effort, in the sense that different techniques double check the same types of defects. Finally, developers are often 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. We especially concentrate on textual UCs as they are most common in industrial practice. The main idea of our approach is that we base the quality assurance activities for UCs on a defect classification scheme that captures common defects in-, and challenges of UCs modeling. Based on an analysis of the defects, we developed tailored quality assurance techniques that focus on special types of 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. 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 (e.g. infeasible requirements or poor maintainability) which is almost impossible with creation guidelines. Applying simulation allows efficient identification of defects in the dynamic behavior and improves the understanding of the interplay of the different use cases, 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 in the ideal case 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, the basic ideas of and the motivation for defect classes is given and a defect classification for UC modeling is defined. 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 preliminary evaluation of our approach are presented. Section 5 gives a conclusion and briefly describes future work.

2 A Defect Classification Scheme for UCs

In order to systematically combine different quality assurance techniques, we consider the types of defects that can be detected with a particular technique. Since a single technique alone cannot address all types of defects equally well, several techniques should be integrated in a way that (1) all kind of defects are targeted by the most suitable technique, and (2) the overlap between types of defects found by different techniques is reduced.

The core of this approach is consequently to derive an appropriate classification of defect types that enables such a systematic integration. To motivate the selection of our defect classification scheme, we first discuss general aspects of defect classification and then, we present our approach to define a defect classification scheme for UCs. In this approach we start with common defects and pitfalls that affect UC quality and therefore should be addressed by an integrated quality assurance approach.

2.1 Basics of Defect Classification

Defect classification plays an important role when measuring software processes. This importance is explained by the fact that defects carry a lot of information that can be analyzed in order to characterize the quality of the development processes, of the quality assurance processes, and of the resulting products.

A defect classification scheme in general contains one or more defect classification attributes that capture various aspects of a defect. For example, [Mel92] proposes a framework of eight high-level key attributes that capture different defect aspects. Each of these defect classification attributes is measured on a nominal or ordinal scale with a set of pre-defined values, the so-called defect classes. The challenge in designing a defect classification is therefore to select an appropriate aspect and corresponding defect classes.

Generally, there are many aspects of a defect: Defects are inserted due to a particular reason into a particular piece of software at a particular point in time. Defects are detected at a specific time with a specific technique by noting some sort of symptom and they are corrected in a specific way. Consequently, there are many different defect classification schemes that target different defect aspects for different purposes. For example, the IEEE Standard Classification for Software Anomalies [IEEE94] aims at tracking the progress of defects through the Defect Resolution Process. The

Hewlett-Packard Scheme [Gra92] aims at deriving process improvement proposals, and the often-used Orthogonal Defect Classification (ODC) Scheme [Chi92] aims at controlling the progress of a development project. In addition to these well-known schemes, defect classification schemes have been developed for specific quality assurance techniques: [Por95] used the set {Missing Functionality, Missing Performance, Missing Environment, Missing Interface, Ambiguous Information, Inconsistent Information, Incorrect or Extra Functionality, Wrong Section} to characterize defects found in requirements inspections, while [BGL96] used the set {Omission, Incorrect Fact, Inconsistency, Ambiguity, Extraneous Information}. [AS02] used this scheme to develop a UC defect taxonomy. The basic idea of the taxonomy is to link UC elements (Actors, Use Cases, Event Flow, Variations, Relations, Triggers) to this defect classification, i.e., to define what the classes mean for the elements (e.g., omission means for actor that not all actors where specified in the UC diagram or the UC description).

[Fre01] presents a process for developing defect classification schemes as well as quality criteria for a good defect classification scheme, which we followed in order to derive a defect classification for UCs. In short, the process is as follows: In a first step it is necessary to decide, based on the indented usage of the scheme, which aspects of a defect are to be captured in a defect classification attribute. In order to make this aspect explicit, it is recommended to define the meaning of the attribute, for example in the form of a question. In a second step, an appropriate set of defect classes are to be derived (for each attribute). This set of defect classes should be tested with a sample of real defects in order to ensure its applicability. In addition it should be checked if the set of defect classes is orthogonal (i.e., for a given defect one class at most is possible) and complete (i.e., for a given defect at least one class is possible). In a third step, all defect classes should be documented with a definition that states, when a defect is to be assigned to a given defect class. This definition helps data collectors to select the right defect classes and therefore contributes strongly to data quality.

2.2 A Defect Classification for UCs

Following the process for developing defect classification schemes presented above, it is first necessary to select a relevant defect aspect that is to be captured. Since the basic assumption of our integrated approach is that there are different types of defects and that different quality assurance techniques focus on different types of defects, we wanted to capture the aspect of how a defect is detected. A survey of the existing schemes revealed that the attribute Defect Trigger of the ODC scheme addresses this aspect. In particular, this attribute addresses the question *What were you checking when you detected the defect* and represents thus a question crucial for distinguishing defects found or prevented with different techniques. In the original definition the attribute values are specifically tailored to the IBM domain and are not directly usable for UCs. Thus, we derived a new set of attribute values for this attribute by adhering to the original definition of the Defect Trigger. Our rationale was that the set of attribute values should contain *quality criteria* of a high-quality UC. Since developers check the document against these criteria in order to find defects or prevent these upfront, these quality criteria are appropriate defect classes for the Defect Trigger. With this rationale in mind, the integration of quality assurance techniques can ensure that all necessary quality criteria are addressed.

In order to define the set defect classes we identified a set of quality criteria. In particular, we used the IEEE standard for requirements specification [IEEE98] as a basis. This standard lists a general set of quality criteria for specifications, namely: *consistency*, *completeness*, *correctness*, *unambiguity*, *verifiability (testability)*, *changeability*, *traceability*, *and prioritisation*. 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 and address all relevant quality criteria, 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 over- and under-specification). 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. The latter two columns ensure that every defect class is well-defined. Note that incomprehensibility typically affects all other quality criteria. Also, we address traceability under comprehensibility, as both relate to structuring means.

This detailed defect classification gives an initial overview of the potential defects that can occur in UC modeling (diagram and textual description).

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	An important exception is not specified, a

Defect Class	Description	Example
	does not contain all necessary UCs. Information that is required for subsequent activities is not present.	certain actor is not considered.
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 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

In order to check whether the defect classification scheme is applicable, we used pitfalls typical for UCs that are mentioned in the literature [Lil99][Fir] and classified them according to our classification scheme (Table 1). In the following the results of this mapping are presented. First, the pitfalls described in the literature and then the matching defect class of our classification are mentioned.

Pitfalls_1: 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). → Incompleteness

Pitfalls_2: 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). → Inconsistency

Pitfalls_3: UCs are written from the system point of view, not the actor's 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. There are too many UCs, because the actor goals are too fine-grained. There are too many relationships between actors and UCs, because the actor roles are too coarse-grained. Text is 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. → Incomprehensibility

Pitfalls_4: UCs are associated with user interface structure. → Inchangeability

Pitfalls_5: Steps of the UC describe internal functionality rather than interaction. → Overspecification

This mapping shows that our classification scheme is indeed usable with typical defects and is therefore a sound starting point for a profound planning of quality assurance techniques. Based on the defect classification, quality assurance techniques most suitable to address a certain defect class can be identified.

2.3 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 only 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. 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, but 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 integration of UC creation guidelines, inspections, and simulation. We briefly describe the basic concepts and show how each technique contributes to the quality of UCs with respect to the defect classes described in the last section. In addition, we show how the quality assurance techniques are combined into an integrated approach. We use "quality assurance techniques" as an umbrella term for constructive quality assurance techniques such as the use of guidelines and templates and analytic quality assurance techniques such as inspections and simulations. The main difference between these two facets of quality assurance is that constructive techniques build in the quality during the creation of an artifact, in our case the requirements. Analytic techniques take an existing artifact as an input and evaluate its quality. However, whenever we do not want to explicitly stress the different meanings we will use the term quality assurance technique for both facets.

3.1 Guidelines for Creating UCs

UC creation guidelines can mainly deal with structuring-related defect classes such as *incomprehensibility*, *ambiguity*, and *incompleteness*. Our guidelines focus on UCs as part of the requirements specification. In that context UCs 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; the full approach can be found in [DPB03]. Our guidelines comprise four 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 approach is 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, for example, the driver is an actor while Bob would be a person, playing the role of the driver. Identify the tasks of these actors. Tasks are characterized through goals that actors want to achieve, for example, the driver wants to move his or her seat in a convenient position. In order to capture the user's point of view, it is important to abstract as much as possible from technical solutions. Tasks, their relationships to each other and to the actors are visualized in UC diagrams. In contrast to ordinary UC diagrams, we distinguish two kinds of task: Those tasks that are mainly influenced by the user and tasks that manly concern the system reaction. The two types of UCs are visualized in different ways. The first are shown as bubbles crossing the border between the system and the environment. The second are shown inside the border (e.g. the UC "Control windowposition partially"). 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.

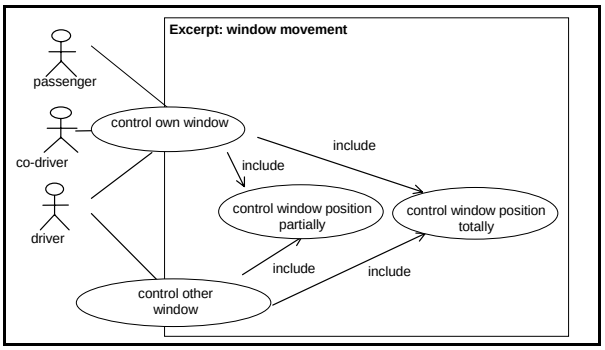

Figure 1: Use case diagram for the position window functionality

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 the interface 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 (e.g. reaching extreme values), to changing system data, problems in carrying out the system reaction, and major modes of operation. 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. In Figure 1 two UCs are added which distinguish the partial and the total movement. 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*. Table 2 shows such a template. 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.

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., exception 2.1 in Table 2),
2. boundary values of controlled variables such as limit positions
3. system behaviour outside of the UC, but visible to it (e.g., exception 2.3, safety opening in Table 2), or
4. problems in carrying out the system reaction (e.g., exception 2.2, technical problem in Table 2)

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 Table 2).

The separation of the requirements into different facets is an important prerequisite for the efficient derivation of the system specification.

Table 2: Filled UC Template for UC "Control window position totally"

Use Case	Control window position totally
Actors	Passenger, driver or co-driver
Intent	Actor positions window to the upper or lower limit
Precondition	None
Description	1. Actor inputs request for total movement up or down 2. System reacts accordingly [Exception 2.1. Actor inputs request for partial movement] [Exception 2.2. Technical problem] [Exeception 2.3. Safety Opening]
Exceptions	2.1 Partial movement => UC "Control window position partially" 2.2 Technical problem => System does not react completely 2.3. Safety Opening => System opens the window totally
Rules	The system activates the "Safety Opening" if the actor request upward window movement, but the window does not move
Mon. Var.	Window_Position, Actor_Input: movement_type (partial, total) and movement_direction (up, down)
Cont. Var.	Window_Position
Quality requ.	None
Postcondition	Window has requested position

3.2 Inspection of Use Cases

Inspections are one of the most efficient quality assurance techniques. Especially in the early life cycle phases, inspections are highly valuable, as each defect that is removed from the requirements cannot cause follow up defects in later phases. Further, the cost for detecting and removing a defect increase with each development phase [BB01]. Inspections add new views on a software artifact by involving different stakeholders in the inspection process. As discussed in

[P+04] there are many stakeholders involved in the development of complex products and it is very important to support their communication. Therefore, we integrate inspections in our approach as an analytic quality assurance technique for UCs. In particular, we use the perspective-based reading (PBR) approach [BGL96][Lai00]. The idea of PBR is that the UCs are inspected from the perspectives of the most important stakeholders. Stakeholders of the UCs are all roles in the development process that are potential users or creators of the UCs. If all these stakeholders agree on the quality of the UCs, it is probable that they are in fact of good quality. Typical stakeholders/perspectives are: (1) test engineers who use UCs as input for acceptance test planning and test case creation, (2) designers who derive high level design diagrams from UCs, (3) users who take the UC to check whether all their requirements on the system are captured, (4) maintainers who have to perform changes on the UCs (5) domain experts who take the UCs as a reference document to decide whether the final system can be build within budget, time, and whether the UCs address the state of the market, and (6) project managers who take the UCs for project planning purposes (e.g. assign working tasks for different iterations of the development).

We chose these perspectives in order to complement our UC creation guidelines. The perspectives focus on those defects that are difficult to address in a constructive way. The user and the domain expert perspectives address *incompleteness* and *incorrectness defects* in the sense that all requirements are captured in the UCs. We address *intestability defects* with the tester perspective, and *infeasibility* and *overspecification defects* with the designer perspective. The maintainer perspective addresses *inchangeability* defects and the project manager perspective analyzes the UCs with respect to the possibility to create a *feasible* project plan for further system development. Other perspectives might also be useful, but in combination with the creation guidelines the above mentioned perspectives are the most valuable ones.,

One major problem with inspections in general is that often the inspectors do not know how to search a document for defects. Often, the requirements engineers are trained in defining UCs but not in reading them for defects. To overcome this issue, the PBR approach provides reading scenarios. These scenarios provide a step-by-step description of the activities an inspector should perform when searching for defects in the UCs. For each of the identified perspectives on the UCs a tailored reading scenario is defined [Lai00]. 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. The quality concern that is addressed is related to the perspective that is assumed. An example for such an introduction would be: “Imagine you are inspecting the UCs from the perspective of a tester. One of your tasks is the definition of a test plan and the creation of test cases based on the UCs. Therefore, you are interested in the testability of the UCs.”

In the *instruction*, an inspector gets concrete guidance on **how** to perform certain activities while searching for defects. The instructions define exactly which documents an inspector should use during the inspection and how to read them. Following these instructions the inspectors get an understanding of the document and can start to perform the inspection. One important aspect of the PBR approach is that the instructions require the inspectors to create real work products relevant tfor the assumed perspective. For example, the designer perspective requires the inspector to derive high-level statecharts from the UCs. Thus, the result of the inspection is not only a list of defects, but also a certain set of state charts that can be used as a starting point for later development activities. Creating real work products makes the inspection more feasible, as the amount of additional effort is not too high (as the effort for later activities can be reduced). However, as in many cases the inspections cannot cover all UCs (especially when the system is too complex, i.e. there are too many UCs to inspect), the derived artifacts must be viewed as an initial basis for later activities. The inspections cannot substitute these activities [Lai00]

The third element of the reading scenarios are the *questions*. The questions support the inspectors in judging whether the document (i.e. the UCs) fulfills the required quality properties. The inspectors should answer the questions while following the instructions given in the scenario. In Figure 2, an excerpt of the reading scenario of the designer perspective is shown.

The motivation for providing such guidance and different perspectives for inspectors is three-fold [BL02]: First, the instructions help an inspector to gain a focused understanding of the UCs. Understanding is a necessary prerequisite for detecting more subtle defects. Detecting subtle defects is the most important aspect of any inspection, as subtle defects are often the most costly defects. By focusing the inspectors on specific aspects (with the perspectives) and give them active guidance more subtle defects can be identified with the PBR approach and in consequence, the inspection becomes more effective. 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 results in the creation of reusable artifacts. This is the most decisive aspect of this inspection technique, compared to other approaches such as checklists. Third, the attention of an inspector is focused on the information most interesting for a particular stakeholder. Thus, the inspector is not overwhelmed by the amount of

information he or she has to inspect. Further, the inspectors are focused on different quality aspects. This reduces the overlap between them and makes the inspection more efficient.

An additional advantage of this inspection technique is that it supports the communication of different stakeholders of the UCs. Thus, we address *incomprehensibility defects*, by bringing together the views of the important stakeholders. The inspections are especially valuable to make the different stakeholders more sensible for the concerns and problems of each other and to transfer/share knowledge between the involved inspectors. In case that other inspection approaches are used (e.g., checklists) it has to be analyzed which quality aspects are addressed with the checklist questions and to balance this with those quality aspects already assured with the guidelines. Further, one would loose the benefits of the PBR approach described in the last paragraph. Note that inspections must in any case be tailored to the context of the company and the project they are applied in. Inspection reading scenarios or checklists that worked for one company might not work for another company due to different context settings.

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

Note that depending on the experience level of the inspectors, different levels of details are required in the scenarios. A highly experienced designer or tester does not need a detailed description on how to derive a high level design or test cases from the use cases. On the other hand, a less experienced inspector who has less experience in performing a certain task (such as the derivation of test cases) requires a detailed description on how to do the task. It is obvious that the more experts of the different perspectives are involved in the inspection process, the more defects can be found. Experienced inspectors will in most of the cases perform better than inexperienced ones. However, the experts are those people that are in most cases not available for an inspection. Then, the reading scenarios provide a means to include also less experiences inspectors and still perform an effective and efficient inspection process.

3.3 Simulation of Requirements

The most difficult defects to detect by inspections or to prevent by creation guidelines are *inconsistency* and *interference* defects. Inexperienced inspectors often do not detect these defects or only with much effort. Therefore, we integrate simulation as a third quality assurance technique that addresses such defects more efficiently. Simulation means that UCs are “executed” in an simulation model. That is, the actor inputs are triggered on the UCs and the system reaction is observed. To apply this approach the textual UCs must first be transformed into a formal, simulatable model. Here one can use sequence diagrams. However, this only allows the simulation of individual UCs. Critical defects are often observed when it comes to the interplay of different functionalities. In consequence, a formal model is needed that allows for simulation of the complete system behavior. In the domain of embedded systems and the automotive industry statechart (SC) models are frequently used for that purpose [Har98]. Typical approaches for transforming UCs into SCs require already formal UCs in terms of sequence diagrams, e.g. [RG99]. To avoid formal sequence diagrams as an intermediate between textual UCs and SCs, we go directly from UC to SC. We perceive this process as more efficient, as we omit an intermediate step which would not provide relevant additional information for the SC creation. 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, with transitions to other states. Exceptions and preconditions are reflected in these reactions (for example in guard conditions on the transitions or as additional transitions). The main behaviour described in the SCs reflects the different steps of the UCs.

To model the formal SCs one can use tools like Rhapsody [Rha]. These tools should be able to “execute” i.e. simulate the SCs in order to allow the validation of the dynamic behavior of the system. On the one hand, this helps to detect *incorrectness* defects, as the user can directly see how the specified system behaves. Moreover, the transformation and simulation allows to check for *inconsistency* defects in the SCs and thus in the UCs regarding the dynamic behavior. Inconsistency is often detected during transformation from textual UC to the SC, when it is not possible to define a clear system reaction, for example, for certain

interaction UCs. Further inconsistencies are detected through simulation when the system does not react as expected. In particular, *interference* defects can also be detected, since it is easy to simulate several UCs concurrently by means of simulating their corresponding SCs. Further, some of these tools provide automatic checks of the SCs (e.g., consistency checks) based on formal model proofs. To optimize our combination approach, we will have to investigate the power of these automated checkers. If no tools are available, the reading scenarios of the inspection have to be adapted to cover inconsistency and interference issues.

It is important to note that transformation of the UCs into the more formal SCs takes additional effort. In addition the transformation is a manual step in which defects might be introduced. Here we perceive our guidelines as a means to help to prevent the introduction of new defects as they clearly describe the transformation steps to be performed. In case that sufficient effort is available, an inspection should be planned in addition to verify the consistency between the UCs and the SCs (i.e., to check that the transformation was done correctly). At this point in time we cannot prove the return on investment of using the simulation in a statistically significant way but we have some indications that show that the simulation of the SCs (i.e. the UCs) pays of (see section 4). The definition of good simulation scenarios, i.e. the definition of input sequences that are used to simulate the UCs, is essential.

3.4 Summarizing the Integrated Approach

By combining constructive and analytic quality assurance techniques, we are able to address all the classes of the defect classification introduced in the beginning of this article, and thus all important quality criteria. In Table 3, 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 3: 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 might 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.

The main drawback of our approach is the additional effort. The development of the formal SC and the performance of the inspections require some effort. We therefore recommend to utilize the effort in a most beneficial way, that is to identify those requirements that are most critical for the success of the system and that bear the highest risks of later losses. These requirements should then be first in the line, i.e. our quality assurance approach should be applied on these requirements. For the less important requirements it might be more efficient to use less sophisticated techniques. For example, a less relevant UC is not simulated, but only inspected from the tester perspective.
We believe that SCs with a structure close to the UC structure are better suited for requirements engineering than a structure that focuses on design optimization. We allow redundancies in the SC models and the class diagram. Thus, it is necessary to restructure the class diagram and the SCs during system design. So one has to trade-off the extra effort for the late restructuring with the ease of simulation and understanding which helps to detect defects more effectively.

Another drawback is that 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 possible by carefully identifying the stakeholders of the UCs and analysing their quality needs and the way these perspectives look for defects in the UCs.

Finally, our approach is focused on UCs and thus on initially informal requirements specifications. We do not take formal requirements specifications like SCR [HBD95] into account. As many stakeholders with different technical backgrounds are involved in the requirements engineering process, we perceive it as an inevitable necessity to start from a more informal specification such as UCs and proceed from there to more formal notations. As SCs are one of those techniques used in the automotive domain we focused our approach around this notation, omitting others like SCR.

4 Evaluation of the Approach

We evaluated several parts of our approach in case studies with students. So far, we have focused on the value of the single techniques; i.e., the UC creation guidelines and the inspection approach. For the simulation we collected qualitative statements about their usefulness. We have not yet evaluated the integration of the different techniques into a combined approach. In the following, we outline the results of the different evaluation parts.

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 lights in the rooms and floors of a university building. 12 students participated in the study. All students had limited experience with the application domain., The students had some experience in performing systematic requirements inspections. Due to teaching regulations we could not design an experiment with a control group not using the guidelines.. The control group would have used an ad-hoc approach. This would have contradicted the teaching goals of the practical course that all students should learn adequate techniques. 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 first step towards a more thorough evaluation.

The software used in the practical course is a reactive system for house automation that was created for and evolved within the course. The system was divided into three subsystems. (1) The graphical user interface (GUI) that offers an interface to control the system. (2) The light control system (Light) that switches lights on and off depending on the presence of people in a room and a floor. (3) The temperature control system (Temp) that controls the room temperature, depending on the presence of people in a room and the current daytime.

A group of 4 people was responsible for the development of each sub-system. The Temp system comprised 21 UCs and the related textual descriptions (scenarios), the Light system comprised 15 UCs and scenarios and the GUI system 34 UCs and scenarios. Based on the problem description, each group had to develop UCs for its sub-system with our UC creation guidelines. To evaluate the usefulness of the guidelines, we used a questionnaire that was given to the students after they completed the UC creation step. It was designed following the model recommended by Davis [Dav89]. Initially the model was used to evaluate the usefulness of a tool in supporting certain tasks. Laitenberger indicated in a study that this model can also be used to evaluate the usefulness of software engineering techniques, after some tailoring of the questionnaire. The basics of the model are three 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). Based on the student rates it is possible to evaluate the degree of agreement to the questions in each category (build the median of all student answers of the question of this category) and then judge the overall usefulness of the technique by interpreting the medians of the three categories.

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. Again we perceived improvement potential for this category. 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. Only two subjects would not use the guidelines again.

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 a controlled case study. 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

The subjects had to use different reading techniques: perspective based reading and a checklist based inspection approach. In the case of PBR, three perspectives were used as explained in section 3: a tester perspective, a designer perspective and a customer/user perspective. The inspector assuming the tester perspective had to derive a set of test cases. The designer perspective had to derive initial SCs, and the user perspective had to create use cases from the problem description and compare these to the use cases under inspection.

To investigate the influence of the perspective based reading approach, we created focused checklists that were comparable to the reading scenarios (this is called CBR in the following). We defined three checklists, each representing one of the perspectives, using similar or the same questions as the corresponding reading scenario. Thus, the focus of each checklist was the same as the focus of the related reading scenario and the checklist provided the same separation of concerns.
We chose a partial factorial design (see Table 4) in which each group participated in two inspections (Run 1 and 2) using some combination of the reading technique (focused CBR and PBR) and inspected artifact (the three subsystems). The inspection teams were similar to the teams that build the different sub-systems (e.g. Group_Light build the sub-system Light and therefore reviewed the other two sub-systems). Thus, in each inspection team 4 students were included. Each sub-system was reviewed by two groups (8 subjects), one group using CBR (4 subjects) and one group using PBR (also 4 subjects).

In each run, the inspection teams used different techniques and different perspectives. For example, the team that used CBR in the first run used PBR in the second run and vice versa. In addition a subject that used the tester perspective in the first run used the designer or the customer perspective in the second run.

Table 4: Design of the controlled case study

	PBR	Focused CBR
Run 1	Group_Light inspects Sys_Temp	Group_GUI inspects Sys_Temp Group_Temp inspects Sys_Light
Run 2	Group_Temp inspects Sys_GUI Group_GUI inspects Sys_Light	Group_Light inspects Sys_GUI

In addition to these questions, 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: For the Temp and GUI subsystems, the teams using PBR, respectively, found 23% and 40% more defects than the teams using focused CBR. However, for the subsystem Light the team using focused CBR found 32% more defects than the team using PBR (see following table).

Table 5: Effectiveness at the team level

	% of defects found by CBR-team	% of defects found by PBR-team
Temp	50%	65%
GUI	44%	73%
Light	83%	57%

Regarding the efficiency (measured in defects found per hour) the study showed that the checklist-based approach is more efficient for the Temp and the Light subsystems (by 27% and 52%, respectively). For the GUI subsystem, the PBR team was 70% more efficient than the CBR team (see following table)

Table 6: Efficiency of the approaches

	Efficiency of CBR-team	Efficiency of PBR-team
Temp	1.93	1.52
GUI	1.14	1.94
Light	2.00	1.31

In summary, the results of our controlled case study do not follow a common pattern. Our analysis showed that the PBR approach improved the effectiveness of defect detection in the GUI system but not in the Light and the Temp system. Our results do not appear conclusive mainly because the very small sample size of the study (12 inspectors organized in 3 teams with 4 inspectors each) hampers running reliable tests for differences. However, there are some hints that the complexity of the subsystems can explain the discrepancy in the results: One possible interpretation for this is that the effort spent 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 too high as they were not that complex, i.e., the checklist was sufficient to gain the understanding of the system. Only in complex documents a systematic approach to gain a profound understanding of the inspected artifact pays off. However, this is a hypothesis that needs to be validated in future research activities. With the similar argumentation we interpret the not conclusive results regarding the efficiency of the two approaches.

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. More detailed results of the inspection case studies can be found in [DCL04].

There are several threats to validity that should be considered with respect to the presented results. Threats to validity influence the outcome of an empirical study (they have an impact on the dependable variables, in our case the number of defects found and the time needed to find them). First there are learning effects, i.e. the students get to know how to perform inspections and perform better in the second run of the case study. We controlled this thread by assigning different reading techniques in the different runs and in addition we varied the perspective the inspectors use in the two runs. For example an inspector that assumed the tester perspective in the PBR inspection can only use the designer or user/customer checklist in the second run. Moreover there are threats that the student did not follow the checklist or the PBR approach while doing the inspection. We controlled this threat by briefing the students after the case study. The results indicate that the students followed the different approaches. To prevent selection effects we chose the students randomly to certain perspectives. In addition we ensured the quality of the checklists and the reading scenarios of the PBR approach by reviewing them by inspection experts. This prevents that the case study results are biased due to poorly designed checklists and reading scenarios. Finally, the usage of students in as subjects in a class experiment is a threat to external validity (i.e., the generalizability of the results). It has been shown that the differences between graduate students and professionals is often not very large [Pre01]. Moreover, according to [Run03] the difference between graduate students and professional software developers is small compared to freshman and graduate students. The UCs that had to be inspected represented UCs of a real system and the number of the designed UCs and related scenarios shows that this is not a pure toy-example. We therefore believe that the results still can be interpreted in a broader sense.

In addition, 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. Some additional findings with respect to the simulation approach can also be found in [DO04].

Again we want to note that the presented results are 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. Therefore, these studies have to be followed with industrial case studies to achieve more generalizability. However, performing initial studies with students is a necessary and important step in the empirical evaluation of any technology. This is due to the fact that a sound case has to be built in order to justify and motivate studies with professional, and therefore more expensive, developers [SCT01]. 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 in the requirements engineering phase. 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.

Acknowledgements

We thank our project partners at Fraunhofer FIRST and colleagues at DaimlerChrysler for fruitful discussions. This approach was developed in the QUASAR project supported by the BMBF under the label VFG0004A.

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

[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, vol. 1, no. 2, 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

[BRJ99] Booch, G.; Rumbaugh, J.; Jacobson, I.; The Unified Modelling Language User Guide; Addison-Wesley, 1999

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

[Chi92] Chillarege, R; Bhandari, I; Chaar, J; Halliday, M; Moebus, D; Ray, B; Wong, M; Or-thogonal defect classification -- A concept for in-process measurements, IEEE Transactions on Software Engineering, vol. 18, pp. 943--956, Nov. 1992

[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; 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

[DO04] Denger, C; Olsson, T; Simulating Textual Scenarios using State charts, 3rd International Workshop on Scenarios and State Machines. Models, Algorithms, and Tools. SCESM04. (2004), 21-26 : Ill., Lit.

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

[Frei01] Freimut, B., Developing and Using Defect Classification Schemes, IESE Report No. 072.01/E, 2001

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

[Gra92] Grady, R; Practical Software Metrics for Project Management and Process Improvement. Prentice Hall, 1992.

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

[HBD95] Heitmeyer, C.; Labaw, B.; Kiskis, D.; Consistency Checking of SCR-Style Requirements Specifications, in Proceedings of the Second International Symposium on Requirements Engineering,March, 1995.

[IEEE94] IEEE Standard Classification for Software Anomalies, IEEE Std. 1044-1993, 1994

[IEEE98] 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), pp. 174-183, 1999

[Mel92] Mellor, P; Failures, faults and changes in dependability measurement, Information and Software Technology, vol. 34, pp. 640--654, Oct. 1992.

[P+04] Paech, B: Denger, Ch.; Kerkow, D., von Knethen, A.; Requirements engineering for technical products – integrating specification, validation and change management, to appear in Silva A. (ed.) Requirements Engineering for socio-technical systems,

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

[Por96] Porter, A; Votta, L; Basili,V; Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment, IEEE Transactions on Software Engineering, 1996, pp 563-575.

[Pre01] Prechelt, L.; Kontrollierte Experimente in der Softwaretechnik, Springer 2001 (in German).

[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

[Run03] Runeson, P.; Using Students as Experiment Subjects – An Analysis obn Graduate and Freshmen Student Data, Proceedings 7th International Conference on Empirical Assessment and Evalutation in Software Engineering, 2003.

[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

[SCT01] Shull, F.; Carver, J.; Travassos, G.; An Empirical Methodology for Introducing Software Processes, Proceedings 8th European Software Engineering Conference, pp.288-296, 2001.