

Author 1, Felix Hausberger, Max Edinger, Andre Meyering

Title

Technical Report

Advisor University of Heidelberg
Prof. Dr. Barbara Paech, Astrid Rohmann

mm dd, year

Abstract

Place the abstract in this section.

Contents

1	Introduction	5
1.1	Systematic Software Testing	5
1.2	Common Fundamentals	6
1.3	Outline	7
2	Systematic Generation of acceptance tests that are executable with FitNesse	10
2.1	Introduction	10
2.2	Literature search	11
2.3	Approach 1: <i>Developing comprehensive acceptance tests from use cases and robustness diagrams</i>	12
2.3.1	Description	12
2.3.2	Application	14
2.4	Approach 2: <i>A Web Framework For Test Automation, User Scenarios Through User Interaction Diagrams</i>	18
2.4.1	Description	18
2.4.2	Application	19
2.5	Comparison	21
2.6	Conclusion	25
3	Testing with a transition system	26
4	Testing with a timing component	27
5	Testing with a classification tree	28
6	Testing with a formal specification	29
7	Testing with system models	30
8	Testing functional and nonfunctional requirements in User Requirements Notation	31
9	Testing Non-Functional Requirements with Risk Analysis	32
9.1	Introduction	32
9.2	Literature Search	32
9.3	Approach 1	34
9.3.1	Description of Approach 1	34
9.3.2	Application of Approach 1	36
9.4	Approach 2	37
9.4.1	Description of Approach 2	37
9.4.2	Application of Approach 2	39
9.5	Comparison	42
9.6	Conclusion	44
10	Testing nonfunctional requirements with aspects	46
11	Conclusion	47

1 Introduction

1.1 Systematic Software Testing

With the rise of smart gadgets and the Internet of Things, more and more parts of our daily life involve technology. And the software required to run our smartphones, computers and other gadgets requires software that becomes more complex as more data can be processed. As software becomes more complex, bugs can have immense consequences as the recent data breaches have shown ¹.

Therefore, proper testing of software becomes more important than ever. But what is software testing? According to the “Guide to the Software Engineering Body of Knowledge” (short: SWEBOK), “Software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain.” [3] This means that it may not even be possible to test every input against the expected output, for example if the input is an infinite data stream. But also for functionality with a finite set of input data, testing every possible combination may not be feasible. Software testing can take more time than developing the program under test. Hence, software testing can be tedious, difficult and expensive. So the question arises how software can be tested in an effective and efficient way.

We need to write tests in a systematic way. This paper contains 9 articles, each describing another approach to systematic software testing.

In chapter 2 we focus on acceptance tests with FitNesse. Communicating requirements between customers and developers can be difficult. Whereas natural language can be too ambiguous, code can be too technical. By using FitNesse, human-readable Fit-Tables can be created that be understood by customers and can be transformed into code automatically.

But having tests may not be enough. Which requirement is covered by which test? Can we be sure that the implementation matches the specification? Traceability becomes necessary and is required to not lose overview over the test cases and covered requirements. This is where chapter 3 comes into play with transition systems which can be used to automatically create test paths through the application by using formal specifications.

After that, chapter 4 looks into testing with a timing component. Integrating different components can reveal errors even if the individual components are properly tested. Furthermore, if the system is a real time system, tests with the same input can lead to different system reactions if the test’s execution time changes.

Following this, chapter 5 looks into decreasing the number of test cases by using classification trees. Because testing all possible parameter cases becomes unmanageable very fast, these classification trees offer a great way to reduce the complexity of such parameter constructs.

¹TODO

Furthermore, these classification trees can be used to generate test cases.

chapter 6 *ist noch nicht online; TODO*

In chapter 7 we then look into testing with systems models which are compared against test models. It is looked into model based testing and how tracability can be used to increase the probability of finding errors and improve test quality.

chapter 8 explains why writing manual tests for functional and non-functional requirements can be not only tedious but error prone due to Copy&Paste of errors in test logic. The section describes how test cases can be automatically created by using the user requirements notation that is used for modelling, analyzing, and controlling the correctness and completeness of functional and non-functional requirements.

Another aspect of testing non-functional requirements can be by using risk analysis. This is where chapter 9 steps in and briefly shows how non-functional requirements can be tested and how risk analysis can be used to prioritize test cases.

The section also shows how architectural non-functional requirements such as code conventions can be tested. chapter 10 expands on this topic by introducing aspects and aspect oriented programming to test and verify non-functional requirements such as software memory limits and memory leaks.

Finally, chapter 11 concludes this paper and summarizes each article.

1.2 Common Fundamentals

All articles build upon a common set of fundamental definitions regarding software testing and requirements. They are listed and mapped to the individual topics in the glossary section of this report. It is highly recommended to refer to the glossary before reading an article or when ambiguities arise while reading an article. However, since it is quite extensive and also contains more topic-specific definitions, the most important terms are defined hereafter.

Like a part of the entries in the glossary section, we use the SWEBOK in its third version as a basis. Although this guide is already quite old (at least the original version from 2004), hence not fully compliant with current research results, its 15 knowledge areas and basic definitions of a body of knowledge are still useful for classifying the approaches presented in this report and establish a common set of technical terms.

Let us begin with requirements. The corresponding SWEBOK knowledge area is “Software Requirements”. Relevant sub chapters are “Software Requirements Fundamentals” and “Requirements Validation”. The guide defines requirements as “a property that must be exhibited by something in order to solve some problem in the real world. [...] An essential property of all software requirements is that they be verifiable as an individual feature as a functional requirement or at the system level as a non-functional requirement. It may be difficult or costly to verify certain software requirements.” [3] This definition already points out the difficulty of verifying requirements that necessitates the use of systematic testing techniques, as explained in section 1.2. Moreover, it distinguishes between functional, representing a feature the software is to provide, and non-functional requirements, specifying the extend of quality. Requirements need to be formulated clearly, unambiguously and quantitatively in order to implement and verify them correctly [3].

For software testing, there is no uniform definition. Therefore, the guide refers to multiple definitions from cited references. In essence, software testing is to assure that specified requirements are met by the implementation or, from a different perspective, find errors indicating that a requirement has not been met. This testing process is performed at different levels, as the requirements definition already touched upon. The SWEBOK guide distinguishes between three test levels: unit testing, verifying isolated functionalities (mostly functional requirements), integration testing, verifying the correct interaction of components and system testing, verifying the behavior of the entire system (mostly non-functional requirements). Correspondingly, these levels are distinguished by the object of the test (single module, multiple modules, entire system), called the target of the test, and the purpose, called objective of the test [3].

The guide presents a wide array of testing techniques. For this report, it is important to take notice of the definition of model-based testing: “A model in this context is an abstract (formal) representation of the software under test or of its software requirements (see Modeling in the Software Engineering Models and Methods KA). Model-based testing is used to validate requirements, check their consistency, and generate test cases focused on the behavioral aspects of the software.”[3] Some of the approaches presented in this report are model based, at least partially. However, it is not always clear what the actual model is and some authors use the term incorrectly.

Finally, it is important to emphasize the difference between different artifacts produced during the testing process, including the afore-mentioned test objectives, test cases (logical or concrete with inputs) as well as executable tests (as code).

1.3 Outline

Following this introduction in chapter 1, nine individual reports each present two different but related approaches for systematic testing in Sections 2-10. The reports introduce their superordinate topic in Sections X.1, outline the results and execution of a literature search based on a given article in Sections X.2 and describe the given and selected approach in Sections X.3.1 and X.4.1 as well as illustrating them using a common set of given requirements in Sections X.3.2 and X.4.2. Finally, the approaches are compared using a common set of questions in Sections X.5 and evaluated in Sections X.6. Section 11 concludes the report. The glossary and bibliography can be found in Sections 12 and 13. In the following, the given requirements (Figure 1.1) and synthesis questions, used for each individual report, are depicted.

Synthesis questions:

1. What is the name of the approach? If no name is provided, the publication title is used.
2. Summary
3. Description of the approach (What does the approach do?)
 - a) Which artifacts and relations between artifacts are used in this approach? Which artifacts are created in the course of the approach? How are the artifacts characterized?
 - b) What is required and/or input for the application of the approach?

- c) What steps does the approach consist of? Which information is used in which step and how? What are the results of the individual steps?
- 4. Benefits of the approach (Whom does the approach help and how?)
 - a) Which usage scenarios are supported by the approach?
 - b) Which stakeholders are supported by the usage scenarios?
 - c) Which knowledge areas from SWEBOK can be assigned to the usage scenarios?
- 5. Tool support for the approach (What tool support is available?)
 - a) What tool support is provided for the approach?
 - b) Which steps of the approach are automated by a tool? Which steps are supported by a tool, but still have to be executed manually? Which steps are not supported by a tool?
- 6. Quality of the approach (How well does the approach work?)
 - a) How was the approach evaluated?
 - b) What are the (main) results of the evaluation?

User Task		Movie Management
Purpose (Goal)		Users manage movies and corresponding performer data of a movie collection.
Frequency		Often and at any time (depending on the user's needs)
Actors		User who wants to manage movies
Sub-tasks:		Example of solution:
1	Describe a movie Add and describe a movie with typical data like its title or alternative titles, release date, production country, release date, runtime, location, IMDB ID, and performers. Or change an existing description. Or view movies (possibly sorted).	Provide default values wherever possible, implemented in <i>create movie</i> , <i>change detail movie data</i> , <i>link existing performer</i> , <i>unlink performer</i> , <i>show movie</i> , <i>list movies</i> , <i>search</i> , <i>sort movies</i> , <i>show performer details</i> and <i>show movie in IMDB</i>
1ap	Remove movie Problems: Removing all movies a certain performer participates in might result in performers associated with no movies.	Ensure the consistency of performers and movies, implemented in function <i>remove movie</i>
2	Describe a performer Add and describe a performer featuring in movies with typical data like first and last or alternative names, biography, country, IMDB ID and date of birth. Or change an existing description. Or view performers (possibly sorted).	Provide default values wherever possible, implemented in <i>create linked performer</i> , <i>change detail performer data</i> , <i>link existing movie</i> , <i>unlink movie</i> , <i>list performers</i> , <i>sort performers</i> , <i>search</i> , <i>show performer</i> , <i>show movie details</i> and <i>show performer in IMDB</i>
2ap	Relate performer to movie Problems: Creating a performer without relating her/him to a movie lead to performers associated with no movies. Thus, a performer needs to be related to a movie.	Ensure that performers are linked to movies on creation, implemented in function <i>create linked performer</i>
2b	Remove performer Removes a performer.	Implemented in function <i>remove performer</i>
3	Manage watched Movies Record date when the movie was last watched.	Implemented in function <i>watch movie</i>
4	Rate Movie or Performer Rate or view ratings (sorted).	Provide a fixed rating value list. Implemented in functions <i>rate movie</i> , <i>rate performer</i> , <i>calculate overall rating of movie</i> and <i>sort movie</i>

Figure 1.1: Requirements in User-Task notation for the MovieManager software, a mobile application for managing movie collections.

2 Systematic Generation of acceptance tests that are executable with FitNesse

2.1 Introduction

This chapter focusses on the creation of acceptance tests that are automatically executable using the tool *FitNesse*. In this first section of the chapter the reason for using this approach is discussed. Furthermore, the general features of FitNesse as well as the needed artefacts like Fit-tables are explained. An article that focuses on the topic of creating acceptance tests that are executable with FitNesse was provided by the supervisors of the seminar. Section 2.2 documents the literature search used to find a different approach. The two articles are then described in the sections 2.3 and 2.4. Moreover, for a better understanding of the presented approaches these chapters include the execution of them on the *MovieManager*. The following section 2.5 includes the comparison of the two approaches using a synthesis matrix. Section 11 provides a Conclusion including the most important insights of the process and an assessment in which situations the approaches might be suitable.

During the software engineering process communication between the developers and the customers is a crucial factor for the success of the product. A problem for the communication is the different use of documents by the two main stakeholders: Customers describe their requirements in natural language whereas the developers create code. Natural language can often be interpreted in different ways, which can lead to unwanted results. And whereas code is more precise, it is often too technical for the customer. Therefore, artefacts are needed that are more precise than natural language and can easily be transformed into code.

One such artefact is a *Fit-table*. These tables store easily readable information about acceptance test cases and can be fully automatically executed using the testing tool *FitNesse* [12]. Creating *Fit-tables* before the development of the software can help the developers to understand the requirements of the customer by implementing the necessary functionality to pass the acceptance tests. Thus the customers receive a software that satisfies all their mentioned requirements. *FitNesse* supports the creation and maintenance of *Fit-tables* as well as the automated execution of the tests represented by the tables. To make this possible *Fixture-Classes* are needed. These classes connect the input values from the *Fit-tables* to the *System-under-test* and are executed by *FitNesse*. An overview of the data exchange during the process is shown in figure 2.1 on the next page. The specific steps are explained in the following:

After the user chooses to execute a set of Fit-tables in FitNesse, FitNesse executes the Fixture-Classes that belong to the selected tables. These tables can contain two types of values: Input values and expected output values. The *Fixture-Class* creates an instance of the *System-under-test* and then transfers the input values into it. Then it extracts the resulting output values from the *System-under-test* and returns them to *FitNesse*. These extracted output values are then automatically compared by *FitNesse* to the expected output values from the Fit-tables. If they are the same, the test was successful and the entry of the table receives the colour *Green*. Otherwise, the affected part of the test failed and the entry receives the colour *Red*.

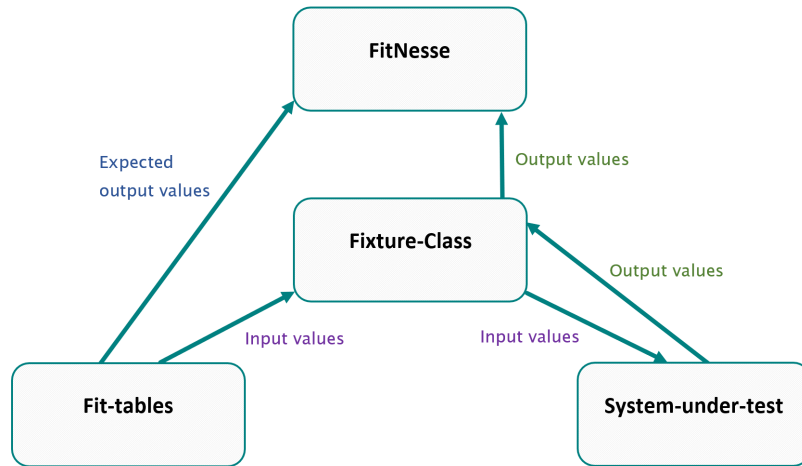


Figure 2.1: Overview of the data exchange in the execution of *Fit-tables* with *FitNesse*. The *Fit-tables* can be created and maintained in *FitNesse*.

2.2 Literature search

The literature search consisted of a search-term-based search as well as forward- and backward-snowballing. As start article the work of El-Attar and Smith [8] was given by the supervisor of this seminar. This article presents an approach to create acceptance tests that can be automatically executed using *FitNesse*. To find more and possibly different approaches the search question was chosen to be:

Which approaches to systematically generate acceptance tests that are executable with FitNesse exist in the literature?

ACM Digital Library [11], *IEEE Xplore* [13], *Springer Link* [14] and *Science Direct* [15] were used as search platforms as they are the most common platforms in the field of computer science. The relevance criteria were chosen as follows:

- *Criterion 1:* The article describes an approach to systematically generate acceptance tests that are executable with *FitNesse* or gives an overview on the use of *FitNesse* in software engineering.

This criterion was chosen to find approaches that are specific to the subject of this chapter. Articles that give an overview over the use of *FitNesse* were also accepted because of their potential to classify the found approaches.

- *Criterion 2:* The article was not published before the year 2009 which is the year that the article by El-Attar and Smith was published.

This criterium was chosen to get a more recent approach than the start article which was by the creation of this chapter already more than 10 years old.

Table 2.1 on the next page provides an overview for the search-term-based literature search. As search terms the terms 'acceptance test' and 'FitNesse' were chosen. These search terms turned

out to be specific enough to fit only a manageable amount of articles. The search resulted in eight relevant articles of which six presented an approach and two gave an overview over the use of *FitNesse*. Both snowballing searches from the start articles did not result in any more relevant articles that were not already found by the search-term-based search. The backward-snowballing did not result in any relevant articles due to their publishing date and hence not passing Criterion 2. One relevant article was found during the forward-snowballing that was already found by the search-term-based search on the platform Springer Link.

Table 2.1: Overview of the search-term-based literature search.

Search platform	Search date	Search restrictions	Search terms	# results	# relevant results	# relevant new results	Used articles
ACM	18.11.20	Publishing year: 2009-2020	„acceptance test“ AND fitnessse	10	4	4	Longo et al., 2016
IEEE Xplore	18.11.20	Publishing year: 2009-2020	„acceptance test“ AND fitnessse	2	1	1	
Springer Link	19.11.20	Publishing year: 2009-2020, no preview-only content	„acceptance test“ AND fitnessse	11	2	1	
Science Direct	19.11.20	Publishing year: 2009-2020	„acceptance test“ AND fitnessse	9	2	2	

The articles that gave an overview over the use of *FitNesse* were not specific about any approaches and only provided general information. Therefore, none of these articles was used. As a second approach to compare to the start article, the work by Longo et al. [9] was chosen. This article also describes an approach to create acceptance tests that can be automatically executed by *FitNesse*. The presented approach differs from the approach of the article by El-Attar and Smith in its use of artefacts. Also it was created by different authors and was published in 2016, so it is a much more recent approach.

2.3 Approach 1: Developing comprehensive acceptance tests from use cases and robustness diagrams

2.3.1 Description

El-Attar and Smith [8] introduce an approach to create acceptance tests that can be automatically executed using *FitNesse*. Their approach is targeted at larger software projects that use a model-based approach such as the use of UML models. It was created such that a non-technical person (e.g. a Business Analyst) can execute it during the early phases of the development of the software. This makes it possible for the developers to follow the approach of Acceptance-Test-Driven-Development (ATDD) because of the possibility of executing the acceptance tests at any time during the development process. ATDD helps the developers during the development process to evaluate which requirements are already implemented and which are yet to be implemented.

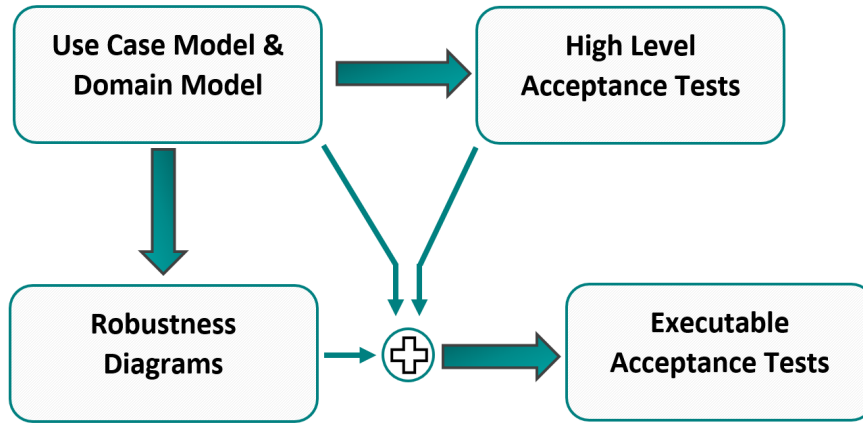


Figure 2.2: Overview of the steps in the approach of El-Attar and Smith.

The approach starts with use case models and domain models as initial artefacts. During the whole process of creating the acceptance tests every step is performed completely manually. The execution of the final acceptance tests is done fully automatically by the tool *FitNesse*. To help with traceability during the approach the authors created a tool called UCAT. This tool does not provide any automation support but allows the user to create use case models and *Fit-tables*. These artefacts can be linked within UCAT which helps to determine which use cases resulted in which acceptance tests. Figure 2.2 shows the rough structure of the approach. The exact steps of the approach are described in the following:

In the first step of the approach **High-Level-Acceptance-Tests (HLATs)** are created. For this a domain model and a Use Case model with Use case descriptions is required in the approach. HLATs are more informal than an executable acceptance test which helps the analyst to be as flexible as possible in describing the acceptance tests at this early stage. Commonly Use Cases contain Use Case descriptions from which the flows of the Use Case can be extracted. HLATs describe the system's expected behaviour during all of the flows of the use cases from the Use Case model. Necessary Pre-Conditions and triggers for the flows are also extracted from the Use Case description while the inputs for the flows can be extracted from the domain model. Expected test results are also denoted in the HLATs. At this point they do not need to contain specific values and can be written in natural language. The general structure of a HLAT is shown in Table 1.2.

Table 2.2: General structure of a HLAT.

Test ID	Description	Expected Result
Name of the Use Case & the flow	Preconditions: Inputs:	Expected result in natural language

After the creation of the HLATs a robustness analysis is performed. To achieve this, for every use case a **robustness diagram** is created. These diagrams combine the use cases from the use case model with the objects from the domain model. They contain actors and entities as well as boundary- and control-objects. For each use case all involved objects and the

connections between the objects are displayed. The involved objects and the communication between them is extracted from the Use Case description. During the creation of the robustness diagrams necessary *objects* or *attributes of objects* may be identified that are not already part of the domain model. These should be added to the domain model. Also missing steps or preconditions in the Use Case description might be found. If this is the case, the Use Case descriptions should also be updated. After this step the HLATs should be adapted to fit the updated domain model and Use Case descriptions.

In the last step all the existing artefacts (possibly except the domain models) are used to create the final product of the approach: **Executable Acceptance Tests (EATs)**. These acceptance tests are in the form of specific *Fit-tables*. To achieve this, the HLATs have to be divided into smaller steps using the information about the Usage Scenario from the related use case description. This step requires human judgment and is not further described. For each of these steps the control flow in the robustness diagram gets traced. In this process the objects and attributes of each step's input, preconditions, outputs and postconditions are determined. These were before stated in natural language in the HLAT and are exchanged in the EAT with more concrete objects and attributes. The steps combined with the corresponding control flow are manually converted into *Fit-tables*. *Fit-tables* used in this approach are either *ActionFixtures*, *RowFixtures* or *ColumnFixtures* [16]. These types of *Fit-tables* can be fully automatically executed using the tool *FitNesse*. The domain models are ideally not required if the steps before were executed properly because the information from the domain models should already be part of the use case descriptions.

Due to the fact that the approach of creating the acceptance tests is done completely manually, the quality of the resulting acceptance tests depends highly on the experience and skills of the person executing the approach. Therefore, the authors state that an evaluation would be beyond the limitations of their work. However, they provide a case example by applying the approach to the software *RestoMapper*. This example is not part of this work because in the following section the execution of the approach is presented with the application *Movie Manager* that is used throughout this whole report.

2.3.2 Application

The approach starts with Use Case and Domain Models. As those are not already described in this article, they are created for this chapter. Figure 2.3 on the next page shows the Use Case Model for the Movie Manager application. It contains the use cases and shows connections between them. For example, removing a movie might result in removing a performer if one of the performers that participated in the movie has no movies anymore after the removal. Such a relation is highlighted in the Use Case Model with the keyword *extend*. The domain model of the Movie Manager application is shown in Figure 2.4 on the next page. It contains the entities Movie and Performer as well as the two views that the user can see.

To illustrate the approach the Use Case *Describe a performer* is used. In the first step of the approach the HLATs for this Use Case need to be created. The User Scenarios for this Use Case are as mentioned in the User Task table:

- Add and describe a performer
- Change the description of an existing performer
- View performers (possibly sorted)

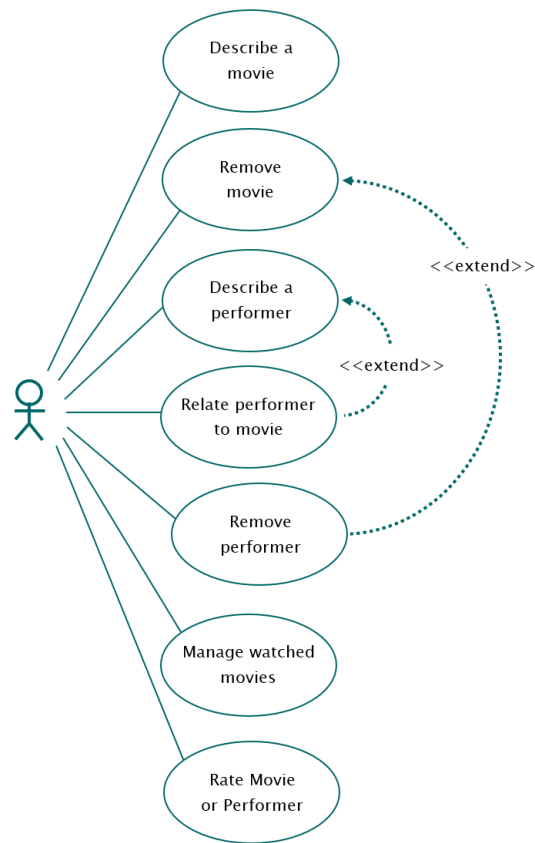


Figure 2.3: Use Case Model for the Movie Manager application.

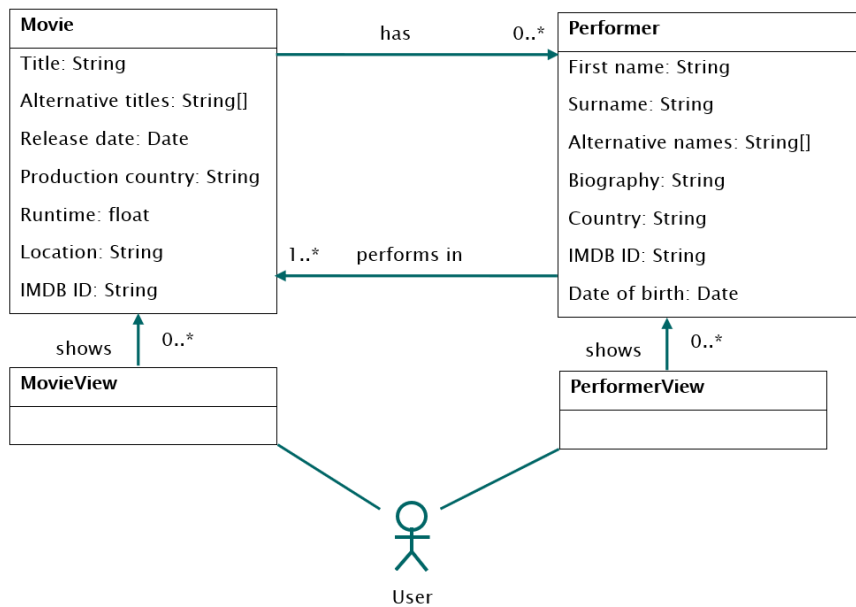


Figure 2.4: Domain Model for the Movie Manager application.

The HLATs for the Use Case *Describe a Performer* are displayed in Table 2.3. Each HLAT describes the necessary preconditions, inputs and the expected results of one User Scenario. This information is extracted from Use Case description.

Table 2.3: HLATs for the Use Case *Describe a performer* of the Movie Manager application.

Test ID	Description	Expected result
Describe-a-performer-new-performer	Precondition: a movie exists Input: values for the attributes of the performer Input: movie that the performer is linked to	The performer is created with the given attribute values. The performer can be selected. The new attribute values can be seen when the performer is selected. The performer is linked to the given movie.
Describe-a-performer-existing-performer	Precondition: a performer exists Input: performer Input: values for the attributes of the performer	The attributes of the selected performer are updated to the provided values. The new attribute values can be seen when the performer is selected.
Describe-a-performer-view-performer	Precondition: – Input: performer list Input: sorted or not	The performer list is shown. Performers are sorted if the user chooses this option.

In the next step a robustness diagram is created using the information from the Use Case Model and the Domain Model. The robustness diagram for the Use Case *Describe a movie* is shown in figure 2.5 on the next page. A robustness diagram contains the involved objects that communicate with the user. These are called boundary objects. An example for a boundary object is *PerformerView* in figure 2.5. It also contains control-objects like *RelateToMovie* in figure 2.5 that makes sure that a performer is related to at least one movie. The last types of objects are the *User* and the entities like *Performer* or *Movie* in figure 2.5. The resulting robustness diagram is used to find new information for the domain diagram. For example, *CheckMovieRelations* needs to find out whether the performer is linked to at least one existing movie. Therefore, the domain model needs to include a list of related movies for each performer or the number of related movies. In this example the first variant (a list of related movies) is used in the domain model. So this specific information does not have to be added. Overall the robustness analysis does not bring up any new information but possibly could for other examples which is why El-Attar and Smith have included it in their approach.

As last step executable acceptance tests are created for each HLAT. These are created in the form of *Fit-tables*. For this example so called *ActionFixtures* are chosen but *RowFixtures* and *ColumnFixtures* are also possible for this approach. *ActionFixtures* contain a Test ID in the first row. Each of the following rows contains one action like entering a value or pressing a button. The ActionFixtures for the three HLATs from table 2.3 are displayed in the tables 2.4, 2.5 and 2.6 on the next page. These Fit-tables are the final result of the approach.

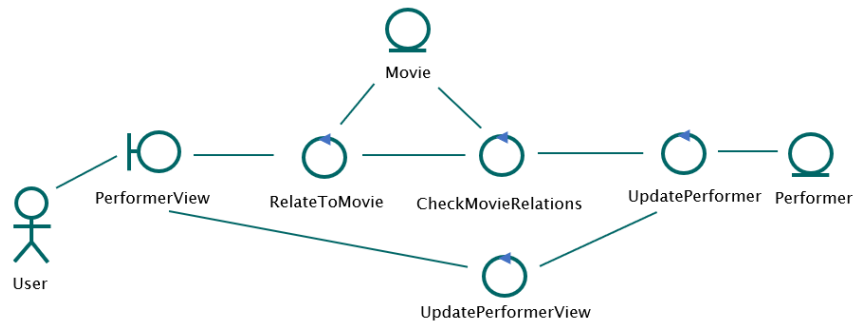


Figure 2.5: Robustness Diagram for the Use Case *Describe a performer* of the Movie Manager application.

Table 2.4: Executable Acceptance Tests for the scenario *Describe a performer, new performer* of the Movie Manager application in form of an *ActionFixture*. A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.

Describe-a-performer-new-performer		
Press	createNewPerformer	-
Enter	FirstName	f
Enter	Surname	s
...	...	
Enter	Biography	b
Check	Performer p with p.firstName == f and p.surname == s exists	
...	...	
Check	p.Biography == b	

Table 2.5: Executable Acceptance Tests for the scenario *Describe a performer, existing performer* of the Movie Manager application in form of an *ActionFixture*. A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.

Describe-a-performer-existing-performer		
Press	selectPerformer	p
Press	changeAttributes	p
Enter	firstName	f
...	...	
Enter	Biography	b
Check	p.firstName == f	
...	...	
Check	p.surname == s	

Table 2.6: Executable Acceptance Tests for the scenario *Describe a performer, view performers* of the Movie Manager application in form of an *ActionFixture*. A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.

Describe-a-performer-view-performers		
Press	viewPerformers	performerList
Enter	sorted	True or false
Check	performerView.performers == performerList	
Check	If sorted == True: isSorted(performerView.performers) == True	

2.4 Approach 2: A Web Framework For Test Automation, User Scenarios Through User Interaction Diagrams

2.4.1 Description

Longo et al. [9] create User Scenarios through User Interaction Diagrams (US-UIDs) which then are fully automatically converted into *Fit-tables* that represent the test data for acceptance tests. To run these acceptance tests a Fixture-Class is needed that connects the test data from the *Fit-table* with the System-under-test. The US-UIDs are created in a tool provided by the authors. They contain functional data such as the involved objects, attributes and functions and also explicit User Scenarios provided by the customer. The User Scenarios provide the test data and combined with the functional data, *Fit-tables* can be automatically created. The functional data represents the top row of the *Fit-table* and the User Scenarios the specific values. Figure 2.6 provides an overview over the steps of the approach. The only step that is executed automatically is marked in this overview. Each of the steps is explained in more detail in the following.

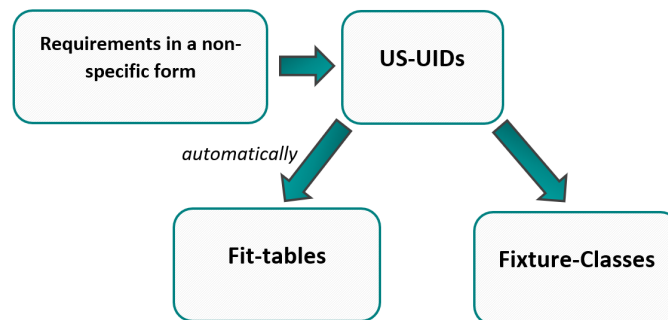


Figure 2.6: Overview of the steps in the approach of Longo et al.

In the first step the US-UIDs are created. This step is described in another work by the au-

thors [10]. Each US-UID contains an explicit User Scenario provided by the customer. The information about the User Scenario is extended by the developer by adding functional information. For example, the User Scenario could provide a specific value for a variable. Then the functional information for this value would be the name of the variable. With this conjunction of explicit and functional values *Fit-tables* can be automatically created. The first row contains the functional information. Each row after the first row represents a User Scenario and contains the given values for the functional information (objects, variables, etc.) given in the first row.

Each row can be used as one specific test case. To execute these test cases a Fixture-Class is needed. This class has to be written by the developers. It creates an instance of the System-under-test and uses *Setter methods* to provide the input from the *Fit-table* to the System-under-test. Through *Getter methods* the resulting values of the System-under-test can be extracted to validate the success of the test. For the evaluation step the results from the Getter methods are compared to the expected values from the *Fit-table*. If they are the same, the test was successful.

To evaluate their approach Longo et al. used their tool to automatically create executable acceptance tests from existing US-UIDs. The developers of the software related to these US-UIDs already manually created test cases for the software. In the evaluation the authors compared these manually created tests to the tests created by their tool. To compare both of them the authors used the techniques *code mutation* and *lack of code*. The technique *code mutation* involved manipulating the values of an array in the software and *lack of code* was executed by removing a class from the software. By using both techniques failed tests could be found in both test sets. The second technique also resulted for both test sets in tests that were not executable. From these results the authors concluded that tests created with their approach can detect test cases that are *successful*, *failed* or *not executable*.

2.4.2 Application

To illustrate the approach of Longo et al. the use case *Describe a performer (new performer)* is used. In the first step an US-UID has to be created that displays the explicit information of a User Scenario as well as the underlying functional information. In this type of model the round boxes are states of the system. The rectangles contain the User's Input whilst the free text in the round boxes describes the system's output. Arrows are used to assign functional names to the data and to denote transitions between states. The US-UID for the example is displayed in figure 2.7. In the beginning the performerList contains only the performers a, b and c with attributes e, f and g. After the execution of the US-UID it also contains performer d with attributes x.

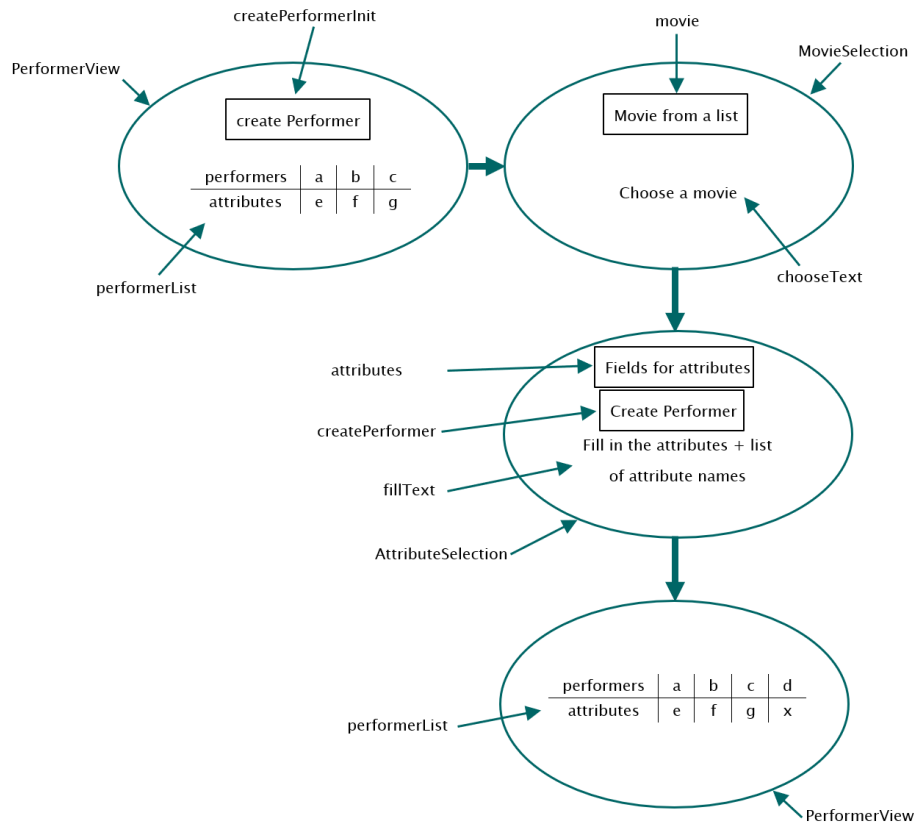


Figure 2.7: US-UID for the Use Case *Describe a performer (new performer)* of the Movie Manager application.

In the next step the functional information from the US-UID needs to be connected to the real objects and attributes of the *System-under-test*. This is done in a Fixture-Class that is marked with *@Fixture*. This class has to be written manually. The data flow during the execution of acceptance tests with FitNesse and the role of the Fixture-Class in this process is described in section ???. Inputs are part of the functional data on the start of the arrows in the US-UIDs. For the first display of the US-UID *PerformerView* the Fixture-Class needs methods to move to the next display, to set the starting performer list and to choose *create Performer*. Results have to be extracted from the System-under-test using Getter-Methods. One such result is the updated performer list in the last state of the *PerformerView*. This result can be compared to the expected result given in the US-UID.

In the final step a *Fit-Table* is created. This step is fully automatic with the tool of Longo et al. because it is only remodeling information from the US-UID. The functional information is placed in the top row whilst the explicit data of the User Scenarios is stored in the following rows. Each row represents an User Scenario. The resulting *Fit-table* for the example is displayed in table 2.7 on the next page.

Table 2.7: *Fit-table* for a specific User Scenario of the Use Case *Describe a performer (new performer)* of the Movie Manager application. The expected results end with a question mark.

Performer- List	Create- Performer- Init	movie	attributes	Create- Performer	performerList ?	d.movie ?	d.attributes ?
[a,b,c]	True	m	x	True	[a,b,c,d]	m	x

2.5 Comparison

As in the other chapters the approaches are compared in a synthesis matrix. The questions are the same as in the other chapters:

1a	Which artifacts and relations between artifacts are used in this approach? Which artifacts are created in the course of the approach? How are the artifacts characterized?
1b	What is required and/or input for the application of the approach?
1c	What steps does the approach consist of? Which information is used in which step and how? What are the results of the individual steps?
2a	Which usage scenarios are supported by the approach?
2b	Which stakeholders are supported by the usage scenarios?
2c	Which knowledge areas from SWEBOK can be assigned to the usage scenarios?
3a	What tool support is provided for the approach?
3b	Which steps of the approach are automated by a tool? Which steps are supported by a tool, but still have to be executed manually? Which steps are not supported by a tool?
4a	How was the approach evaluated?
4b	What are the (main) results of the evaluation?

	Approach 1	Approach 2
1a	<p>Initial artefacts: Use Case and Domain Models are used to create high level acceptance tests and robustness diagrams Robustness diagrams combine the information from the use cases and the domain model. During the creation of the robustness diagrams for the use cases objects and attributes may be identified that are missing in the domain model. The domain model is updated with this missing information. High level acceptance tests (HLATs) deliver an informal description of acceptance tests. They are tables and use keywords that are chosen by their creator. For each flow of a use case from the Use Case model a HLAT is created. Fit-tables are a form of executable acceptance tests that can be automatically executed by the tool FitNesse. For each HLAT a Fit-table is created using the information about the control flow in the respective robustness diagram.</p>	<p>User Scenarios through User Interaction Diagrams (US-UIDs) show the interaction during a User Scenario. They include User-Input, System-Output, states of interaction and transitions between states. Fit-tables are a form of executable acceptance tests that can be automatically executed by the tool FitNesse. The Fit-tables in this approach need a specific Fixture-Class for each Use Case that allows the flow of information between the Fit-table and the System-under-test.</p>
1b	<ul style="list-style-type: none"> • Use Case Model • Domain Model 	<p>Requirements in a non-specific type</p>
1c Part 1	<p>As initial artefacts Use Case models and Domain models are used. In the first step a HLAT is created for each flow of each use case from the Use Case model. The information about the preconditions, inputs and triggers for the HLATs gets extracted from the domain model. The second step is the creation of robustness diagrams for the use cases from the Use Case model. These diagrams also include the objects from the domain model and model the communication between those in the specific use case. If objects or attributes are found in this step that are necessary but not yet part of the domain model, they are added to the domain model. All the other models are updated to fit the domain model.</p>	<p>In the first step the US-UIDs are created by using the known requirements. The customer delivers the User Scenario and the developer add the functional information for the data used in the scenario. The Fit-tables are created automatically from the information of the US-UIDs. This step is done by the web framework.</p>

	Approach 1	Approach 2
1c Part 2	In the last step a Fit-table is created for each HLAT using the control flow that can be seen in the robustness diagram. Finally the created Fit-tables can be combined with the Use Case from the Use Case model that they belong to.	
2a	Business Analysts receive a systematic approach to create acceptance tests in the Fit-syntax. Customers receive a product that fits their requirements. Developers receive acceptance tests that they can use to determine which requirements of the customers they have already implemented and which they have to work on.	Customers and developers receive an approach to develop acceptance tests together that include User scenarios provided by the customer. Developers receive acceptance tests that they can use to determine which requirements of the customers they have already implemented and which they have to work on.
2b	<ul style="list-style-type: none"> • Developer • Customer • Business Analyst 	<ul style="list-style-type: none"> • Developer • Customer
2c	Developer: Software Construction (Test-driven development) Business Analyst & Customer: Software Testing	Customer/Developer: Software Requirements, Software Testing Developer: Software Construction (Test-Driven Development)
3a	The authors provide the tool UCAT in which Use Case Models and Fit-tables can be created and linked. The created Fit-tables can be automatically executed using FitNesse.	A web framework is provided by the authors in which US-UIDs can be created and converted to Fit-tables. These Fit-tables can be executed with FitNesse.
3b	The tool UCAT serves as an editor to create Use Case models and Fit-tables. Those two artefacts can also be linked in UCAT. Every other step in the creation of the acceptance tests is done without tool support. No step of the approach is done automatically.	The creation of the US-UIDs is supported by the web framework which serves as an editor. Converting the US-UIDs to Fit-tables is done fully automatically by the web framework. Every other step in the creation of the acceptance tests is done without tool support.

	Approach 1	Approach 2
4a	<ul style="list-style-type: none"> • Case study with an application (RestoMapper) • No real evaluation 	The authors created tests automatically from existing US-UIDs of an existing application using their approach. The resulting test set was compared to an existing test set that was created manually. The code of the application was manipulated using code mutation and lack of code. For the method code mutation the values of an array were changed manually. For lack of code a class was deleted.
4b	The approach can be applied on an example. Otherwise no evaluation results because the authors state that an evaluation is beyond the limitations of their work. The reason for this is that the quality of the created tests in this approach highly depends on the experience and skill of the person executing the approach.	Code mutation and lack of code resulted for both test sets in failed tests. Lack of code also resulted in not executable tests. The authors concluded that the tests created by their approach can successfully classify tests as successful, failed or not executable.

Both approaches provide a possible way to create acceptance tests that are executable with the tool *FitNesse*. El-Attar and Smith utilize use case models and domain models as their initial data while Longo et al. only need a non-specific description of the use cases. Generally El-Attar and Smith use more artefacts as their approach needs use case models, domain models, high-level acceptance tests and robustness diagrams as stepstones to the final representation of the executable acceptance tests. In the approach of Longo et al. only US-UIDs need to be created which are then automatically converted to *Fit-tables*. In contrary to El-Attar and Smith the approach of Longo et al. requires the creation of some code in the process: This is the case because the used *Fit-tables* differ between the two approaches. While El-Attar and Smith use the specific table types ActionFixture, RowFixture and ColumnFixture, Longo et al. use easier *Fit-tables* that are connected to the System-under-test via a Fixture-Class. This Fixture-Class has to be written manually.

Both approaches provide a tool to combine artefacts with the resulting acceptance tests which helps traceability. For both approaches the creation of the executable acceptance tests is still mostly or completely manual. While the approach of El-Attar and Smith uses no automation during the creation of the executable acceptance tests, the last step of the approach of Longo et al. is fully automatically. This is possible because the US-UIDs created in the approach of Longo et al. are a different way to display the information of a *Fit-table* and therefore can be directly converted to a *Fit-table*. The execution of the final acceptance tests is fully automatic for both approaches.

Both approaches involve customer and developers as stakeholders. The customer delivers the User Scenarios and receives (because of the development of acceptance tests) potentially a final product that is closer to his needs. The developers receive automatically executable tests that help them during the development process to determine which requirements are already satisfied and which still need to be implemented. While in the approach of Longo et al. the creation process of the acceptance tests is done by the customer and the developers together, in the

approach of El-Attar and Smith a Business Analyst is responsible for this process.

El-Attar and Smith only visualize their approach through an example and state that the approach cannot be validated in their work because it is beyond the limitations of their work. The reason for this is that all the steps to create the acceptance tests are done manually and therefore depend on the experience and skill of the analyst performing the steps. Longo et al. include a small evaluation in their work. They compare the tests created by their approach to tests that are created without guidelines from the same US-UIDs. During the testing phase they conclude that the tests created by their approach can be classified as *successful*, *failed* or *not executable*. Also changes in the source code of the System-under-test resulted in failed tests for both of the test sets which leads the authors to the conclusion that both the tests created without guidelines as well as the tests created with their approach can detect fails in the System-under-test.

2.6 Conclusion

The literature search showed that creating acceptance tests that are automatically executable with the specific tool *FitNesse* is not a widely researched topic in the literature. However, approaches exist that differ in their process to create tests. Two of these approaches were presented in this chapter:

The approach by El-Attar & Smith is aimed at larger projects and therefore, might not be useful for smaller products. It requires the creation of a lot of UML models. If an analyst exists that has experience in creating these models and at least a few of the used models are created anyway in the engineering process, then this approach might be useful.

The second approach by Longo et al. could be used for smaller projects where the customer is heavily involved. The customers have to be involved because they have to provide the User Scenarios in this approach. The approach is heavily dependent on the creation of US-UIDs that contain the information of Fit-tables in a different (possibly better) way. If the developers and customers prefer US-UIDs over Fit-tables and they want to use User Scenarios, this approach might be useful.

Overall, in the considered approaches the creation of acceptance tests is a process that is highly dependent on the experience and skill of the persons involved. Once the executable tests are created they are an easy way to measure how well the requirements of the customer are implemented.

3 Testing with a transition system

4 Testing with a timing component

5 Testing with a classification tree

6 Testing with a formal specification

7 Testing with system models

8 Testing functional and nonfunctional requirements in User Requirements Notation

9 Testing Non-Functional Requirements with Risk Analysis

9.1 Introduction

In contrast to functional requirements (FRs) that describe the program’s functionality, i.e. how it processes data and user input, non-functional requirements (NFRs) describe constraints that the program must adhere to [3]. Parts of NFRs are performance and security requirements which can directly affect the end-user but also maintainability requirements which are more important to development teams. While there are lot of resources about testing FRs in the form of unit-, integration- and end-to-end-tests, no common testing framework or guidance for testing NFRs exists.

For this reason, we look into two approaches [4] and [5]. While [4] was given in advance by the advisors of the seminar, [5] was found through a literature search, which is described in section 9.2. Each approach will be described and applied to an example in the form of movie management software in the respective sections 9.3 and 9.4.

Both approaches are compared in section 9.5 by using a synthesis matrix. We will look at which NFRs are covered and how they are tested. The results of this chapter will be summarized and concluded in section 9.6.

Please refer to the glossary for the following terms used throughout this chapter: test case, use case, NFR.

9.2 Literature Search

The starting point for the literature search was the paper given to us [4]. Based on this paper, we formulated the central research question:

“Which approaches for testing non-functional requirements systematically with risk analysis exist?”.

We focused on finding articles that covered the three most important keywords and phrases for the topic: *testing*, *non-functional requirements* and *risk analysis*. A quick search using these three phrases resulted in IEEE Xplore having the most promising results, whereas ACM¹ only showed a few. Because the given article [4] can also be found on IEEE Xplore², we focused our search onto that site but still looked at ACM.

¹<https://dl.acm.org/>

²<https://ieeexplore.ieee.org/Xplore/home.jsp>

To be able to evaluate the relevance of papers found during the literature search, we defined three relevance criteria:

1. Does the article cover non-functional requirements? They must not only be mentioned as a side note next to functional requirements.
2. Does the article combine risk analysis with tests?
3. Does the article cover *testing* of non-functional requirements?

Even though these relevance criteria basically only cover the research question, they filter out most non-relevant papers as we will see later on.

The search was carried out by using forward and backward snowballing as well as by using search terms with combinatorial modifiers. Only two papers reference [4] according to its IEEE Xplore site, both of which cover functional but not non-functional requirements. The paper itself references 23 papers. Of those papers, only few covered the first criterion and none covered the third criterion. [4] itself does not cover risk analysis as a main research topic but only covers it in a side note (see section 9.3). This is why search-term based search was performed using the key terms: non-functional requirements, testing and risk analysis.

Table 9.1 lists an excerpt of the term-based search. Listed are only those searches that returned promising results or highlight issues I encountered during the search. It can be seen that, if all keywords are combined using the AND operator with the default restrictions, no relevant results were returned. After a feedback from one advisor, the search was changed so that “non-functional requirements” and “testing” were expected in the paper’s abstract, but “risk analysis” was searched for in *all* metadata including the full text. It turned out that no papers were found which mentioned risk analysis as well as the other two keywords in their abstract.

We also discovered that the spelling of the term “non-functional” had a huge impact on the results returned by IEEE Xplore.

Table 9.1: Term based search results

Source	Date	Search query and restrictions	#Results (relevant)
IEEE Xplore	2020-11-11	"non-functional requirements" AND testing AND "risk analysis"	3 (0)
IEEE Xplore	2020-11-11	non-functional requirements AND testing AND risk analysis	12 (0)
IEEE Xplore	2020-11-11	risk AND "non-functional" and test	23 (2)
IEEE Xplore	2020-11-29	((("Abstract":nonfunctional requirements) AND "Abstract":Test) AND "Full Text & Metadata":"risk analysis")	2 (1)
IEEE Xplore	2020-11-29	((("Abstract":non-functional requirements) AND "Abstract":Test) AND "Full Text & Metadata":"risk analysis")	3 (0)
ACM	2020-11-29	[Abstract: "risk"] AND [Abstract: test*] AND [Abstract: "non functional"]	4 (1)

ACM	2020-11-29	[Abstract: test] AND [Abstract: "non functional"] AND [[Full Text: "risk analysis"] OR [Full Text: "risk"]]	20 (1)
-----	------------	---	--------

After this initial search, the resulting papers were evaluated and one paper was chosen. The papers to chose from included:

- “Scenario-Based Assessment of non-functional Requirements ” [6]
- “Alignment of requirements specification and testing: A systematic mapping study” [7]
- “Using Automated Tests for Communicating and Verifying Non-functional Requirements” [5]

Scenario-Based Assessment of non-functional Requirements covers all three criteria we defined at the start of our search. However, it limits itself to complex socio-technical systems and only looks at one non-functional requirement, which is the system’s performance. It limits itself to the evaluation of the reliability of certain aspects of software which interacts with humans to calculate the risk of human errors occurring. This is done by implementing scenarios–hence the title “scenario-based assessment”. The testing aspect of this paper is limited to human-interactions whose risks are evaluated. If scenarios fail this risk assessment, then so will the test.

Alignment of requirements specification and testing: A systematic mapping study is about papers that cover non-functional requirements. It is a study about such papers and lists approaches that are used to test NFRs. Some of which are mentioned in other chapters of this paper. However, none cover risk analysis. The paper itself does not give much insight into testing non-functional requirements itself.

The chosen article which we will further evaluate in the following sections is *Using Automated Tests for Communicating and Verifying Non-functional Requirements*. It covers the non-functional requirement “maintainability” and how it can be tested. It further explains it by using practical examples. However, the chosen paper does not cover risk-analysis. Since no paper could be found that covers all criteria, we were advised to focus on the testing of non-functional requirements and leave out risk-analysis.

9.3 Approach 1

9.3.1 Description of Approach 1

In their paper “Control Cases during the Software Development Life-Cycle” [4], J. Zou und C. J. Pavlovski based their work on so called “control cases” and “operating conditions” as tools for modeling and controlling NFRs.

“Control cases” are used as a format to communicate and discuss NFRs between management, requirement engineers, developers, and system users and to define qualitative attributes of the system.

Their work focuses on determining and revealing problems early on, for example bad perfor-

mance or security risks. The classic software development life cycle often focuses on these topics too late or not at all. However, control cases require NFRs to be defined first. To define them, the paper starts by introducing operating conditions. Operating conditions model constraints that apply to the system or a specific use case. These constraints are then used to model NFRs, hence the operating condition can be seen as a high level view on NFRs. Defining such constraints that apply to a certain use case is left to the reader or rather is mentioned as a step of the business process modeling.

Operating conditions can belong to one or more use cases and are not unique to a specific one. Conditions such as “Transaction Volume Condition: >400 concurrent users” can be applied to different use cases [4] and model exactly that: a condition under which the use case operates.

Control cases—as the name suggests—control the operating conditions and can be used to ensure that they are complied to. This makes it possible to mitigate business risks which may affect the business if the operating condition and its constraints are violated. Their paper visualizes the connection between these artifacts using an uniform modeling languages (UMLs) diagram, which can be seen in Figure 9.1 below.

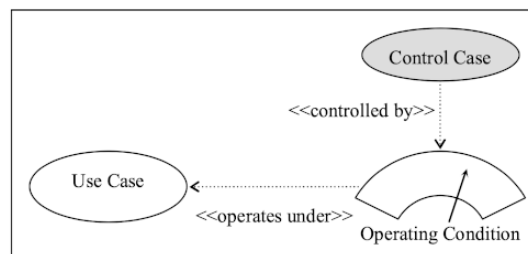


Figure 9.1: Association in Use Cases Modelling [4]

The paper creates such a control case by introducing a fictional example of a traveling agent. All of the previously mentioned artifacts are created during the “Business Process Modeling” and are refined throughout the software development life cycle. This means that operating conditions and control cases are defined together with use cases and can be incorporated together in a use case model. The paper does this for their fictional example which can be seen in Figure 9.2.

In this graphic, control cases are visualized as shaded ellipses and operating conditions as speedometers, though unspecified by the paper. This graphic also emphasizes that operating conditions are not bound to one specific use case but can be applied to different ones. And the control case is specific to one operating condition.

The reader is guided through all steps of the software development life cycle, so that a control case can be defined which is then used as the basis for a test case. Because the control case is associated to an operating condition, the test case is associated to it transitively as well. The test case exists to verify that the controls put in place to manage the operating condition are effective.

The paper does not give a detailed instruction how to model test cases. It only instructs testers to simulate the operating condition, for example by creating a huge work load on the server. With this simulation relevant metrics can be extracted that are used to verify the test case.

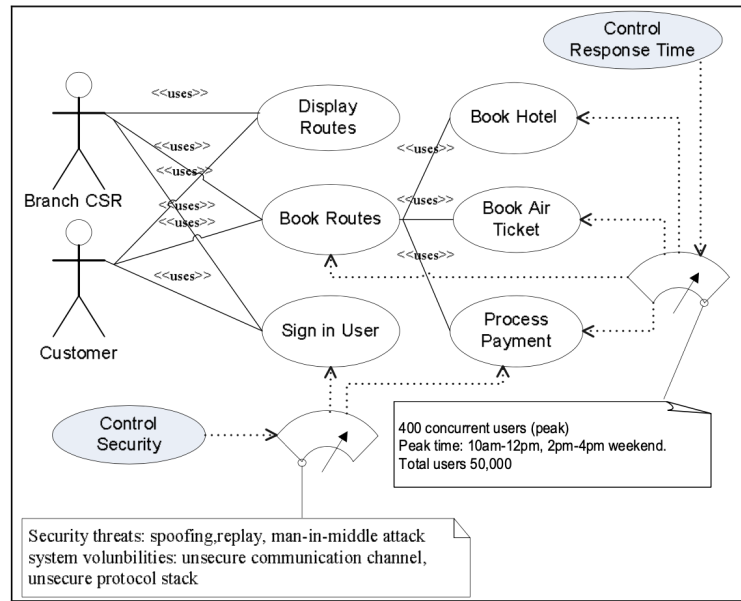


Figure 9.2: Use Case Model with Control Cases [4]

9.3.2 Application of Approach 1

J. Zou und C. J. Pavlovski focus on creating control cases. This is done for the movie manager example.

We first define one goal of our software: it must contain a movie list view that has smooth scrolling and can handle a large amount of movies. This also defines a constraint and therefore our operating condition: we must operate a smooth list view. If this cannot be accomplished an associated risk may affect the business. The control case bundles all of this in a matrix which is defined as in Table 9.2 on page 37.

Based on this control case, developers can start to implement the software. During testing stage, functional requirements can be tested by basing them on use cases. Non-functional requirements, on the other hand, can be tested by creating tests from control cases. The tester needs to simulate the operating condition. In our example above, that would mean to simulate the scrolling condition by creating a huge list of movies. The steps that must be executed for the test are combined into a test case, such as Table 9.3 on page 38.

By determining the operating condition that is associated to a use case, we were able to create a control case that reflects the NFRs. Based on the operating condition, we then created a test case that checks if the controls put in place by the control cases are enough to mitigate the business risk. Following this pattern, tests for non-functional requirements can be created systematically step by step.

Table 9.2: Control Case for Approach 1 of Topic 9

Control Case: Performance of the movie list view
Control Case ID: CC-001
Operating Condition: Scrolling Speed Condition
Description: The control case describes the “smoothness” while scrolling through the movie list view. Scrolling must be smooth. If it is not then users may assume bad performance.
NFR Category: Performance and Capacity
Associated Use Cases: Show movies in list view
Technical Constraints: GUI Framework, Operating System (e.g. 32bit system only allows addressing of 4GB main memory)
Vulnerability: Unknown number of movies. Users may only have a few or thousands of movies. Analyzing movies (or doing other work) must not lead to the movie list view being unresponsive. Having a lot of movies must not make the program run out of memory.
Threat Source: None (local software used by one user)
Operating Condition: There may be tens of thousands of movies. Assuming that each movie object has a size of 600kB (only meta data and a small thumbnail), loading 20,000 movies would lead up to 12GB of memory usage ³ . All movies must be represented in a list view.
Business Risk: If scrolling is not smooth, the user may switch to other software or leave a bad rating.
Probability: medium (likely few users are affected)
Risk Estimation: low (users with huge databases may accept higher load times or sluggishness in the UI)
Control: <ol style="list-style-type: none"> 1. Only load visible movies into main memory. Use “infinite scrolling” techniques. Remove those movies from main memory that are not visible to the user. 2. Only load the title into main memory. Load other details only if required. This reduces the memory footprint.

9.4 Approach 2

9.4.1 Description of Approach 2

In “Using Automated Tests for Communicating and Verifying Non-functional Requirements” [5], Robert Lagerstedt describes how testing NFRs can be automated by introducing a tool-based approach. The author only looks at NFRs in regards to software architecture which affects code quality in the sense of maintainability and security.

By looking at software architecture aspects as NFRs, Lagerstedt describes how software may be written by listing some architectural requirements. It should not have dependencies from lower code components into higher but only vice versa. Certain functions must not be called from some components to ensure encapsulation. Some functions may be blacklisted due to security

³From personal experience by maintaining a media manager. Users regularly report more than 10,000 movies in their database.

Table 9.3: Test Case for the movie manager example of topic 9, approach 1

Associated Control Case ID: CC-001
Test Objectives: Verify that the movie list view has no visible hiccups when scrolling through the list of movies.
Preconditions: Movie manager is up and running.
Test Steps: <ol style="list-style-type: none"> 1. Create 20.000 movies and load them into the movie manager 2. Open the movie list view 3. Scroll through the list of movies
Expected Result: <ol style="list-style-type: none"> 1. The end of the list view is reached. 2. No hiccups while scrolling were visible, i.e. no “sluggishness”.
Notes: The test must be performed on a system that has at most 8 GB of RAM to reflect common end-user hardware.
Test Result: Pass / Fail

concerns. All of these requirements are part of the software architecture and therefore a huge part of software quality and maintainability [5].

These NFRs must be communicated to developers. According to Lagerstedt, this is done by guidelines written by software architects. The compliance of these guidelines is often verified by different reports. These reports may be written for each code change as part of a code review or by other teams. Lagerstedt visualizes this in a simple UMLs diagram as can be seen in Figure 9.3. The graphic uses a rather high distance between the developer and the compliance report on purpose to symbolize that the two are asynchronous, this means that the report is not automated and feedback reaches the developer not immediately.

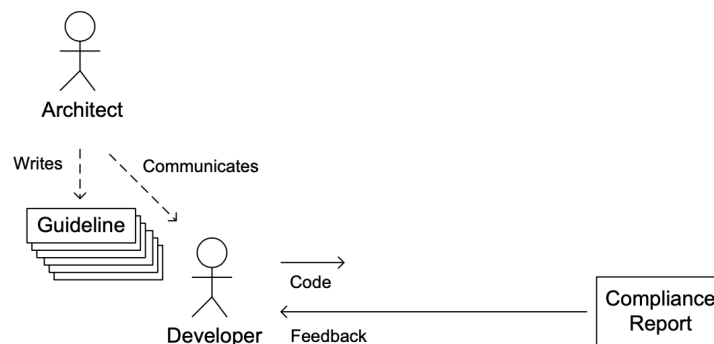


Figure 9.3: The common way of communicating architectural requirements [5]

This way of communicating guidelines is not very cost-efficient. Every developer has to read and understand the guidelines. Developers must be re-trained when changes are made or if

too many guidelines violations occur, because they have been forgotten. This is quite time consuming and prone to error. Creating reports about guideline compliance is time consuming as well. Furthermore, while code review should be performed for all code changes, mistakes may slip through.

That is why the author proposes automated testing of software architecture NFRs. This allows a fast tool-based feedback loop in which the developer gets a code review that can be incorporated without other developers having to look out for violations of guidelines. On top of that, by having this tight feedback loop, developers can learn the guidelines in an iterative way. Little to no training is required, which saves time to make new guidelines known to all developers.

The guidelines are written as tests. These tests can be included in existing static code analysis tools such as linters and other code checkers. Developers can see the results of such tools. Furthermore guidelines are communicated to the developer in case of a test failure. Lagerstedt uses Figure 9.4 to visualize this approach. Developers get feedback through different tools that the architect extends. Tools such as the editor, compiler or static analysis tools.

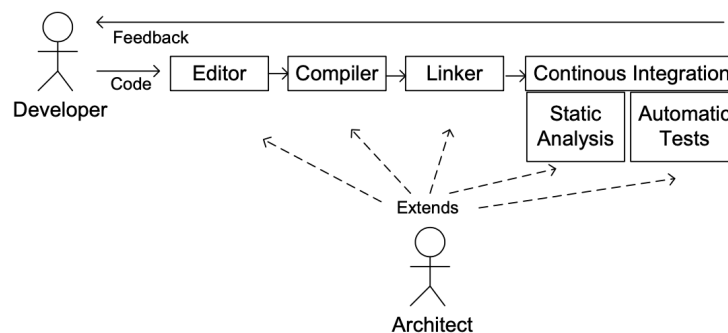


Figure 9.4: Suggested solution of communicating requirements [5]

According to Lagerstedt's personal experience, a tool based approach is superior to a guideline-only one. Productivity is increased while the time spent on communicating architectural guidelines is decreased. The number of non-compliant code is lower for the tool-based approach than for using guidelines and reports only.

9.4.2 Application of Approach 2

The paper works with architectural NFRs but does not explain how those can be modeled. To be able to apply the approach, we introduce another system function to the movie manager example that is listed in Table 9.4. This system function and the following NFRs are based on personal experience in maintaining an open-source media manager.

In Table 9.5 on page 40, two NFRs are listed which were created for the system function in Table 9.4. These two NFRs are based on personal experience. Both are transformed into pseudo code so that the NFRs can be executed automatically as part of the code review.

While these two NFRs can be written as guidelines, especially point two may be violated and may slip through code review. Violating point two may result in security issues or at least in unexpected behavior if the HTML contains unescaped characters.

Table 9.4: New system function for application of approach 2 of topic 9

Name	Export the movie to HTML
Description	An existing movie is exported to a single HTML file which can be viewed in any modern web browser
Precondition	Movie exists
Input	Movie details
Postcondition	HTML file exists with the movie's contents
Output	HTML file

Table 9.5: NFRs for the application of approach 2 of topic 9

No.	NFR	Explanation
1	IMDb IDs are encapsulated in a class	<p>All IMDb IDs have a certain format. They start with the string "tt" and end with 7-8 numbers. The ID must be validated which cannot be ensured by using a simple string. This is why an encapsulation in a class is required. Furthermore the programming language's type system can help to identify conversion bugs as well.</p> <p><i>Implementation in pseudo code</i></p> <pre> 1 for each \$variable in \$source: 2 if \$variable.startsWith("imdb") then 3 if typeof(\$variable) != "ImdbId" then 4 throw new Exception("Wrong class") 5 </pre>

2	Exported strings are escaped	<p>This is a security concern and can be implemented in different ways. We assume that an HTML-exporter was created which takes a movie object as an argument. This object may contain texts which contain HTML elements themselves. These elements need to be escaped. To ensure this NFRs, all strings must be run through a certain function which escapes strings. Because this may be missed by the developer, a new string-subclass is introduced which escapes its input automatically, e.g. <code>EscapedString</code>. Only this string class may be used in the HTML exporter.</p> <p><i>Implementation in pseudo code</i></p> <pre> 1 for each \$functionCall in \$HTMLExporter: 2 if \$functionCall == "writeText" then 3 \$arg = argument of(\$functionCall); 4 if typeof(\$arg) != "EscapedString" then 5 throw new Exception("Wrong_class") 6 </pre> <p><i>Note:</i> We assume that <code>writeText</code> is a method of a generic HTML-class which the HTML-exporter uses itself. We assume that the method cannot be changed to accept another argument type. Otherwise the language's type checker could already be able to find this issue.</p>
---	------------------------------	---

9.5 Comparison

For an improved comparison of these two approaches, a synthesis matrix is provided which references the following questions:

1. Description of the approach (What does the approach do?)
 - a) Which artifacts and relations between artifacts are used in this approach? Which artifacts are created in the course of the approach? How are the artifacts characterized?
 - b) What is required and/or input for the application of the approach?
 - c) Which steps does the approach consist of? Which information is used in which step and how? What are the results of the individual steps?
2. Benefits of the approach (Whom does the approach help and how?)
 - a) Which usage scenarios are supported by the approach?
 - b) Which stakeholders are supported by the usage scenarios?
 - c) Which knowledge areas from SWEBOK can be assigned to the usage scenarios?
3. Tool support for the approach (What tool support is available?)
 - a) What kind of tool support is provided for the approach?
 - b) Which steps of the approach are automated by a tool? Which steps are supported by a tool, but still have to be executed manually? Which steps are not supported by a tool?
4. Quality of the approach (How well does the approach work?)
 - a) How was the approach evaluated?
 - b) What are the (main) results of the evaluation?

No. Approach 1 [4]

1a Operating conditions are formed that work under specific use cases. These, on the other, hand are controlled by control cases and can be operated under them. Control cases describe the business risks in case that the operating condition cannot be fulfilled. Because use cases and control cases are tightly connected to each other, they can be modeled in one consolidated model.

Approach 2 [5]

Coding guidelines are written and transformed into tests that can be used by tools in code review. These point out issues that the developer can fix. Guidelines are characterized by the fact that they describe the code architecture.

- | | |
|---|--|
| <p>1b There are no preconditions because we start defining control cases at the beginning of the software development process, for example at the “Business Process Modelling”-step.</p> | <p>The guidelines cover code architecture. They must be transformable into automated tests (e.g. by a static code analyzer).</p> |
| <p>1c Preconditions/Constraints must be extracted from which NFRs are created, e.g. performance or security constraints. These constraints are modeled by operating conditions for which control cases are created. Their purpose is to mitigate business risk which is essentially the failure to fulfill the operating condition. For each control case a test case is added that checks if the controls put in place by the control case are effective. The test case basically recreates the operating condition, for example by using stress testing.</p> | <p>Code guidelines such as naming conventions or prohibited function-calls are defined. These are transformed into automated tests that can be executed by the developer (i.e. a tool based approach). The exact process is not explained and it is left to the reader how this may be implemented. It is only pointed out that existing tools such as compilers or static code analysis tools can be extended and used.</p> |
| <p>2a Early modeling of non-functional requirements. Being able to control requirements throughout the whole software development life cycle.</p> | <p>Maintainability, quality and security of the code base can be hold up to standards and may even be improved by giving automated feedback that points out NFRs which are violated by the developer.</p> |
| <p>2b Management, Requirements Engineer, Developers, Testers</p> | <p>Developer / test writer, Software Architect</p> |
| <p>2c Software Requirements (functional and non-functional requirements, acceptance tests), Software Testing (model based techniques)</p> | <p>Software Testing (Software Testing Tools, Test Techniques), Software Maintenance (Software Maintenance Tools)</p> |
| <p>3a No tool support for generating “Control Case”-Boxes and other artifacts</p> | <p>Existing static code analysis tools (e.g. linters), which can be extended by further tests.</p> |
| <p>3b No automation is done. Automation is only proposed as another step which can be implemented, e.g. through code generation with SysML.</p> | <p>Only code testing is performed automatically. And only tests for NFRs which were extracted from the software architects guidelines and that were transformed into automated tests. Those tests can be executed automatically during code review, e.g. by a continuous-integration service which tests each code change. Writing the tests is still a manual job.</p> |
| <p>4a The approach was explained by creating a fictional example and going through all steps of the software development life cycle by extending the example. No evaluation was performed, though.</p> | <p>No evaluation was performed. The conclusion, i.e. success of the approach, is based on personal experience only.</p> |

4b N/A

Based on his experience in both small and large organizations, Lagerstedt concludes that automated verification of non-functional requirements by using tool-chain feedback is superior to classic guidelines that need to be checked by humans. By evaluation of his prior experience, he concludes increased productivity and a decrease in time spent on communicating architectural requirements.

If we compare the two papers using the synthesis matrix above, we notice that they do not share a lot. That is not surprising: the second paper is very specific and only deals with architectural NFRs in code. The first paper, on the other hand, can be applied to different NFRs, not limiting itself to a specific one. Only the first paper mentions risk analysis but only as part of a control case.

Both papers do not give specific instructions how test cases can be modeled. While the first paper only says to “simulate the operating condition” [4], it leaves out details. For example security NFRs are explicitly mentioned but it is left out how an operating condition for that NFR can be simulated. Also the example test case from the paper is essentially a stress test. The second paper leaves it to the reader to develop automated tests and only mentions that static code analysis tools can be used.

While the second paper talks about test automation, it does not talk about creating tests automatically but rather about running them automatically [5]. The first paper does not include any automation step at all. Neither for creating test cases automatically nor for running them.

Both do not include an evaluation of their results besides personal experience. The first article states no evaluation at all and only discusses the approach for defining the control case.

9.6 Conclusion

Both articles deal with NFRs. While [4] describes how these can be defined and controlled, it does not specify a way to test them except for simulating the operating condition. In the same way there is no description of how the business risk affects the test case except for defining the test-priority. However, it explains in great detail how control cases and operating conditions can be defined and how they interact with use cases and functional requirements, which raised my interest in the overall topic of testing NFRs. But the lack of detailed explanation for test case creation makes it difficult for me to assess the usefulness of the approach. After reading the paper I may know how to model NFRs with operating conditions but still wonder how they can be properly tested.

[5] on the other hand leaves it to software architects to define NFRs. The paper only uses architectural NFRs that exist as code conventions and other guidelines. The author describes why having automated tests is a necessity of software development in regards to cost efficiency and how it mitigates human error during code review which can be seen as a risk to code maintainability. This corresponds to my personal experience. By using a code formatter, the amount of formatting related review comments went down to zero. By introducing a new linter rule, I was able to automatically fix company branding issues in product messages which none

of my colleagues were even aware of. I can therefore only emphasize that communication of NFRs is more effective and efficient when a tool based approach is used.

Finally, both articles mention risk analysis only as a side note, if mentioned at all. It is left to the reader where risks are mitigated. The conclusion is that NFRs with higher risks need to be paid more attention to by giving the tests higher priority.

10 Testing nonfunctional requirements with aspects

11 Conclusion

12 Bibliography

- [1] Chen, K, Zhang, W., Zhao, H.: An approach to constructing feature models based on requirements clustering. In: 13th IEEE International Conference on Requirements Engineering (RE05). pp. 31-40. (2005)
- [2] Institut für Geographie Lehrstuhl für Allgemeine Wirtschafts- und Sozialgeographie: An Hinweise zum wissenschaftlichen Arbeiten. http://www.geogr.uni-jena.de/fileadmin/Geoinformatik/Lehre/backup_05_2007/pdf-dokumente/Skript_WissArbeiten.pdf
- [3] PLATZHALTER
- [4] J. Zou and C. J. Pavlovski, "Control Cases during the Software Development Life-Cycle," 2008 IEEE Congress on Services - Part I, Honolulu, HI, 2008, pp. 337-344, doi: 10.1109/SERVICES-1.2008.46.
- [5] R. Lagerstedt, "Using automated tests for communicating and verifying non-functional requirements," 2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET), Karlskrona, 2014, pp. 26-28, doi: 10.1109/RET.2014.6908675.
- [6] A. Gregoriades and A. Sutcliffe, "Scenario-based assessment of nonfunctional requirements," in IEEE Transactions on Software Engineering, vol. 31, no. 5, pp. 392-409, May 2005, doi: 10.1109/TSE.2005.59.
- [7] Z. A. Barmi, A. H. Ebrahimi and R. Feldt, "Alignment of Requirements Specification and Testing: A Systematic Mapping Study," 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, 2011, pp. 476-485, doi: 10.1109/ICSTW.2011.58.
- [8] El-Attar, M., Miller, J. Developing comprehensive acceptance tests from use cases and robustness diagrams. Requirements Eng 15, 285–306 (2010).
- [9] Longo, D., Vilain, P., Pereira da Silva, L., Mello, R.: A web framework for test automation: user scenarios through user interaction diagrams. In Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services (iiWAS '16). Association for Computing Machinery, New York, NY, USA, 458–467 (2016).
- [10] Longo, D. H., and Vilain, P.: Creating User Scenarios through User Interaction Diagrams by Non-Technical Customers. In 27th International Conference on Software Engineering and Knowledge Engineering. 330-335 (2015).
- [11] ACM Digital Library. <https://dl.acm.org/>
- [12] FitNesse. <http://docs.fitnessse.org/FrontPage>
- [13] IEEE Xplore. <https://ieeexplore.ieee.org/Xplore/home.jsp>
- [14] Springer Link. <https://link.springer.com/>
- [15] Science Direct. <https://www.sciencedirect.com/>
- [16] FitNesse: Writing Fit Tables. <http://fitnessse.org/FitNesse.UserGuide.WritingAcceptanceTests.FitFramework.WritingFitTables>

List of Figures

1.1	Requirements in User-Task notation for the MovieManager software, a mobile application for managing movie collections.	9
2.1	Overview of the data exchange in the execution of <i>Fit-tables</i> with <i>FitNesse</i> . The Fit-tables can be created and maintained in FitNesse.	11
2.2	Overview of the steps in the approach of El-Attar and Smith.	13
2.3	Use Case Model for the Movie Manager application.	15
2.4	Domain Model for the Movie Manager application.	15
2.5	Robustness Diagram for the Use Case <i>Describe a performer</i> of the Movie Manager application.	17
2.6	Overview of the steps in the approach of Longo et al.	18
2.7	US-UID for the Use Case <i>Describe a performer (new performer)</i> of the Movie Manager application.	20
9.1	Association in Use Cases Modelling [4]	35
9.2	Use Case Model with Control Cases [4]	36
9.3	The common way of communicating architectural requirements [5]	38
9.4	Suggested solution of communicating requirements [5]	39

List of Tables

2.1	Overview of the search-term-based literature search.	12
2.2	General structure of a HLAT.	13
2.3	HLATs for the Use Case <i>Describe a performer</i> of the Movie Manager application.	16
2.4	Executable Acceptance Tests for the scenario <i>Describe a performer, new performer</i> of the Movie Manager application in form of an <i>ActionFixture</i> . A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.	17
2.5	Executable Acceptance Tests for the scenario <i>Describe a performer, existing performer</i> of the Movie Manager application in form of an <i>ActionFixture</i> . A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.	17
2.6	Executable Acceptance Tests for the scenario <i>Describe a performer, view performers</i> of the Movie Manager application in form of an <i>ActionFixture</i> . A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.	18
2.7	<i>Fit-table</i> for a specific User Scenario of the Use Case <i>Describe a performer (new performer)</i> of the Movie Manager application. The expected results end with a question mark.	21
9.1	Term based search results	33
9.2	Control Case for Approach 1 of Topic 9	37
9.3	Test Case for the movie manager example of topic 9, approach 1	38
9.4	New system function for application of approach 2 of topic 9	40
9.5	NFRs for the application of approach 2 of topic 9	40