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

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 show [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 FitNess, 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 loose overview over the tests 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.

[1] TODO

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

chapter 6 *ist noch nicht online; TODO*

In **??** 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 9steps 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 *create movie*, *change detail movie data*, *link existing performer*, *unlink performer*, *show movie*, *list movies*, *search*, *sort movies*, *show performer details* and *show movie in IMDB*
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 *remove movie*
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 *create linked performer*, *change detail performer data*, *link existing movie*, *unlink movie*, *list performers*, *sort performers*, *search*, *show performer*, *show movie details* and *show performer in IMDB*
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 *create linked performer*
2b	**Remove performer** Removes a performer.	Implemented in function *remove performer*
3	**Manage watched Movies** Record date when the movie was last watched.	Implemented in function *watch movie*
4	**Rate Movie or Performer** Rate or view ratings (sorted).	Provide a fixed rating value list. Implemented in functions *rate movie*, *rate performer*, *calculate overall rating of movie* and *sort movie*

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*.

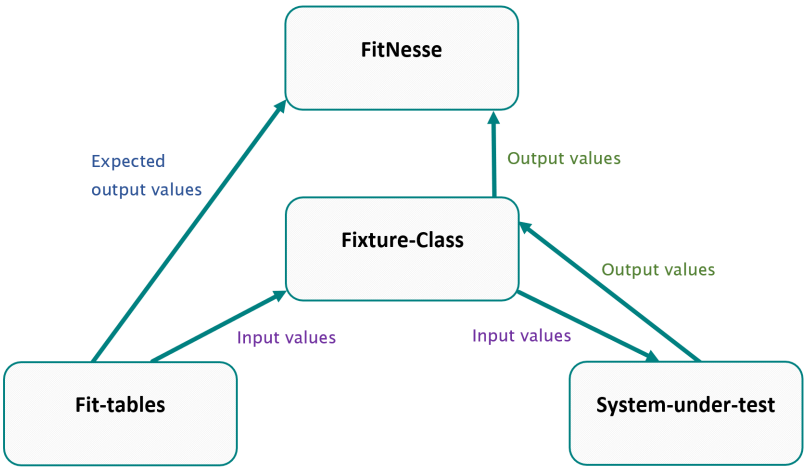

Figure 2.1: Overview of the data exchange in the execution of *Fit-tables* with *FitNesse*. The Fit-tables can be created and maintained in FitNesse.

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 criterion 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 fitnesse	10	4	4	**Longo et al., 2016**
IEEE Xplore	18.11.20	Publishing year: 2009-2020	„acceptance test“ AND fitnesse	2	1	1	
Springer Link	19.11.20	Publishing year: 2009-2020, no preview-only content	„acceptance test“ AND fitnesse	11	2	1	
Science Direct	19.11.20	Publishing year: 2009-2020	„acceptance test“ AND fitnesse	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.

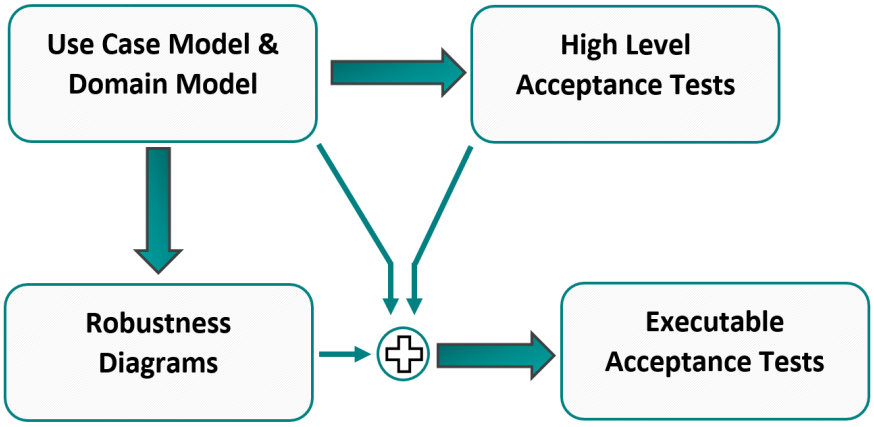

Figure 2.2: Overview of the steps in the approach of El-Attar and Smith.

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 where 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 where 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)

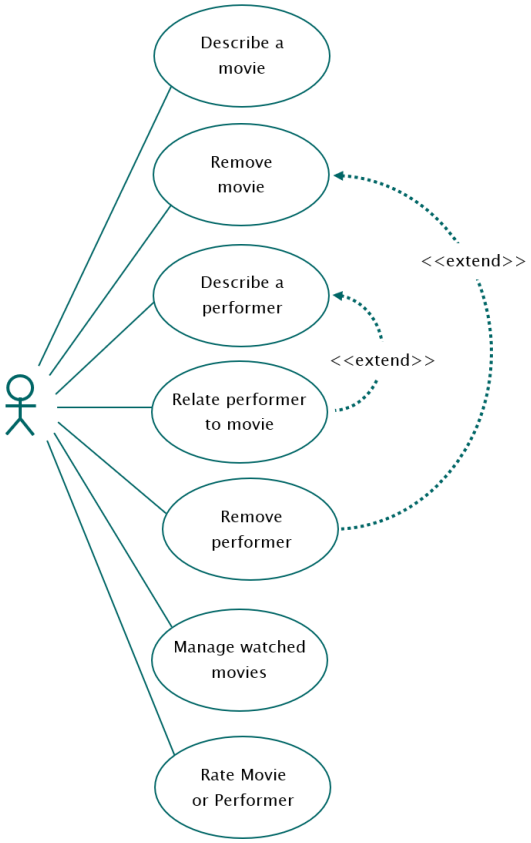

Figure 2.3: Use Case Model for the Movie Manager application.

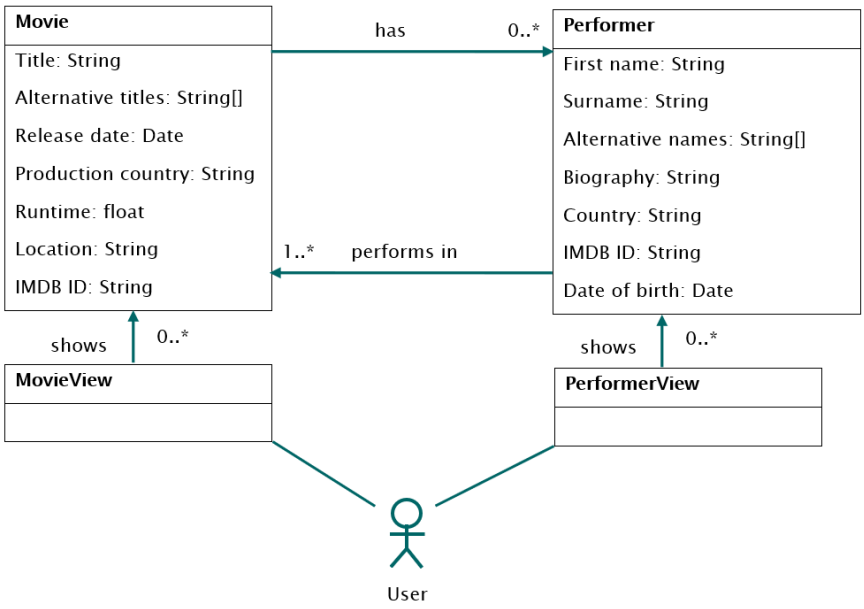

Figure 2.4: Domain Model for the Movie Manager application.

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.

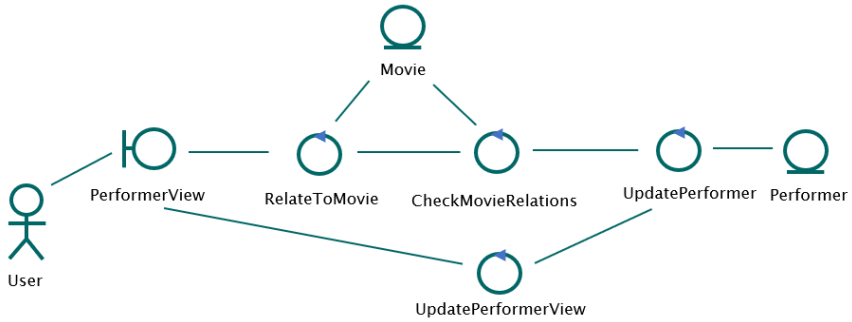

Figure 2.5: Robustness Diagram for the Use Case *Describe a performer* of the Movie Manager application.

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.

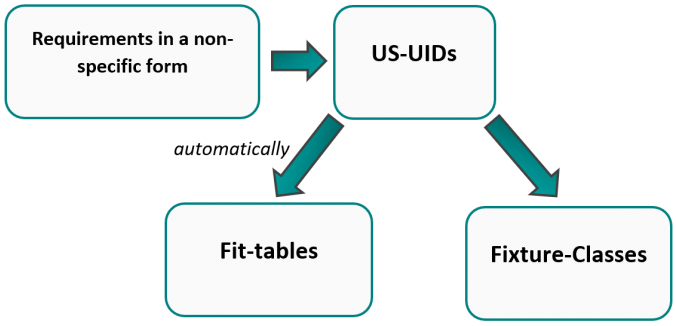

Figure 2.6: Overview of the steps in the approach of Longo et al.

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.

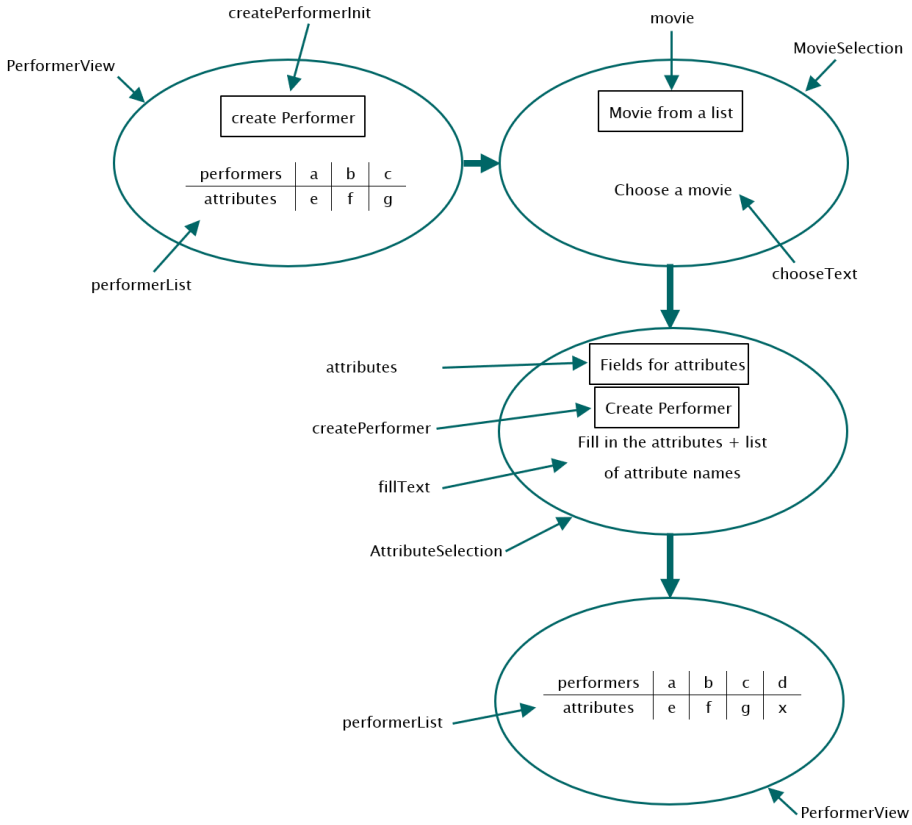

Figure 2.7: US-UID for the Use Case *Describe a performer (new performer)* of the Movie Manager application.

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-Perfomer	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 artefacts and relations between artefacts are used in this approach? Which artefacts are created in the course of the approach? How are the artefacts 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	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.	**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.
1b	• Use Case Model • Domain Model	Requirements in a non-specific type
1c Part 1	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.	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.

	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	• Developer • Customer • Business Analyst	• 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	• 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 intermediate steps 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

3.1 Introduction

System tests are used to make sure that clients receive exactly the kind of software they previously specified within the order contract submitted to the software vendor. Oftentimes what was specified and what was implemented does not match entirely in the end. The software vendor faces traceability problems to track which requirements could be covered in which tests and therefore which requirements got implemented. How does one make sure that each atomic functional and robustness requirement specified is covered in the requirements' implementation?

To solve this, the process of formal definition of requirements and matching system tests must be brought closer together. Testing should already be possible in early stages of development, to be precise during the specification phase already. To automatically derive test scenarios from means of the specification area *transition systems* are used. They help to generate test paths as possible combinations of fine-grained functional requirements received through the traversal. An equivalent approach can be used to derive robustness tests as well. In this process requirements can furthermore be tested on their consistency, correctness and integration and eventually can be refined further.

For this purpose the two approaches [19] and [20] were analyzed. While [19] was given in advance by the advisors, [20] was discovered through an extensive literature search shown in section 3.2. Both approaches will be explained in the following sections 3.3 and 3.4 and applied to the movie management software example. A comparison between the two approaches will be drawn using a synthesis matrix shown in section 3.5. The main results of testing with transitions systems will be summarized in section 3.6.

Please refer to the glossary in order to receive a common understanding of the following terms used in the sections below: contract, coverage criterion, interaction overview diagram, object constraint language, operation, test case, test objective, test scenario, transition system, UC-System, UC-SCSystem, use case, use case scenario, XML metadata interchange.

3.2 Literature Search

The literature research was driven by the central research question (RQ):

„Which approaches for automatic generation of system tests exist that are using contract enriched use cases or other use case related means of the specification area within a transition system simulation model?“

The focus during this literature search was on finding a second approach to automatically generate system tests from means of the specification phase by exhaustively simulating a transition

system to generate test paths similar to [19]. But as [19] is restricted on using *contract* enriched *use cases* and *use case scenarios*, the way how *test objectives* are derived was this time freely selectable to receive another new, but similar approach. The pre-search results were promising both on IEEE Xplore and ACM. Only some papers on ACM could not be accessed publicly. The number of results was considered to be sufficient to cover all relevant scientific papers, which is why the research was restricted on these two platforms. Furthermore, two relevance criteria inspired by the central RQ were defined:

- Does the method described in the article generate system tests automatically from use cases or other use case related means of the specification area?

- Are test objectives generated using some kind of simulation model based on use case contracts (pre- and postconditions) or similar transition system approaches?

As system tests can not exclusively be derived from means of the specification area, an article should restrict to generating system tests from use case related means of the specification area. To derive test objectives a transition system should be simulated exhaustively based on contract definitions (pre- and postconditions).

The search was done using both snowballing and search term techniques. 140 papers were found referencing [19] and 46 articles were referenced by [19]. The results of the snowballing search can be found in Table 3.1. The backward snowballing approach was restricted on references stated in the *Related Work* chapter as all other references relate to preceding work that serve as basic knowledge to realize the transition system approach in [19]. Additionally, most of the references were quite old since the original paper was published in 2006. Therefore not all papers could be found on IEEE Xplore or ACM, which is why the number of considered papers is even lower than the number of original references found. Why specific papers were considered not suitable or only partly suitable is documented in [21].

Table 3.1: Results of snowballing techniques

	Yes	**Possibly**	**No**
Forward snowballing	5	9	63
Backward snowballing	2	1	2

Seach-term based search was done using the following key terms: system tests, automatic generation, transition system, simulation model, use cases, contracts. Only papers published between 2006 and 2020 having the search term "test" and "use case" in their publication title were evaluated.

Table 3.2: Results of search-term based technique

Source	Date	Search restrictions	Search query	#Results
IEEE Xplore	2020-11-21	"system tests" in document title; "automatic generation" in document title; "transition system" in full text & metadata; "simulation model" in full text & metadata; "use cases" in document title; "contracts" in full text & metadata;	"system tests" AND "automatic generation" AND "transition system" AND "simulation model" AND "use cases" AND contracts	0
IEEE Xplore	2020-11-21	"system tests" in document title; "automatic generation" in abstract; "transition system" in full text & metadata; "use cases" in document title; "contracts" in full text & metadata;	"system tests" AND "automatic generation" AND "transition system" AND "use cases" AND contracts	0
IEEE Xplore	2020-11-21	"test" in document title; "transition system" in full text & metadata; "use cases" in document title	"system tests" AND "transition system" AND "use cases"	82
ACM	2020-11-21	"system tests" in title; "automatic generation" in title; "transition system" in full text; "simulation model" in full text; "use cases" in title; "contracts" in full text;	"system tests" AND "automatic generation" AND "transition system" AND "simulation model" AND "use cases" AND contracts	0
ACM	2020-11-21	"system tests" in title; "automatic generation" in abstract; "transition system" in full text; "use cases" in title; "contracts" in full text;	"system tests" AND "automatic generation" AND "transition system" AND "use cases" AND contracts	0
ACM	2020-11-21	"system tests" in title; "transition system" in full text; "use cases" in title	"system tests" AND "transition system" AND "use cases"	0
ACM	2020-11-21	"test" in title; "use case" in title	"system tests" AND "use cases"	8

From the resulting papers, only one was considered suitable as a potential second paper. After the search for potential articles to be evaluated was finished, a choice between eight remaining papers from the initial search had to be made:

- System Testing using UML Models [22],
- An Automatic Tool for Generating Test Cases from the System's Requirements [23],
- Automated Test Case Generation from Use Case: A Model Based Approach [24],
- Requirements Document Based Test Scenario Generation for Web Application Scenario Testing [25],
- An Approach to Modeling and Testing Web Applications Based on Use Cases [26],
- Test cases generation from UML state diagrams [27],
- Requirements by Contracts allow Automated System Testing [28],
- An Automated Approach to System Testing Based on Scenarios and Operations Contracts [20].

The decision criteria are based on the different search terms mentioned above and the already defined criteria. Additionally, focus of the selected paper should lie on creating system tests for any generic application area, not just UI related parts of an application.

The paper *An Automatic Tool for Generating Test Cases from the System's Requirements* was not chosen as it does not focus on testing the consistency of use case combinations with contracts to build test objectives as in the original paper. Furthermore, it is not as in-depth as the original paper. Contract enriched use cases could neither be found in *System Testing using UML Models.*

Automated Test Case Generation from Use Case: A Model Based Approach really embodies the principle of state base modeling based on use cases with its *interaction finite automaton* (IFA), but doesn't introduce a formal language to define use cases and its contracts.

Requirements Document Based Test Scenario Generation for Web Application Scenario Testing as well as *An Approach to Modeling and Testing Web Applications Based on Use Cases* are specifically optimized for web application *test scenarios* and therefore not as general and universally applicable as the original paper.

Test cases generation from UML state diagrams and *Requirements by Contracts allow Automated System Testing* could unfortunately not be accessed in full length in IEEE Xplore.

The chosen article to further evaluate is *An Automated Approach to System Testing based on Scenarios and Operations Contracts*, as it introduces a second way to create system tests from use case scenarios as UML 2.0 models by enriching it with contracts and by transforming the formalized scenarios to a transition system to derive test objectives. It uses a more graphical approach to define use cases instead of using a formalized language to do so and goes deeper down into use-case level instead of system-level test generation. A more in-depth comparison between the two papers can be found in section 3.5.

3.3 Automatic Test Generation: A Use Case Driven Approach

3.3.1 Description

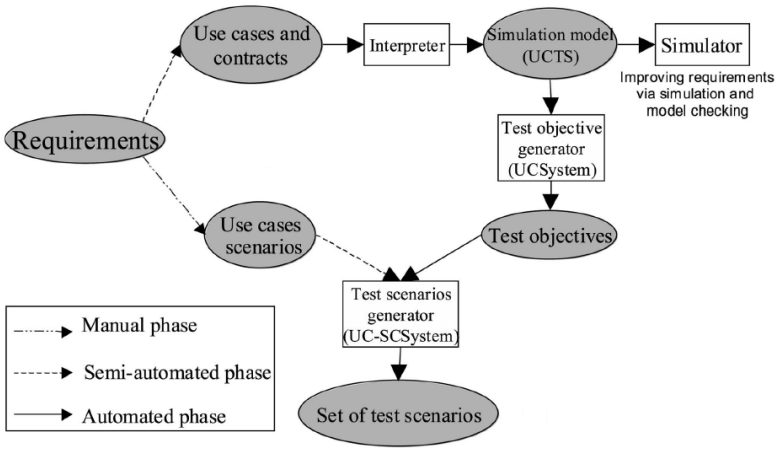

Figure 3.1: Flow of generating test scenarios [19]

The approach consists of two phases (see Figure 3.1). In the first phase, UML use cases get enhanced with contracts (pre- and postconditions). Use cases can be seen as the systems' main functions whereas contracts are used to infer the correct partial ordering of functionalities that the system should offer. They express the ordering constraints between the use cases during the simulation process to build the transition system. The contracts are made executable for logical evaluation by writing them in the form of requirement-level first-order logical expressions. They consist of predicates with and are used to describe facts in the system like actor state, main concept state, roles, and more. Each predicate can either evaluate to true or false, but never to undefined. A dedicated editor tool helps to manage the predicates and guides the design of contracts to maintain a nonredundant, minimal set of contracts and predicates. Contracts therefore specify the system properties to make a use case applicable (preconditions) and define the system properties (predicates) acquired after their application (deterministic postconditions). Parameters to use cases can either be system actors or main concepts of the use case, which are represented as normal instantiated objects of classes during the test scenario generation process.

Through exhaustive simulation by the prototype/interpreter-tool *UC-System* a *use case transition system* (UCTS) is built, which serves as a model for all valid sequences of use cases. Therefore first an initial state and enumerations of different business entities that later serve as parameters to use cases are declared. Then all use cases get instantiated by replacing the set of formal parameters with all the possible combinations of their possible actual values (i.e. actors and main concepts). To apply an instantiated use case the simulation state and the precondition of an instantiated use case must match. The simulation state is then updated according to the postcondition of the contract. The initial state defines which predicates are true from the very beginning, the current simulation state covers all instantiated predicates which are evaluated to true. Now all possible paths are traversed exhaustively until a final UCTS was generated.

During this step, the requirement engineer has also the chance to check and possibly correct the requirements before the tests are generated. Inconsistencies between predicates and contracts can be identified as well as underspecification or errors in the requirements. Invariants can also be checked. The number of states in the transition system can be calculated with

$$maxsize_{UCTS} = 2^{n_{ip}} \tag{3.1}$$

where

$$n_{ip} = p \cdot max_{instances}^{max_{param}}. \tag{3.2}$$

n_{ip} describes the amount of instantiated predicates, p the number of predicates, $max_{instances}$ the maximum amount of instances for a parameter (actors and main concepts of the use case), and max_{param} the maximum amount of parameters per predicate p. In practice, many of the potential states are not reachable and only a small number of instances are necessary for achieving a proper statement coverage.

Now relevant test objectives get extracted from the UCTS by applying predefined *coverage criteria*.

The *All Edges* (AE) criterion makes sure that all state transitions are covered, whereas the *All Vertices* (AV) criterion guarantees that all states (predicates) are reached within the set of test objectives. The *All Instantiated Use Cases* (AIUC) criterion is helpful in case a state transition can be done by multiple use cases or a use case leads to no state change. A combination od AV and AIUC is the *All vertices and All Instantiated Use Cases* (AV-AIUC) criterion. The most strict criterion is probably the *All Precondition Terms* (APT) criterion, which makes sure that all possible ways to apply each use case are exercised. Now, these criteria are mainly introduced to generate functional tests. The *Robustness* criterion on the other hand explicitly exercises a use case in as many different ways as to make its precondition false. Therefore valid test paths are generated, which lead to an invalid application of a use case to generate robustness tests from. All algorithms are based on breadth-first search in the UCTS to obtain small test-objectives that are human-understandable and meaningful.

Subsequently, in the second phase, test scenarios get generated by replacing each use case in a test objective with the according use case scenario that is compatible in terms of static contract matching. This is done by the prototype-tool *UC-SCSystem*. The use case scenarios were also attached with contracts beforehand. This time the contracts contain more detailed pre- and postconditions. There are contracts that rely on the rest of the model, they are written in *Object Constraint Language* (OCL), and there are contracts that rely on the predicates of the use cases. Now the exchange of messages involved between the environment and the system is also specified. Note that all use case scenarios are system-level scenarios. Eventually, additional parameters and messages need to be passed manually before executable *test cases* can be generated. The process results in executable test scenarios that get evaluated using the statement coverage metric.

One possible challenge of the approach is that the simulation model has to be compact enough to avoid combinatorial explosion of the internal states. Therefore the two-phase approach was chosen and parameters to instantiate use cases during the simulation can often be restricted to only the main system concepts and actors. Furthermore, the above-mentioned test objective generation criteria were identified through experimental comparisons and help to keep the number of test objectives in a reasonable scope.

The generated test scenarios can either lead to a pass verdict, a fail verdict (in case a postcondition is violated), or an inconclusive verdict. The latter is invoked if a precondition is evaluated to false and the test scenario was not executed entirely. This could be because of underspecification or because of inappropriate test data. To solve this either a new initial state (test data) has to be defined or additional test cases that execute the remaining use case scenarios need to be provided.

To evaluate the approach the original authors used three software products: An Automated Teller Machine (ATM) with 850 lines of code, an FTP server with 500 lines of code, and a virtual meeting (VM) server with 2.500 lines of code. Statistics on the amount of generated test cases can be found in Table 3.3.

Table 3.3: Statictics of the generated test cases

	ATM	**FTP**	**VM**
# use cases	5	14	14
# nominal UC-scenarios	5	14	14
# exceptional UC-scenarios	17	14	14
# generated functional test cases	6	14	15
# generated robustness test cases	17	33	65

Taken the example of the VM server, most coverage criteria reached up to 70% code coverage, when including robustness test cases even up to 80%. For more detailed information see Table 3.4.

Table 3.4: Statement coverage reached by the generated test cases

	ATM	**FTP**	**VM**
% of functional code covered	100%	90.7%	100%
% of robustness wrt. the spec covered	42.31%	38.6%	52%
% of code covered (total)	94.76%	72.5%	80%

All coverage criteria are almost equal in their achieved code coverage, with the exception of the AV criterion. Here the code coverage is low as not all use cases can be covered, especially those use cases that do not change the system state are missing. The ratio between the covered statements and the amount of generated test cases gives information about the efficiency of the generated test scenarios. Here the AIUC and APT criteria scored best. The APT criterion manages to reach 100% functional code coverage with only 15 test cases. The efficiency of the robustness criterion on the other hand scored quite low, 65 test cases could only cover up to 50% of the equivalent robustness code. Therefore the approach works well for functional code, but not so well for robustness code. This is because only violations of the use case attached preconditions are taken into account, inappropriate test data, or violating the more detailed preconditions of use case scenarios are not included in the test generation process.

One possible extension of the approach was also considered. Activity diagrams could be chosen to model the use case dependencies in a more graphical approach, which is then shown in the upcoming approach two.

3.3.2 Application

In the first step, the test objectives have to be derived. Therefore we define the use cases and their contracts (Listing 3.1) as requirement-level first-order logical expressions. The contracts

are used to infer the correct partial ordering of functionalities that the system should offer. Only the use cases that really impact the state of the transition system were specified for this example. The notation used is equal to the one proposed in the paper. *UC* introduces the identifier and parameters of a use case, *pre* marks the beginning of the precondition logical expression and *post* the beginning of the postcondition logical expression. Furthermore, logical operators like *and* and *or*, quantifiers like *forall* and *exists*, and implications by using *implies* can be used. The expression *@pre* ensures that a given logical expression has already been evaluated to true when evaluating the precondition.

Listing 3.1: Contracts attached to use cases

```
UC createMovie(m: movie)
post createdMovie(m)

UC createLinkedPerformer(p: performer, m: movie)
pre createdMovie(m)
post createdPerformer(p) and createdLink(p,m)

UC rateMovie(m: movie)
pre createdMovie(m)
post calculatedOverallRating(m)

UC ratePerformer(p: performer)
pre createdPerformer(p)
post forall(m: movie){ createdLink(p,m)@pre implies calculatedOverallRating(m) }

UC linkExistingMovie(m: movie, p: performer)
pre createdMovie(m) and createdPerformer(p)
post not createdLink(p,m)@pre implies (createdLink(p,m) and
 calculatedOverallRating(m))

UC linkExistingPerformer(m: movie, p: performer)
pre createdMovie(m) and createdPerformer(p)
post not createdLink(p,m)@pre implies (createdLink(p,m) and
 calculatedOverallRating(m))

UC unlinkMovie(m: movie, p: performer)
pre createdMovie(m) and createdPerformer(p) and createdLink(p,m)
post calculatedOverallRating(m) and not createdLink(p,m) and not exists(m2: movie){
 createdLink(p,m2) }@pre implies not createdPerformer(p)

UC unlinkPerformer(m: movie, p: performer)
pre createdMovie(m) and createdPerformer(p) and createdLink(p,m)
post calculatedOverallRating(m) and not createdLink(p,m) and not exists(m2: movie){
 createdLink(p,m2) }@pre implies not createdPerformer(p)

UC removeMovie(m: movie)
pre createdMovie(m)
post not createdMovie(m) and forall(p: performer){ not createdLink(p,m) } and not
 exist(m2: movie){ createdLink(p,m2) }@pre implies not createdPerformer(p)

UC removePerformer(p: performer)
pre createdPerformer(p)
post not createdPerformer(p) and forall(m: movie){ not createdLink(p,m) and
 calculatedOverallRating(m) }
```

After that, the UC-System prototype/interpreter tool should build the UCTS (Figure 3.2) through exhaustive simulation. The pool of parameters was restricted to one movie and performer to avoid a combinatorial explosion for this example. To build instantiated use cases the set of formal parameters are replaced with all the possible combinations of their actual values. In our case, we use the most simple approach by just having one possible combination. Furthermore, the predicate calculatedOverallRating is no longer considered. Note that only predicates that evaluate to true are listed in the states as in the original paper.

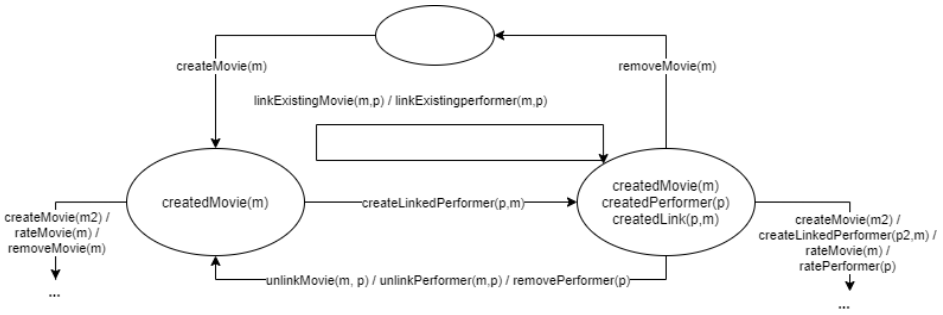

Figure 3.2: The use case transition system

After applying an instantiated use case in the transition system (in case the precondition of the contract was fulfilled) the simulation state is updated according to the contracts' postcondition.

Depending on the selected coverage criterion, we receive different test objectives as correct sequences of use cases. The robustness criterion was not considered in this example, but its application is coherent to the functional coverage criteria. How many test objectives are derived depends on the internal implementation of UC-System and cannot be predicted for this example. Let's assume that one test objective is the test path createMovie(m) -> createLinkedPerformer(p,m) -> removeMovie(m). Then the use case scenarios from Figure 3.3 are used to replace the use cases in the test objectives. It helps to specify the exchange of messages involved between the environment and the system.

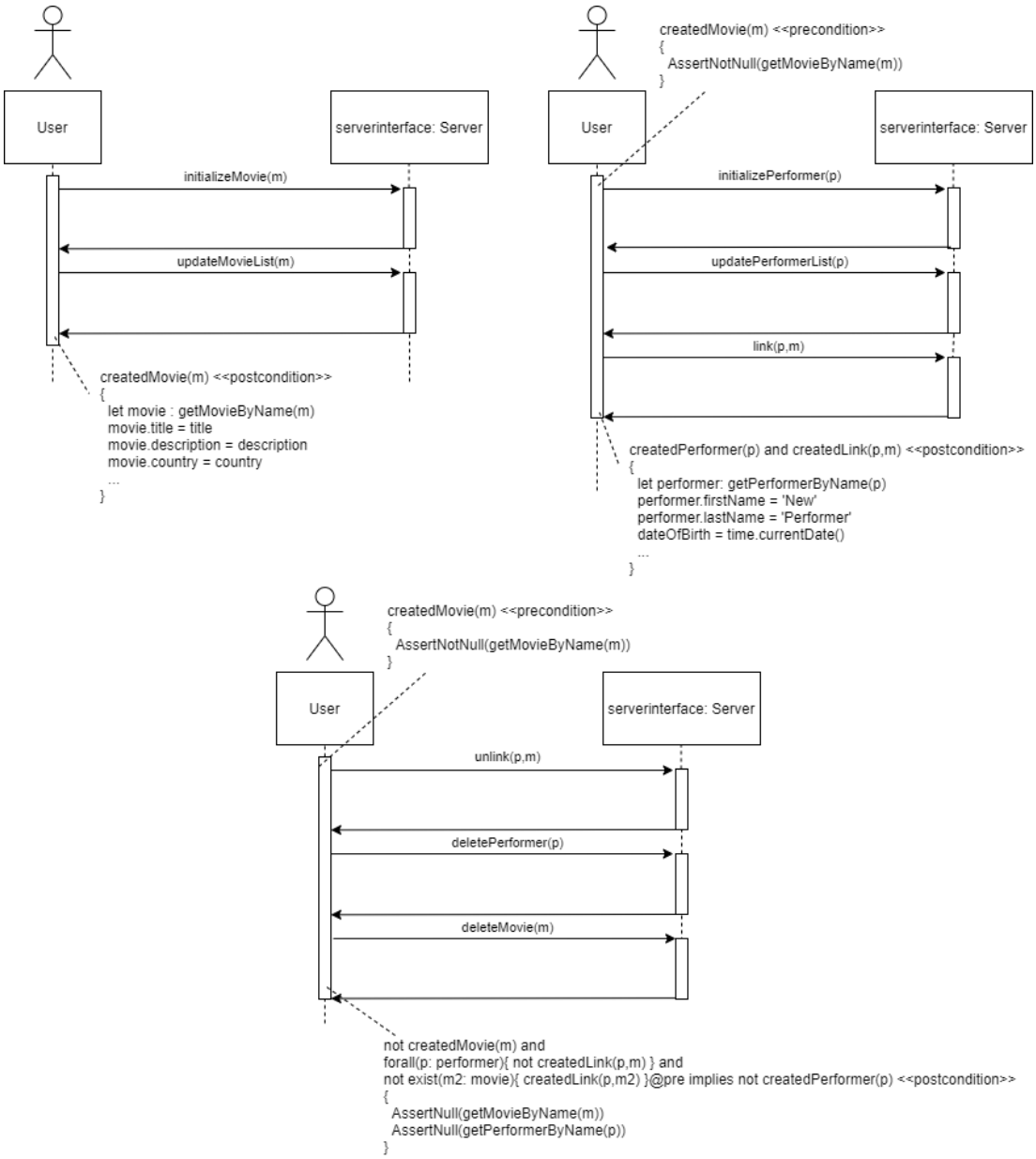

Figure 3.3: The use case scenarios

Note that the use case scenarios may still be incomplete for the execution. They contain the main messages exchanged between the tester and the SUT and say how the system has to be simulated to perform a use case and how to react to the simulation. To know how the system has to be simulated, the use case scenarios contain more detailed contracts written in OCL besides the contracts written as logical expressions that were provided by the use cases.

The prototype-tool UC-SCSystem uses the shown implementation in the use case scenarios to derive executable test scenarios as JUnit tests.

3.4 An Automated Approach to System Testing based on Scenarios and Operations Contracts

3.4.1 Description

Based on the suggested improvements in the first paper, the second paper uses a more graphical approach using *interaction overview diagrams* (IOD), a special form of activity diagram used to show control flow, to derive test paths. It helps to start testing in the early stages of development. Each node in the IOD represents either an interaction diagram (sequence diagrams) or interaction occurrences that show an *operation* invocation. Every IOD corresponds to one use case. The IODs get enhanced with contracts written in the *object constraint language* (OCL) and they get transformed into a contracts transition system (CTS) which models all scenarios of the IOD. Here the states are represented by the contracts and the transitions by the operations (interaction diagrams or interaction occurrences) in the IOD. The CTS is built by a tool using the *XML metadata interchange* (XMI) file for the IOD and the operation contracts in OCL as inputs. A state is created against each precondition and each postcondition of the operations. Logical if-then-else conditions are resolved by combining their testing condition with the result, therefore two different sub-states are created. The surrounding postcondition is then a composition of the two sub-states. Additional transitions are added for all conditional flows leading to alternative scenarios and their guard conditions are attached to them. Additional CTS flows help to further refine the requirements by spotting potential unwanted behavior or underspecification. The CTS is often visualized in a matrix.

Through traversing the CTS test paths get derived. Therefore different coverage criteria are defined. The simplest coverage criterion is the *state coverage* criterion, which generates test paths until all states of the CTS are covered. The criterion is already covered by the wider *transition coverage* criterion, which makes sure that all transitions are traversed before stopping the test path generation. The most expensive criterion is the *transition pair coverage* criterion, each possible transition pair needs to be covered in the test path generation process. It was detected, that the transition coverage criterion delivers a reasonable amount of test paths, but is not suitable for fault detection every time (compared to the transition pair coverage criterion which scores best in this task).

Besides using a graphical approach with IODs instead of using a formal requirement level language, the key difference to the first paper is that test scenarios do not get generated on system-level but rather on use case level due to the fact that contracts are not attached to use cases (which can be compared to a complete IOD) but rather to all operations within the IODs. It therefore serves as a platform to generate more in-depth test scenarios (as well as for negative test cases). The paper does not provide additional steps on how to generate executable test scenarios from the retrieved test paths.

3.4.2 Application

The second approach differs from the first one as this time a transition system is built on a concrete use case, in our case the use case to unlink a movie from a performer. Input to the approach in this paper is the IOD (Figure 3.4) with separate contracts defined in an OCL file (Listing 3.2). IODs are a special form of activity diagrams used to show the control flow. The nodes in our case are UML sequence diagrams and define the operations of the CTS. The states are represented by the contracts themselves.

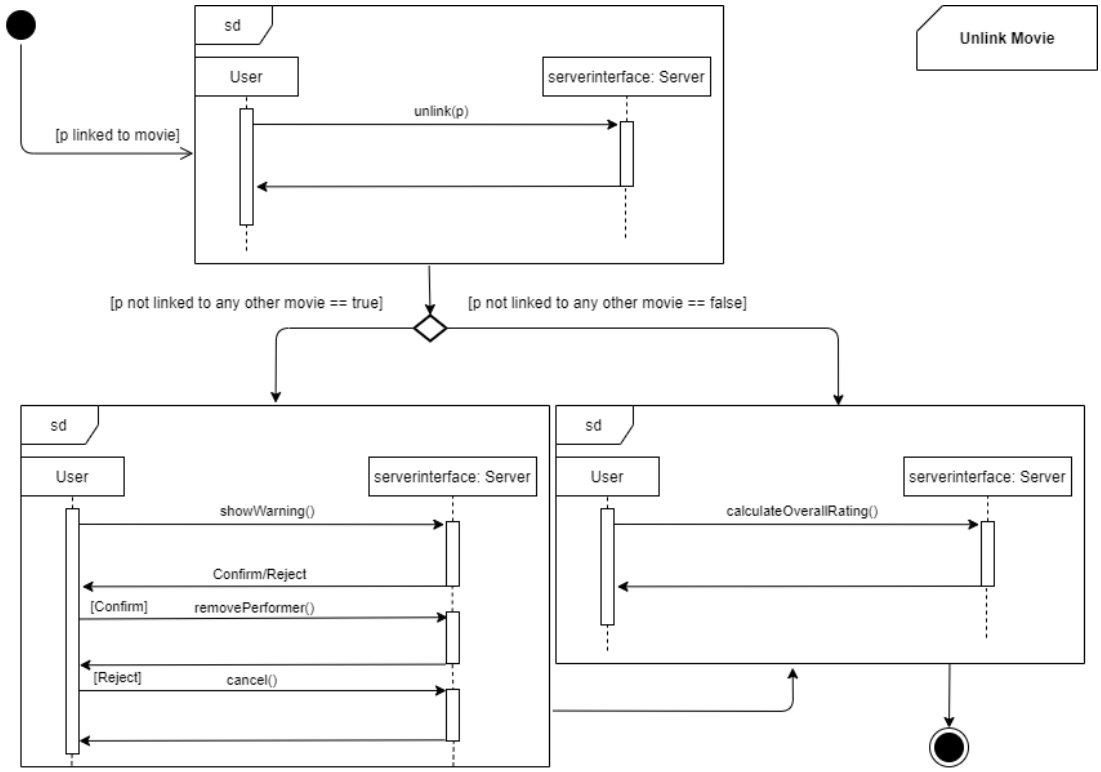

Figure 3.4: The interaction overview diagram

Listing 3.2: Contracts written in OCL

```
context Movie::unlink(performer)
pre  self.performers[performer] -> not isEmpty()
post self.performers[performer] -> isEmpty()

context MovieManager::removePerformer(performer)
pre forAll(movie | movie.performers[performer] -> isEmpty())
post self.performers[performer] -> isEmpty()

context Movie::calculateOverallRating()
post self.overallRating = 0.5 * (self.mean(self.performers.getRatings()) +
 self.rating)
```

Based on the IOD and the specified contracts the CTS matrix gets defined and leads to the CTS shown in Figure 3.5.

Operations	Pre	Post	Composite States
O_1	S_0	S_1 OR S_2	A
O_2	S_1	S_3 OR S_4	B
O_3	S_3 OR S_4	S_2	
O_3	S_2	S_5	

The operations (sequence diagrams in the IOD) can be thought of as the transitions in the CTS, whereas the states match the specified postconditions (maybe thought of as starting points of arrows).

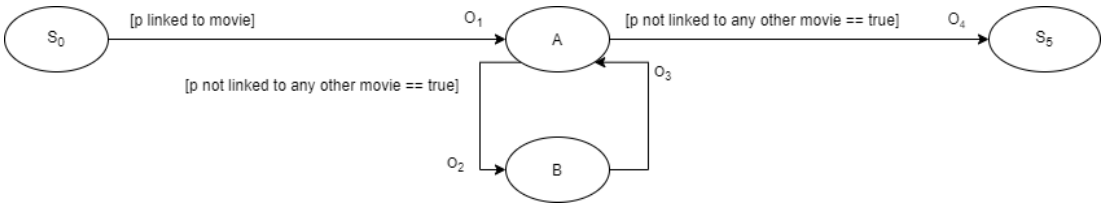

Figure 3.5: Contracts Transition System

For instance, the state S_0 describes the state when a performer p is linked to the given movie. One can then apply the operation O_1 to unlink the performer, resulting in the composite state A where either the performer is not linked to any other movie S_1 or the performer is still linked to another movie S_2.

Based on a coverage criterion the test paths get derived. Different from the first approach no test scenarios get generated. The paper only shows a new more low-level, graphical approach to generate test paths as this was even a suggested improvement from the authors of the first paper.

3.5 Comparison

To better compare the two approaches a synthesis matrix based on the following questions is provided:

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) What 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 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?

Nr.	Approach [19]	Approach [20]
1a	Use cases describing the basic operations in the transition system. Contracts that are attached to the use cases describing the states in the transition system. A contract consists of pre- and postconditions that specify the system properties to make a use case applicable and which properties are acquired by the system after its application. Parameters to contracts are actors and main concepts of the use case. The transition system (UCTS) itself as a simulation model to derive test objectives. The states in the UCTS are given through the predicates defined in the contracts, the transitions are triggered when applying an instantiated use case. Test objectives describing the test paths. Test objectives are derived by traversing the UCTS using a specific coverage criteria. Use case scenarios to build test scenarios from test objectives. Use case scenarios contain the main messages exchanged between the tester and the SUT, they define how the system has to be simulated to perform a use case and how to react to the simulation.	IODs holding all scenarios and operations of a use case. Operations can either be interaction uses or sequence diagrams inside the IOD and define the state transitions. Contracts written in OCL that are attached to the operations describing the states in the transition system. The CTS describing the transition system. Test paths derived from the CTS using coverage criteria.
1b	Use cases, contracts written as logical expressions, use case scenarios (sequence diagrams), initial system state, selected coverage criterion, and additional use case scenario parameters.	IODs, contracts written in OCL, selected coverage criterion, possibly manual resolving of conflicts in the CTS matrix.

1c	To express the ordering constraints between use cases, each use case is attached by a contract. To simulate the use cases, the set of formal parameters of the contracts are replaced with all possible combinations of their actual values. The use cases are then called *instantiated*. To apply an instantiated use case the precondition of its contract must match with the current simulation state. Afterwards the simulation state is updated according to the postcondition of the use case. Exhaustively simulating the system results in the UCTS. To derive test objectives the transition system is traversed according to one of the predefined coverage criteria AE, AV, AIUC, AV-AIUC, or APT. To build test scenarios a use case scenario can replace a use case at a certain stage of execution if the state reached at this stage locally implies the precondition of the use case scenario. Executable test scenarios are generated by the prototype-tool UC-SCSystem.	Each operation in the IOD was enriched with its own contract written in OCL. To build the CTS, first all operations have to be identified from the IOD. The operations are taken from the sequence diagrams or from other operations expressed as interaction occurrences in the IOD. Using the contracts of operations, the states for the CTS are identified. Eventually, conflicts in the CTS have to be resolved such as logical if-then-else conditions, equal contract statements, or join nodes. After the CTS was built, test paths are derived from the CTS by applying a coverage criterion, which is either state-, transition- or transition pair coverage.
2a	Automatic test generation from use cases and use case scenarios. Requirement validation by identifying inconsistencies, underspecifications, and invariants.	Deriving test paths from IODs. Further requirement validation on use case level.
2b	Test writers / Developers, Requirement Engineers.	Test writers / Developers, Requirement Engineers.
2c	Software Requirements (definition of a software requirement, functional requirements, acceptance tests), Software Testing (model-based techniques, objectives of testing, evaluation of the tests performed), Software Engineering Models and Methods (preconditions, postconditions and invariants, behavioral modeling, analysis for consistency, and correctness, traceability).	Software Requirements (definition of a software requirement, functional requirements, acceptance tests), Software Testing (model-based techniques), Software Engineering Models and Methods (preconditions, postconditions and invariants, behavioral modeling, analysis for consistency and correctness).
3a	Dedicated editor to design use cases with contracts, UC-System to build the UCTS simulation model, and to derive test objectives from it. UC-SCSystem to exchange the use cases by use case scenarios in order to build the executable test case scenarios.	UML 2.0 as a standard for IODs, OCL as a formal language to write the contracts, prototype tool to derive test paths.
3b	Writing the use cases and contracts is supported by a dedicated editor, but has to be done manually. Deriving test objectives from use cases and contracts through the transition system is done automatically by UC-System. Use case scenarios have to be specified manually, the generation of test scenarios works semi-automatically with UC-SCSystem as it may need additional parameters from the tester.	Only the IOD and contract specification has to be done manually, the complete approach was then automized by a prototype tool.

4a	The approach was evaluated by looking at the statement coverage of three sample programs and the efficiency of test case scenario generation.	The approach was evaluated by looking at the number of test paths generated to cover all success scenarios and fault detections.
4b	Code coverage with the most coverage criteria was around 80%. The coverage criteria differ in efficiency. AE, AV, and AV-AIUC perform with low efficiency, the sets of test cases are larger than in AIUC and APT. APT reaches 100% functional test coverage with only 15 test case scenarios. Testing robustness leads to a high number of generated test case scenarios that only cover about 50% of the corresponding code. The approach is good for functional testing, but bad for robustness testing.	Using the transition criterion to derive test paths leads to a reasonable amount of test paths and covers all alternative flows in the IOD, but is not suitable for fault detection at any time. The transition pair coverage criterion guarantees the maximum fault detection, but leads to a high amount of test paths. State coverage captures all success scenarios.

Both approaches have a transition system as a core component in common. Both use contracts to define the states in the transition system. A transition can be applied in case the precondition is fulfilled, the state is then updated according to the postcondition. The transition system is used to derive test paths. Furthermore, both approaches define coverage criteria to generate a certain amount of test cases from the transition system. Both have the AE/transition coverage criterion and AV/state coverage criterion in common. Using this core idea it is also easy in both approaches for requirement engineers to validate their specified requirements during the simulation of the transition system. Both papers draw a similar conclusion, that the core approach is helpful to generate functional test case scenarios for the requirements, nevertheless, there is a lack to achieve a comparable coverage for robustness code for fault detection.

Nevertheless, the two approaches differ in their case of application. While [19] uses use cases to automatically generate system-level tests, [20] restricts to generate test paths for a specific use case. It therefore does not need additional use case scenarios as an input like [19] to generate test scenarios as these are already natively given in the IOD.

Besides the case of application, the input parameters in both approaches are different as well. [19] uses requirement-level first-order logical expressions to describe use cases and their contracts. A more graphical approach was chosen in [20] by specifying a use case with an IOD instead of using a formal language.

[19] has the special characteristic that it demonstrates an almost fully automized end-to-end test scenario generation process until concrete test execution, which is missing in [20]. Nevertheless, this could easily be implemented in [20] as well using similar tool support as in [19].

3.6 Conclusion

Testing with a transition system delivers a method on how to bring requirement specification and system test generation closer together, eliminating traceability problems between what was specified and what was implemented. Requirements can easily be validated on their consistency, correctness, and on possible underspecification while at the same time test paths can be derived through traversing the transition system.

In literature search, a second paper besides the given one was found that follows the process of automatically generating test paths through traversing a transition system, but this time uses the in [19] proposed extension of IODs as a form of activity diagrams instead of use cases as its input besides contracts. To find this article both snowballing and search-term-based techniques were used, whereas the choice of relevant articles was based on previously specified relevance criteria. Eight resulting papers were found according to these criteria, [20] was finally chosen.

Approach [19] almost shows a fully automated way to derive executable test scenarios as JUnit tests. Contract enriched use cases using a formal language based on requirement-level first-order logical expressions are used as an input to the simulation model. Through exhaustive simulation, a transition system is build from which test objectives are derived using coverage criteria. It was empirically proven that the AIUC and APT criterion perform the most efficient and the most extensive measured by statement coverage. Giving contract enriched use case scenarios as a second input helps to generate concrete executable test scenarios.

[20] switches from system level to use case level test generation. It uses contract enriched IODs as graphical input and builds a transition system on it. Test paths are generated using the transition coverage criterion. No further methodology was introduced to generate executable test cases, but as the approach is already specified on use case level, the generation of test code is self-explanatory (a similar tool like UC-SCSystem from [19] can be used).

One weakness of both approaches remains the capability to generate a sufficient amount of robustness tests for fault detection. Both approaches only rely on contract violations and do not sufficiently cover data variations as test path generation is only based on one initial state chosen by the test creator. Still, for the automatic generation of functional test scenarios, both approaches show industry-standard and highly applicable solutions.

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 ACM[1] only showed a few. Because the given article [4] can also be found on IEEE Xplore[2], 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 UML diagram, which can be seen in Figure 9.1 below.

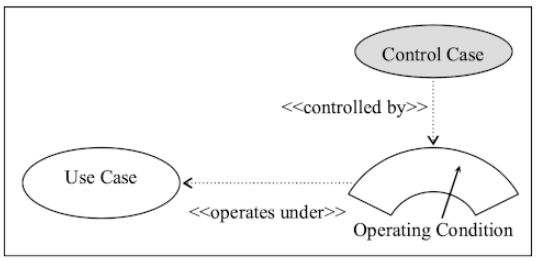

Figure 9.1: Association in Use Cases Modelling [4]

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.

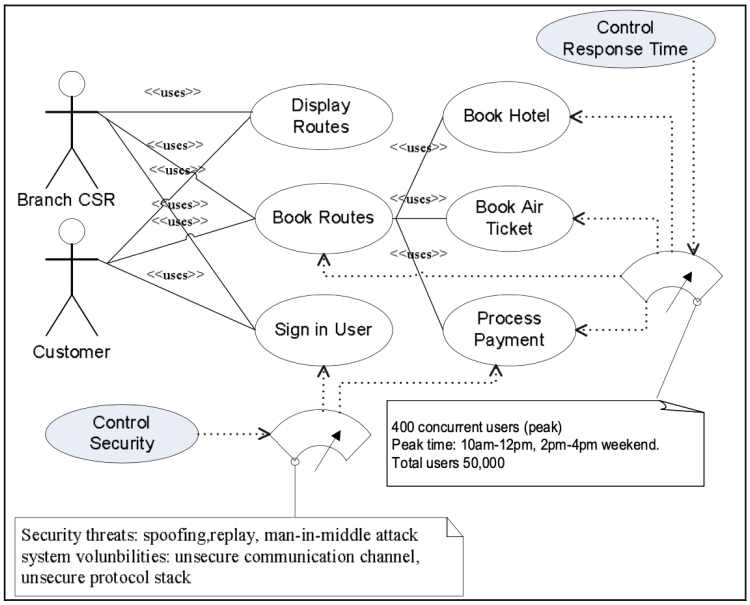

Figure 9.2: Use Case Model with Control Cases [4]

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 53.

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 54.

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: 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

[3]From 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: 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: 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 UML 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.

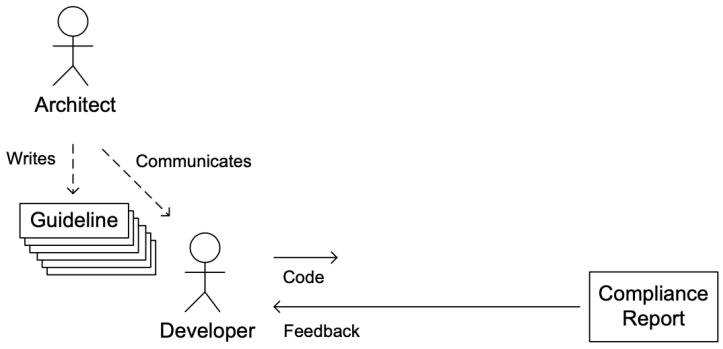

Figure 9.3: The common way of communicating architectural requirements [5]

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.

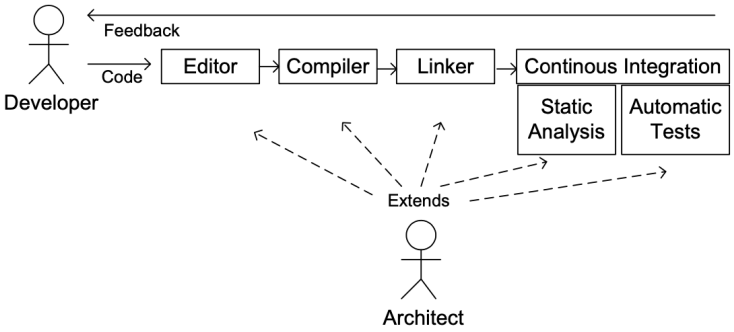

Figure 9.4: Suggested solution of communicating requirements [5]

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 56, 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	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. *Implementation in pseudo code* `for each $variable in $source:` ` if $variable.startsWith("imdb") then` `  if typeof($variable) != "ImdbId" then` `   throw new Exception("Wrong class")`

2	Exported strings are escaped	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 NFR, 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. `EscapedString`. Only this string class may be used in the HTML exporter. *Implementation in pseudo code* ``` for each $functionCall in $HTMLExporter: if $functionCall == "writeText" then $arg = argument of($functionCall); if typeof($arg) != "EscapedString" then throw new Exception("Wrong class") ``` *Note:* We assume that `writeText` 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.

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]	Approach 2 [5]
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.	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.

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.	The guidelines cover code architecture. They must be transformable into automated tests (e.g. by a static code analyzer).
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.	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.
2a	Early modeling of non-functional requirements. Being able to control requirements throughout the whole software development life cycle.	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.
2b	Management, Requirements Engineer, Developers, Testers	Developer / test writer, Software Architect
2c	Software Requirements (functional and non-functional requirements, acceptance tests), Software Testing (model based techniques)	Software Testing (Software Testing Tools, Test Techniques), Software Maintenance (Software Maintenance Tools)
3a	No tool support for generating "Control Case"-Boxes and other artifacts	Existing static code analysis tools (e.g. linters), which can be extended by further tests.
3b	No automation is done. Automation is only proposed as another step which can be implemented, e.g. through code generation with SysML.	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.
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.	No evaluation was performed. The conclusion, i.e. success of the approach, is based on personal experience only.

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

10.1 Introduction

Software testing ensures that specified requirements, functional and non-functional, are met by the implementation. Non-functional requirements are especially hard to test because they do not describe what the software does, but how and to what extend of quality it does it. Hence the also customary term quality requirements. They pose numerous challenges for software testing and software quality due to their system-wide effects and often crosscutting nature. They do not only concern one part of the software and its source code, but multiple or even the entire system. For instance, the memory requirement: "The system only uses two gigabytes of main memory at most at all times." has a restrictive influence on other requirements and system functionality like performance requirements. This impedes the observance of the widely propagated principle of separation of concerns, introduced by Parnas [34] and Dijkstra [30], throughout development and testing.

By virtue of the afore-mentioned problems, there are only few tools and systematic approaches for testing non-functional requirements. Frequently, they are evaluated subjectively, resulting in a loss of traceability between test code, source code and requirements. Aspect orientation is a technique that can be harnessed for testing non-functional requirements. It is a programming paradigm, aiming to modularise such requirements (in the context of AOP, they are called system-level concerns in contrast to core-level concerns), similar to the modularisation of functional requirements using objects. An aspect modularises a system-level concern and thereby represents a system-wide functionality, accessible to multiple classes and other parts of the software. Let us consider a common application for AOP: Listing 10.1 shows a simple logging aspect in C++. It consists of three main components:

- Joinpoint: The point in the program code, the aspect is executed. It can be before, after or around something (here: before).
- Advice: Associates the joinpoint with an activity (here: a printf statement).
- Pointcut: Selects a suitable joinpoint out of the set of all joinpoints and associates it with a method or function (here: %::%()).

Listing 10.1: **Logging Aspect in C++.**

```
aspect LoggingAspect{
   public:
      pointcut logMethods() = call("% %::%()");
      advice logMethods() : before(){
         printf("> Enter: %s\n", JoinPoint::signature());
      }
};
```

In this example, the printf logging statement is executed each time before the %::%() method is called. Based on these principles, many possibilities for systematic and automated testing are conceivable.

In the following, the execution and results of a literature search based on a given article are presented in Section 10.2. Subsequently, the given article , which assesses the use of aspects for testing non-functional requirements as well as the suitability of certain non-functional requirements for aspect-oriented testing is described and applied to an example in Section 10.3. Likewise, for a selected article found using the literature search in Section 10.4. It expands upon the basic ideas of the first article and partially automates the creation of test aspects. Thereafter, the results of a literature comparison are presented in Section 10.5. Finally, in Section 10.6 both approaches as well as the general concept of testing using aspects are evaluated.

10.2 Literature Search

To find relevant literature, a literature research based on the following search question was conducted: ***Which approaches for systematic creation of tests (for non-functional requirements) using aspects exists?*** Both forward and backward snowballing proceeding from the given article as well as a term-based search were carried out. The used terms were test, aspects, AOP and aspect-oriented. Restrictions to reduce the number of hits are described in Table 10.1. First, the upper term from Table 10.1 with everything in the publication title was used, then the lower term from Table 10.1 with test in the title, aspects in the abstract and AOP and aspect-oriented in all meta data. Source platforms for the search were IEEE and ACM. IEEE because the given article as well as articles referencing it can be found here. ACM because there are many available and peer-reviewed articles differing from IEEE. Content-based relevance criteria were derived directly from the search question:

- The article must cover the creation of tests using aspects because this is the main topic of the given article. Furthermore, the criterion is used to exclude articles covering testing of aspect-oriented software with conventional methods.
- The article must describe systematic approaches for the creation of tests, since this is the superordinate topic of the seminar.
- The article must describe test methods for non-functional requirements, as this is the subtopic of the given article.

In addition to these content-based criteria, there were some rather soft criteria:

- The article must be from different authors.
- The article must be written in English or German language.

Table 10.1: **Search Terms, Restrictions and Sources.**

Term	**Restrictions**	**Sources**
test AND (**aspects** OR **AOP** OR **aspect-oriented**)	• all in title	• IEEE • ACM
test AND **aspects** AND **AOP** AND **aspect-oriented**	• test in title • aspects in abstract	• IEEE • ACM

Table 10.2 shows the execution and results of the literature search. For further selection, the relevant articles were divided into three categories:

- overviews of testing using aspects,
- examples for testing using aspects,
- improvement or monitoring of conventional (unit) test using aspects.

The given article can be classified between overview and example. Via the literature search, I wanted to find an article providing automation and tool support, seen as the given article has shortcomings in that regard. Moreover, all relevance criteria must be fulfilled and it would be beneficial if the article covers at least two out of three classification categories.

Many articles violate the first criteria, because they cover the testing of aspect-oriented software without using aspects. Some articles were not chosen because they just cover one classification category, therefore not really adding new information and approaches.

The article that stood out was Duclos et al.: "ACRE: An Automated Aspect Creator for Testing C++ Applications" [31] because it covers all categories and satisfies all criteria. It includes an extensive general section followed by the description of an approach for automated creation of test aspects on an example to improve or replace conventional tests. Since the article was found using forward snowballing, it directly references the given article and proposes solutions for problems the latter raises. This article was selected.

Table 10.2: **Literature Research Documentation.**

Source	Date	Restrictions	Term	Results	Relevant	Used	Comments
IEEE*	11.11.2020	none	forward snowballing	10	5	∇	-
IEEE*	11.11.2020	none	backward snowballing	22	5	none	-
IEEE*	11.11.2020	none	"All Metadata":test AND ("All Metadata": aspects OR "All Metadata":AOP OR "All Metadata":aspect-oriented)	220,605	?	none	too general, not considered
ACM†	11.11.2020	none	[All: test] AND [[All: aspects] OR [All: aop] OR [All: aspect-oriented]]	172,120	?	none	too general, not considered
IEEE*	11.11.2020	title	"Document Title":test AND ("Document Title": aspects OR "Document Title":AOP OR "Document Title":aspect-oriented)	158	11	none	first 50 considered
ACM†	12.11.2020	title	[Publication Title: test] AND [[Publication Title: aspects] OR [Publication Title: aop] OR [Publication Title: aspect-oriented]]	311	4	none	first 50 considered
IEEE*	12.11.2020	test: title, aspects: abstract	((("Document Title":test) AND "Abstract":aspects) AND "All Metadata":AOP) AND "All Metadata":aspect-oriented	33	16	none	-
ACM†	12.11.2020	test: title, aspects: abstract	[Publication Title: test] AND [Abstract: aspects] AND [All: aop] AND [All: aspect-oriented]	31	6	none	-

**: https://ieeexplore.ieee.org/Xplore/home.jsp †: https://dl.acm.org/ ∇: [31]*

10.3 Metsä et al.: "Testing Non-Functional Requirements with Aspects: An Industrial Case Study"

10.3.1 Description

The research article "Testing Non-Functional Requirements with Aspects: An Industrial Case Study" by Jani Metsä of Nokia in association with Mika Katara and Tommi Mikkonen [33] of Tampere University of Technology was published in 2007 as part of the Seventh International Conference on Quality Software. According to the authors, the main goal of software testing is to ensure that requirements are met by the implementation. There are already several systematic approaches for testing functional requirements, but only few for testing non-functional requirements. To tackle this problem, they turned to aspect-oriented programming as a potential testing technique. In their paper, they try to answer the following research questions:

1. To what extent can aspect-oriented techniques be harnessed to test non-functional requirements?

2. Which non-functional requirements lend themselves for testing with aspects?

The authors conducted a case study, analysing 150 requirements of an existing industrial embedded system (a quality verification software for mobile phones running Symbian OS). They were able to identify 16 crosscutting non-functional requirements for which seven test aspects were formulated and implemented. In detail, in the first step, a set of provided system requirements or characteristics are analysed and corresponding non-functional requirements (which might be crosscutting) derived. One non-functional requirement is derived from one or more system requirements, a system requirement can comprise multiple non-functional requirements. In a second step, the non-functional requirements are categorised (performance, robustness, ...), and corresponding testing objectives are derived. One testing objective is derived from one or more non-functional requirements. Finally, the test aspects can be formulated based on the testing objectives and non-functional requirements categories. One test aspect can comprise multiple testing objectives.

The approach proved to be easy and test coverage could be increased significantly. The use of aspects enables non-invasive testing throughout the software lifecycle. Furthermore, separation of (test) concerns can be achieved by modularizing crosscutting non-functional requirements. As a result, maintainability, reusability as well as tracing between requirements and test code can be improved. Based on the experiences gained, the authors concluded that especially system-wide and crosscutting non-functional requirements, such as security, performance, reliability and robustness, are suitable for aspect-oriented testing. To sum up, the article presents a systematic approach – a systematology - to derive testing objectives and test cases with related test aspects from non-functional requirements. It does not provide automation for any step, hence the main problems the authors identify: the lack of tool support and the need for testing personnel to be firm with aspect-oriented programming.

10.3.2 Application

The non-functional requirements (REQ) are derived from the user task or the corresponding system characteristics (Table 10.4). They have system-wide effects (hence affect multiple system-functions) and are of crosscutting nature. For example, REQ5 and REQ6 set performance constraints on REQ7 and REQ8 as well as REQ3; REQ3 and REQ4 set reliability and robustness constraints on each other; REQ9 sets security constraints on REQ1 and REQ2.

Subsequently, the non-functional requirements are categorised and corresponding test objectives can be derived (Table 10.5).

Finally, a test aspect for one or multiple requirement categories is formulated (Table 10.6). These test aspects would then need to be implemented manually using an AOP-Framework (AspectC++ [29], AspectJ [32], ...). For instance, the Memory Aspect could work like this: every time the constructor or destructor of an entity class (Movie, Performer, ...) is called, a counter will be incremented or decremented by the size of the class.

Table 10.4: **Initial System Requirements of the Movie Manager App.**

System Characteristic	**Derived Requirement**
Data is managed consistently by the system.	REQ1: Default values are provided (wherever possible). REQ2: Entities are linked consistently (a performer must always be linked to at least one movie).
The system is fault tolerant and able to report faulty behaviour.	REQ3: The system can recover from hang situations. REQ4: The system can identify correct and incorrect system behaviour.
The system runs on a mobile phone with limited amount of resources and thus has a strict memory footprint.	REQ5: The system must occupy at maximum W bytes of ROM REQ6: The system must occupy at maximum X bytes of RAM.
The system is fast and responsive.	REQ7: The system can respond to requests after Y time units after power-up. REQ8: All user requests are handled in Z time units.
The system might hold sensitive data and is therefore secure.	REQ9: The system follows the Android App security best practices (e.g. a password is required for sensitive data changes).

Table 10.5: **Testing Objectives for Non-Functional Requirements.**

REQ	**Testing Objective**	**Requirement Category**
REQ1 REQ2	TO1: Supervise data consistency and integrity.	Security (Integrity).
REQ9	TO2: Check password protection.	Security
REQ3	TO3: Generate hang situation.	Robustness
REQ3 REQ4	TO4: Analyse system reliability.	Reliability
REQ5 REQ6	TO5: Supervise memory consumption.	Performance (Memory)
REQ7	TO6: Measure time consumed from power-on to the system being in a responsive state.	Performance
REQ7 REQ8	TO7: Measure time consumed on serving requests and executing system-functions.	Performance

Table 10.6: **Formulated Test Aspects.**

Test Aspect	**Description**	**Test Objective(s)**
Integrity Aspect	Checks if all specified default values are provided and performers are linked to at least one movie.	TO1
Security Aspect	Tries to execute all data changing system-functions without providing the password. It should ‘t be possible to commit the changes.	TO1, TO2
Robustness Aspect	Generates a request jam to test if the SUT can recover from the hang situation.	TO3
Reliability Aspect	Collects information on SUT states and failures.	TO4
Memory Aspect	Supervise memory consumption by tracking all memory allocations and deallocations.	TO5
Performance Aspect	Measure function execution times.	TO6, TO7

10.4 Duclos et al.: "ACRE: An Automated Aspect Creator for Testing C++ Applications"

10.4.1 Description

Etienne Duclos, Sébastien Le Digabel, Yann-Gaël Guéhéneuc and Bram Adams [31] of École Polytechnique de Montréal published their article "ACRE: An Automated Aspect Creator for Testing C++ Applications" in 2013 as part of the 17th European Conference on Software Maintenance and Reengineering. They state that software should be faultless and in accordance with requirements yet testing costs need to be acceptable. Systematic and developer-friendly tools for testing functional and non-functional requirements are required to achieve this goal. However, such tools are sparce, especially for non-functional requirements. The authors review related approaches for testing with aspects, including the approach by Metsä et al., and conclude that high-level tool support and automation are required to solve the problems raised in these publications. To achieve this, they try to answer the following tripartite research question:

Can automatically generated test aspects be used to ...

1. ... detect memory leaks?

2. ... test invariants?

3. ... detect interference bugs?

The authors present ACRE (automated aspect creator), a tool that automatically generates test aspect code using a domain specific language. Provided a set of test cases or objectives or a bug report, the DSL statement describing the test aspect can be derived. The type of the bug or the category of the underlying NFR of the test case determines the type of the test aspect that must be chosen in this step. Afterwards, the test aspect is generated automatically based on the DSL description. ACRE takes the entire source code as an input and looks for DSL statements. They are parsed to generate the corresponding test aspects.

Using ACRE, the authors were able to detect one error of each type (1. – 3.) in the mathematical optimisation software NOMAD. Thanks to the DSL description of the test aspect, testing personnel does not need to have extensive knowledge about aspect-oriented programming, just about the DSL syntax. Although the creation of test cases from requirements or bug reports is not automated, the DSL and the types of available test aspects provide support in that regard. However, this means that the approach is currently limited to specific use cases (memory leaks, invariants, interference bugs in C++ applications). Nevertheless, it would be easy to extend the available functionalities and transfer the approach to other programming languages with support for aspect-oriented programming.

To sum up, the article presents a tool for the automatic generation of test aspects with given test cases formulated in a domain specific language.

10.4.2 Application

The non-functional requirements with corresponding test objectives or a bug report must be given, as the approach does not present a way to derive them. Let us consider the same requirements (REQ1-9) and test objectives (TO1-7) as in Table 10.4 and Table 10.5. Furthermore, a user of the Movie Manager App has filed the following bug report:

Table 10.7: **Bug Report.**

Summary: Memory Leak when removing all movies of a performer.
Description: Deleting all Movies linked to a performer leads to the removal of the performer from the performers list, but no memory* is not actually freed.
Steps to reproduce: 1. Consider a performer with one linked movie. 2. Delete the linked Movie.
Actual results: The performer and movie are removed from the respective lists, but no memory is freed.
Expected results: The performer and movie are removed from the respective lists and both objects are no longer in memory.

*(*For the purpose of this example, let us consider all data is kept in main memory.)*

Based on the given test objectives or the bug report, the test aspects must be formulated manually using a domain-specific language. However, the DSL provides certain types of Aspects (Counter, Logging, Timing, Checking) and guidelines that help formulate the test cases. Table 10.8 shows the aspect types corresponding to the given test objectives and test aspects of approach 1. The DSL formulation to test for memory leaks in the Performer class could look like this:

Listing 10.2: **DSL Statement For Counter Aspect**

```
// // name: MemoryAspect
// // type: Counter
// // className: Performer
// // namespace: MovieManager
```

The parameters *'name'* and *'type'* are required and determine the name and type of the test aspect. 'className' and 'namespace' are optional and specify, which class are concerned by the aspect. By default, the file in which the DSL statement was found is searched for namespaces and the name of the file is considered to be the name of the class. It must be placed somewhere within the source code, for example above or below the tested function or class or in a separate file. Afterwards, ACRE takes the entire source code as an input, looks for DSL statements (always starting with // //) and generates the corresponding test aspects automatically. For the example above, ACRE generates the memory aspect for the Performer class as shown in Listing 10.3. In essence, the counter aspect initialises a static counter variable that is incremented each time the Performer constructor is called and decremented each time the destructor is called. After the execution of the main method, the final count as well as a warning in case of a memory leak (counter > 0) are printed. In this manner, a class causing a memory leak can be detected. The timing and logging aspects are quite similar to the counter aspect. The checking aspect is more complex and allows the declaration of variables as well as the use of for-loops, while-loops and if-clauses. Note that the timing aspect in its current form is used for interference bug testing, hence it measures access times to variables. However, a timing aspect for measuring

function call times could easily be implemented by starting a timer before each function call and stopping it after each function call in the advice of the aspect.

Table 10.8: **Aspect Types for Given Test Objectives.**

Test Aspect	**Description**	**Test Objective(s)**	**Aspect Type**
Integrity Aspect	Checks if all specified default values are provided and performers are linked to at least one movie.	TO1	Checking
Security Aspect	Tries to execute all data changing system-functions without providing the password. It should 't be possible to commit the changes.	TO1, TO2	Checking
Robustness Aspect	Generates a request jam to test if the SUT can recover from the hang situation.	TO3	Checking, Logging
Reliability Aspect	Collects information on SUT states and failures.	TO4	Logging
Memory Aspect	Supervise memory consumption by tracking all memory allocations and deallocations.	TO5	Counting
Performance Aspect	Measure function execution times.	TO6, TO7	Timing (not in current form)

Listing 10.3: **Generated Counter Aspect Code.**

```
aspect MemoryAspect{
   public static int _Eval_Performer = 0;

   pointcut Eval_Performer() = "MovieManager::Performer";
   advice Eval_Performer() : slice struct{
      class Eval_Performer{
         public:
            // constructor -> increment
            Eval_Performer(){
               MemoryAspect::_EvalPerformer++;
            }
            // destructor -> decrement
            ~Eval_Performer(){
               MemoryAspect::_EvalPerformer--;
            }
      }
   };

   // print counter (and warning)
   advice execution (main(...)) : after () {
      printf("Final count of Eval_Performer: \%d\n",  _Eval_Performer)
      if(_Eval_Performer > 0)
         printf("Memory Leak! \n")
   }
};
```

10.5 Comparison

The approaches have been compared using a set of synthesis questions, as shown in Table 10.9.

Both approaches offer the same benefits: using test aspects, it is possible to test system-wide crosscutting non-functional requirements in a non-invasive way (without modifying or instrumentalising the source code). As a result, modularity, maintainability reusability as well as traceability of requirements to corresponding tests can be improved. Multiple stakeholders can benefit from both approaches, in particular testing and maintenance personnel. Correspondingly, both approaches use and produce similar artifacts: test objectives are derived from requirements and ultimately implemented using aspects.

However, there are differences concerning the specific individual steps, the level of automation as well as the relating quality of the approaches. Approach 1 describes a systematology to derive aspect test cases from requirements. These have to be implemented manually. In contrast, approach 2 assumes the test objectives to be provided and generates corresponding test aspects automatically. Subsequently, the most significant difference is the tool support: approach 1 does not provide any tool support, but rather describes a systematology. Approach 2 fully automates the final step of approach 1 – from test objectives to test code – and assists with the set-up of the concrete test cases through the structure of the domain specific language.

This is reflected in the evaluation of the respective approach as well. Both draw the conclusion that aspects are well suited for testing non-functional requirements. Approach 1 criticises the lack of tool support, approach 2 counteracts this exact problem. In Summary, the two approaches are quite similar. However, approach 1 is less extensive concerning the automation of the technique and should be regarded as a rather fundamental research. Approach 2 expands on the ideas of approach 1 and enhances their quality by partially automating the process.

Table 10.9: **Synthesis Matrix.**

Question	Name	Approach 1: **Testing Non-Functional Requirements with Aspects**	Approach 2: **ACRE**
3 **Description**	a) artefacts and relationship between artefacts	system requirements in natural language (provided), NFRs (derived), testing objectives (derived), test aspects (derived)	test objectives or bug report (provided), DSL-statements within the source code (derived), test aspects (generated)
	b) preconditions/ input	system requirements	test objectives or bug report
	c) steps, results, informations	Step 1: Provided a set of system requirements, NFRs (which might be crosscutting) are derived. One NFR is derived from one or more system requirements, a system requirement can comprise multiple NFRs. Step 2: The NFRs are categorised and corresponding testing objectives are derived. One testing objective is derived from one ore more NFRs. Step 3: Test aspects can be formulated based on the testing objectives and NFR categories. One test aspect can comprise multiple testing objectives.	Step 1: Provided a set of testing objectives or a bug report, the DSL statement describing the test aspect can be derived. The type of the bug or the category of the underlying NFR of the testing objective determines the type of the test aspect that must be chosen in this step. Step 2: The test aspect is generated automatically based on the DSL description.
4 **Benefits**	a) supported usage scenarios	Easy and non-invasive testing throughout the software lifecycle: during initial development for debugging or validating NFRs, during system testing and for maintaining tasks. Separation of concerns by modularizing crosscutting NFRs. Tracing of NFRs and corresponding tests	Easy and non-invasive testing throughout the software lifecycle: during initial development for debugging or validating NFRs, during system testing and for maintaining tasks. Separation of concerns by modularizing crosscutting NFRs. Tracing of NFRs and corresponding tests
	b) supported stakeholders	Developers, testing and maintenance personnel	Developers, testing and maintenance personnel
	c) corresponding SWEBOK-Knowledge Areas	Software Requirements (NFRs, Requirements Tracing), Software Testing (System Test), Software Maintenance	Software Requirements (NFRs, Requirements Tracing), Software Testing (System Test), Software Maintenance
5 **Tools**	a) tool support	none (except the aspects themselves)	ACRE (Automated Aspect Creator), DSL
	b) level of automation	none	Test cases and testing objectives must be derived manually from the requirements. The DSL facilitates the test case design. The test aspect code is generated automatically from the DSL statements.
6 **Quality**	a) evaluation	Case study with requirements of an existing industrial embedded system. Comparison of test coverage with and without test aspects.	An empirical study aiming to find errors in the mathematical optimisation software NOMAD was conducted. The approach was contrasted with other common techniques and tools.
	b) evaluation results	Test aspects proved to be an easy and non-invasive technique for testing NFRs. Seperation of (test) concerns by modularising NFRs. BUT: lack of tool support, complicated build process.	Automatically generated test aspects proved to be suitable for finding memory leaks, interference bugs and testing invariants. The approach is easy to use because developers do not need to know aspect-oriented programming itself, only the DSL syntax and semantics. BUT: derivation of test cases from requirements and DSL statement input still manual

10.6 Conclusion

In summary, this report presents two articles describing possible approaches to harness aspect-oriented techniques for testing non-functional requirements. A literature search based on the first article was carried out to obtain an overview of the current state of research and find relevant literature for this report. The selected article builds upon the ideas of the given article and presents a tool for (partially) automating the creation of test aspects. The two approaches have been compared and illustrated using a literature synthesis with a set of synthesis questions and a synthesis matrix as well as an application to an example.

The results of both research articles and the synthesis and example, conducted for the report, indicate that aspects are suitable for testing non-functional requirements in a systematic manner due to multiple reasons: Firstly, the basic functionality of aspects carries an inherent systematology and lends itself for testing. An aspect comprises a set of statements that can be executed every time before, after or around a function call, which can easily be utilised to check pre- and postconditions or to measure resource consumption and runtimes. In addition to that, they are non-invasive, meaning the tested source code does not need to be modified and the test aspect can easily be separated from the main code, reused or removed. Furthermore, test aspects are especially suitable for testing non-functional requirements because they modularise the system level concerns, non-functional requirements pose, thus maintaining separation of concerns. System-wide functionality, that is spread out through the entire system and its source code can be tested using one aspect. This also improves traceability between the non-functional requirements and the corresponding tests.

Metsä et al. point out that the lacking tool support and the need for developers to have a firm understanding of aspect-oriented programming pose possible challenges. Duclos et al. solve these problems (partially) by introducing ACRE, a tool that parses DSL statements to generate test aspects automatically. However, the tool only supports four aspect types and is limited to C++ applications. The lack of subsequent research articles (after 2013) indicates that systematic testing using aspects is not in focus of current research anymore. Nevertheless, existing approaches and tools, as presented in this report, should be further refined to make use of the aforementioned advantages of aspect-oriented testing, especially for non-functional requirements. Because aspect-oriented programming lost popularity in recent years, applying aspect-oriented testing techniques in practice is difficult because only few developers and testers are able to implement them. However, using a domain specific language as an intermediate layer of abstraction, as described by Duclos et al, tackles this problem and enables relevant stakeholders to use test aspects as small, re-usable and none-invasive test modules throughout the software lifecycle at the system test level.

11 Conclusion

This chapter presents first of all the most important insights from each of the individual chapters. For this, the conclusions from each topic are summarized. Then, the improved knowledge areas from the SWEBOK are listed and their use over the chapters is compared. At the end a final conclusion over all the topics is presented.

Chapter x (citation for topic 2: Acceptance testing with FitNesse) presents different approaches to create acceptance tests that can be automatically executed with the tool FitNesse. The literature search showed that this topic using the specific tool FitNesse is not popular in the research. The presented approaches are highly dependent on the experience and skill of the person executing the approach. Most of the steps are done manually and require human judgement. Therefore, the approaches are only recommended if the required artefacts are already part of the engineering process and a person with experience with these artefacts takes part in the engineering process.

Chapter x (citation for topic 3: Testing with a transition system) presents approaches to automatically derive test scenarios from means of the specification area using transition systems. This improves traceability between the specification and implementation and allows for an easy validation of requirements. The test cases can be derived by traversing the transition system. Both presented approaches do not generate a sufficient amount of robustness tests for fault detection and don not sufficiently cover data variations. However, the presented approaches are still recommended for the automatic generation of functional test scenarios.

Chapter x (citation for topic 4: Testing with a timing component) presents approaches to test real-time requirements. The literature search showed that this topic is not popular in the research. One of the presented approaches (citation) involves manually executed, intermediate steps. Therefore, this approach cannot be recommended in the presented form. The other approach (citation) executes all intermediate steps automatically and can be recommended for usage.

Chapter x (citation for topic 5: Testing with a classification tree) focuses on testing with a classification tree. This technique offers the possibility to reduce the number of test cases. The presented approach describes how requirements can be transformed into a classification tree for the input parameters of a system model. Another presented article shows that classification trees are, despite existing, more recent methods, still relevant.

Chapter x (citation for topic 7: Testing with system models) focuses on testing with system models. The literature search showed that the articles relevant to this topic are mostly published between 2000-2009. Both presented approaches use requirements traceability in their model-based test generation processes and provide a tool support to generate tests automatically. However, both approaches lack the detailed explanation on how the test cases are generated via the provided test generation tools. Since in the first approach (citation) provided test generation tool, Qtronic, is not existing anymore, the second approach (citation), which provides LEIRIOS Test Designer tool (currently named Smartesting), could be recommended for the automatic test generation with system models.

Chapter x (citation for topic 8: Testing FRs and NFRs in URN) presents approaches to automatically generate test cases for functional and non-functional requirements from User Requirements Notation. The literature search showed that this topic has not been researched enough, as this test generation process requires an intermediate step. The first presented approach (citation) addresses both functional and non-functional requirements, however, the provided Validation Framework is not available and without it, this approach cannot be used. Although the second presented approach (citation) generates test cases only for functional requirements, it is still recommendable as the required frameworks, jUCMNav and JUnit, are still available and the whole process can be done quickly without problems.

Chapter x (citation for topic 9: Testing NFRs with risk analysis) focuses on testing non-functional requirements with risk analysis. Both presented approaches deal with non-functional requirements and mention risk analysis only as a side note. Although the first presented approach (citation) describes how non-functional requirements can be defined and controlled, it does not specify a way to test them except for simulating the operating condition. The second presented approach (citation) leaves it to software architects to define non-functional requirements and uses only architectural non-functional requirements that exist as code conventions and other guidelines. The conclusion is that non-functional requirements with higher risks need to be paid more attention to by giving the tests higher priority.

Chapter x (citation for topic 10: Testing NFRs with aspects) presents approaches to harness aspect-oriented techniques for testing non-functional requirements. While the first presented approach (citation) points out the lacking tool support and the need for developers to have a firm understanding of aspect-oriented programming pose possible challenges, the second presented approach (citation) solves these problems (partially) by introducing ACRE, a tool that parses DSL statements to generate test aspects automatically. However, this tool only supports four aspect types and is limited to C++ applications. The literature search showed that systematic testing using aspects is not in focus of current research anymore due to its implementation difficulty. Nevertheless, existing approaches and tools should be further refined to make use of the advantages of aspect-oriented testing, especially for non-functional requirements.

The improved knowledge areas of the SWEBOK according to the synthesis matrices of the individual chapters are the following:

Knowledge areas from the field of *Software Testing* were improved by every topic. Improvements in the field of *Software Requirements* are part of every topic except topic 5. Topics 5, 8, 9 and 10 specifically improved a knowledge area from the field of *Software Maintenance*. The field of *Software Engineering Models and Methods* is improved by the topics 3, 4, 7 and 8. Only the approaches from topic 2 and one of the approaches of topic 5 are mentioned to help with *Software Construction*.

In summary, while the topics 7 and 10 are not popular in current research, the topics 2, 4 and 8 are generally not popular in research. However, still two different approaches could be found and are described for each topic of this work. Also most of the presented approaches are recommended under appropriate circumstances. These circumstances include having persons with the needed skills as in Topic 2 or having access to specific tools as in Topics 7 and 8. It should be mentioned that even though these approaches seem useful, often certain steps are not explained in detail. In particular, this is the case for the test generation steps such as in Topic 3, 7, and 9.

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.fitnesse.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://fitnesse.org/FitNesse.UserGuide.WritingAcceptanceTests.FitFramework.WritingFitTables

[17] 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)

[18] 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

[19] Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.-M.: Automatic Test Generation: A Use Case Driven Approach. In: IEEE Transactions on Software Engineering (Volume: 32, Issue: 3, March 2006). https://doi.org/10.1109/TSE.2006.22.

[20] Raza, N., Nadeem, A., Iqbal, M. Z. Z.: An Automated Approach to System Testing based on Scenarios and Operations Contracts. In: Seventh International Conference on Quality Software (QSIC 2007). https://doi.org/10.1109/QSIC.2007.4385504.

[21] Hausberger, F.: Research planning, research and mid-term presentation, https://github.com/fidsusj/SWE-Seminar, last accessed 2021/01/10.

[22] Sarma, M., Mall, R.: System Testing using UML Models. In: 16th IEEE Asian Test Symposium (ATS 2007). https://doi.org/10.1109/ATS.2007.102.

[23] Ibrahim, R., Saringat, M. Z., Ibrahim, N., Ismail, N.: An Automatic Tool for Generating Test Cases from the System's Requirements. In: 7th IEEE International Conference on Computer and Information Technology (CIT 2007). https://doi.org/10.1109/CIT.2007.116.

[24] Chen, L., Li, Q.: Automated test case generation from use case: A model based approach. In: 2010 3rd International Conference on Computer Science and Information Technology. https://doi.org/10.1109/ICCSIT.2010.5563772.

[25] Zhang, X., Tanno, H.: Requirements document based test scenario generation for web application scenario testing. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). https://doi.org/10.1109/ICSTW.2015.7107465.

[26] Li, L., Miao, H.: An Approach to Modeling and Testing Web Applications Based on Use Cases. In: 2008 International Symposium on Information Science and Engineering. https://doi.org/10.1109/ISISE.2008.265.

[27] Kim, Y. G., Hong, H. S., Cho, S. M., Bae, D. H., Cha, S. D.: Test cases generation from UML state diagrams. In: IEEE Proceedings - Software, Vol. 146, No. 4, pp. 187-192, Aug. 1999. https://doi.org/10.1049/ip-sen:19990602.

[28] Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.-M.: Requirements by Contracts allow Automated System Testing. In: Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE November 2003). https://dl.acm.org/doi/10.5555/951952.952350.

[29] AspectC++ Development Team: About the Project. https://www.aspectc.org/ last accessed 2020/12/28

[30] Dijkstra, E.: A Discipline of Programming. In: Prentice-Hall Series in Automatic Computation Englewood Cliffs, New Jersey. (1976)

[31] Duclos, E., Le Digabel, S., Guéhéneuc, Y., Adams, B.: "ACRE: An Automated Aspect Creator for Testing C++ Applications". In: CSMR 2013 Genova. pp. 121-130. (2013). 10.1109/CSMR.2013.22

[32] Eclipse Foundation: AspectJ. https://www.eclipse.org/aspectj/ last accessed 2020/12/28

[33] Metsä, J., Katara, M., Mikkonen, T.: "Testing Non-Functional Requirements with Aspects: An Industrial Case Study". In: QSIC 2007 Portland. pp. 5-14. (2007). 10.1109/QSIC.2007.4385475

[34] Parnas, D.: On the criteria to be used in decomposing systems into modules. In: Communications of the ACM. (1972)

List of Figures

List of Tables