Author 1, 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

[@bugwelle Introduction(1/2)]

1.1 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 "Guide to the Software Engineering Body of Knowledge" (short: 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 1.1. 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, fin 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 behaviour 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).

1.2 Outline

Following this introduction in Section 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 Testing with natural language processing

3 Acceptance testing with FitNesse

4 Testing with a transition system

5 Testing with a timing component

6 Testing with a classification tree

7 Testing with a formal specification

8 Testing with system models

9 Testing functional and nonfunctional requirements in User Requirements Notation

10 Testing Non-Functional Requirements with Risk Analysis

10.1 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 10.2. Each approach will be described and applied to an example in the form of movie management software in the respective sections 10.3 and 10.4.

Both approaches are compared in section 10.5 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 10.6.

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

10.2 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 10.3). This is why search-term based search was performed using the key terms: non-functional requirements, testing and risk analysis.

Table 10.1 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 10.1: 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.

10.3 Approach 1

10.3.1 Description of Approach 1

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

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

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

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

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

Control cases–as the name suggests–control the operating conditions and can be used to ensure that they are complied to. This makes it possible to mitigate business risks which may affect the business if the operating condition and its constraints are violated. Their paper visualizes the connection between these artifacts using an uniform modeling languages (UMLs) diagram, which can be seen in Figure 10.1 below.

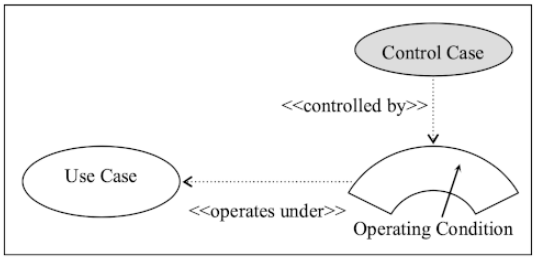

Figure 10.1: Association in Use Cases Modelling [4]

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 10.2.

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.

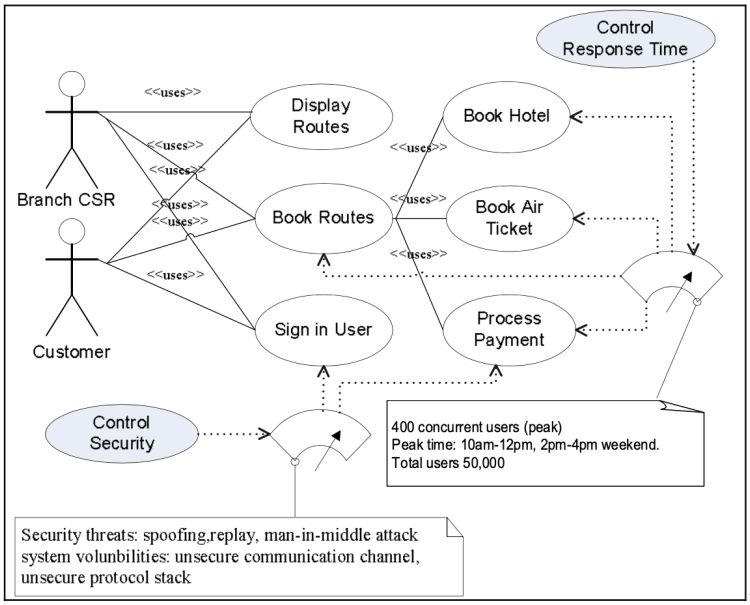

Figure 10.2: Use Case Model with Control Cases [4]

10.3.2 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 10.2 on page 21.

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 10.3 on page 22.

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 10.2: 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.

10.4 Approach 2

10.4.1 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 10.3: 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 UMLs diagram as can be seen in Figure 10.3. 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.

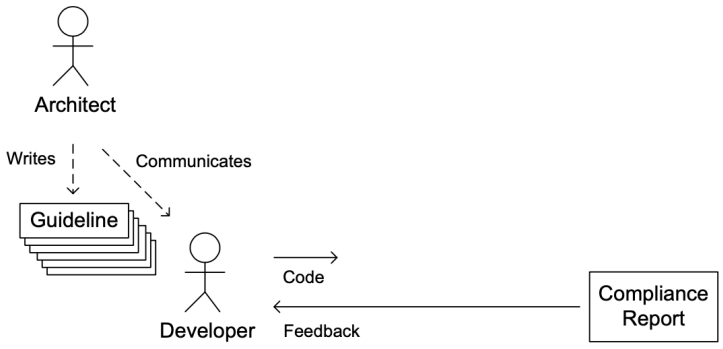

Figure 10.3: The common way of communicating architectural requirements [5]

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 10.4 to visualize this approach. Developers get feedback through different tools that the architect extends. Tools such as the editor, compiler or static analysis tools.

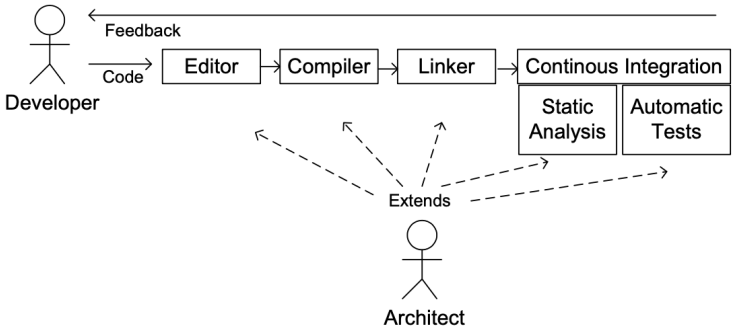

Figure 10.4: Suggested solution of communicating requirements [5]

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.

10.4.2 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 10.4. This system function and the following NFRs are based on personal experience in maintaining an open-source media manager.

In Table 10.5 on page 24, two NFRs are listed which were created for the system function in Table 10.4. 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 10.4: 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 10.5: 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 NFRs, all strings must be run through a certain function which escapes strings. Because this may be missed by the developer, a new string-subclass is introduced which escapes its input automatically, e.g. `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.

10.5 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.

10.6 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.

11 Testing nonfunctional requirements with aspects

12 Conclusion

Fusce vitae quam eu lacus pulvinar vulputate. Suspendisse potenti. Aliquam imperdiet ornare nibh. Cras molestie tortor non erat. Donec dapibus diam sed mauris laoreet volutpat. Sed at ante id nibh consectetuer convallis. Suspendisse diam tortor, lobortis eget, porttitor sed, molestie sed, nisl. Integer enim nisl, lacinia in, pretium eu, viverra a, odio. Quisque at quam eget risus placerat porttitor. Suspendisse convallis, elit vitae mattis pharetra, orci nisl ultrices sapien, ac interdum metus lorem iaculis diam. Nunc id nunc sit amet nisl tincidunt congue. Curabitur et sapien.

13 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.

List of Figures

List of Tables