Institut für Informatik
der Technischen Universität München

Reducing System Testing Effort by Focusing on Commonalities in Test Procedures

Benedikt Hauptmann

Institut für Informatik
der Technischen Universität München

Reducing System Testing Effort by Focusing on Commonalities in Test Procedures

Benedikt Hauptmann

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Jörg Ott

Prüfer der Dissertation:

1. Prof. Dr. Dr. h.c. Manfred Broy
2. Prof. Dr. Arie van Deursen
   Delft University of Technology / Niederlande

Die Dissertation wurde am 09.03.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.06.2016 angenommen.

Abstract

Testing is one of the most common ways to assess the correctness of software, but also one of the most effort-consuming activities in software development. System testing, as one type of testing, is an extraordinary effort driver. Expenses arise while creating, executing and maintaining test cases, which are the key artifacts in testing. The goal of testing is to execute a software system in many possible ways to improve our confidence that there are no errors in the system. This leads to one outstanding characteristic of test cases: The inherently high *commonality in the flow of test steps* (the test procedures). However, this commonality often results in redundant parts of test cases, so called test clones. To understand this phenomenon, we performed a study analyzing clones in industrial system test suites. We uncovered two consequences to testing activities: First, clones in tests directly lead to similar test execution. Since this strongly influences whether test automation pays off, clones should be considered when making a decision for or against test automation. Second, clones increase maintenance costs of test artifacts since changes in cloned parts have to be done multiple times.

This thesis addresses these consequences by proposing three ways to reduce effort for creating, executing, and maintaining system test cases: First, we support test engineers in choosing between execution modes at hand (even in the presence of clones). We propose an effort estimation model to balance advantages and disadvantages of test automation and manual testing. An industrial case study (41 test cases) demonstrates that our approach is applicable in practice and that the overall effort of system testing can differ strongly between execution modes. A coarse benefit estimation indicated that, in our case study, an adequate execution will reduce the overall system testing costs up to ∼20 – ∼30% (within two years).

Second, we present an constructive approach to make tests easier to execute and maintain. Test clones often overlap and can be complex to understand which makes it difficult to extract them. Using grammar inference techniques, we identify how to cut clones into suitable units for reuse. In two industrial case studies (97 test cases in total), we demonstrate the relevance of the problem and the effectiveness and applicability of our approach (in 16 out of 18 cases, test engineers considered our approach useful). In our cases, a coarse benefit estimation showed that our approach pays off after the test suites are maintained 2 – 3 times. If maintained more often (5 – 10 times), the overall system testing costs will shrink up to ∼10%.

Third, in addition to constructive quality improvements, we introduce *Natural Language Test Smells*, an analytic approach to identify quality problems in test cases regarding test executability and maintainability. We applied our approach to 9 industrial test suites (5,433 test cases in total) and found out that our approach has a precision of 73.8% and uncovered ∼80,000 correct findings in total. Test engineers considered 75% of the inspected findings as relevant enough to fix them either immediately or at next opportunities. In our cases, a coarse benefit estimation showed, that removing smell findings can reduce the overall system testing costs up to ∼10%.

Acknowledgments

One of the most important things I learned as an academic is that scientific work is a team sport. And similarly to other team sports, there is not only one player who is doing the match; many people have to stick together to make achievements. In the last years, I was in the fortunate situation of having many colleagues and friends who supported me, motivated me, and inspired me: I am very thankful to my supervisor Prof. Manfred Broy, who managed to create a wonderful and fruitful atmosphere for scientific work in his research group. I am grateful for his guidance and motivational support I got as his student. Furthermore, I want to thank my co-supervisor, Prof. Arie van Deursen. I appreciate his inspiring discussions and his hospitality during my visit in his group in Delft. Besides my two supervisors, I am deeply thankful for the support of Elmar Jürgens. Especially in the first years of my PhD, his guidance as a postdoc taught me to structure my ideas and to focus my work. I furthermore thank my close colleagues Maximilian Junker, Sebastian Eder, and Henning Femmer. Without their inspiration, support, and commitment, most of my ideas would not have been realized. I am very happy that I was (and still am) part of such a great team. Also, I want to thank Andreas Vogelsang, Daniel Méndez, Florian Deißenböck, Hanna Skiba, Jonas Eckhardt, Lars Heinemann, Marco Kuhrmann, Peter Braun, Stefan Wagner, Tanya Mirzayans, and Veronika Bauer for reviews of my work. Their feedback was often hard but always fair and helped me a lot. Special thanks goes to Silke Müller, the heart and soul of our research group. Her organization skills often helped me to survive the bureaucracy in our university. I want to thank my sister Lisa and her friend Diana Wirth for doing parts of the layout of my thesis. I guess it is quite obvious which page the two designed. Furthermore, since I was always focusing on applying and evaluating my research in practice, I am also very thankful for the cooperation with our industry partners. In particular, I want to thank Rainer Janßen and Rudolf Vaas for our long-time cooperation with Munich Re, which was the basis for most of my work. But most important, I am particularly thankful to my beloved girlfriend Christiane. By dealing with my temper during stressful times of my PhD, she probably had to pay the heaviest price of this work. Thank you so much, Chrissi.

Always be yourself! Unless you can be Batman, then always be Batman.

The Internet

Contents

1 Introduction **1**
1.1 System Test Cases . . . 3
1.2 Problem Statement . . . 8
1.3 Approach of this Thesis . . . 8
1.4 Contributions . . . 9
1.5 Case Study Partners . . . 12
1.6 Outline . . . 12

2 Fundamentals, Terms, and Definitions **15**
2.1 Functional System Testing . . . 15
2.2 System Test Execution and Corresponding Artifacts . . . 17
2.3 A Generic System Test Process . . . 20
2.4 Similarity of Test Cases . . . 26
2.5 Summary . . . 30

3 State of the Art **31**
3.1 Automating Testing Activities . . . 31
3.2 Balancing Automation in Testing . . . 35
3.3 Optimizing Goals of Test Suites . . . 39
3.4 Optimizing Representations of Test Suites . . . 41

4 Clones in Manual System Tests **47**
4.1 Research Goal . . . 47
4.2 Research Questions . . . 47
4.3 Study Design . . . 49
4.4 Results . . . 52
4.5 Interpretation and Discussion . . . 58
4.6 Threats to Validity . . . 60
4.7 Summary . . . 61
4.8 Conclusions to System Testing Life-Cycle Activities . . . 61

5 Choosing Execution Modes **63**
5.1 Test Meta-Model . . . 64
5.2 Relative Cost-Benefit Analysis . . . 65
5.3 Cost Estimation Model . . . 67
5.4 Evaluation . . . 68
5.5 Benefit Estimation . . . 75

5.6 Summary 80

6 Test Refactoring Using Grammar Inference 81

6.1 A Conceptual Test Meta-Model 84
6.2 Significance of Overlapping Clones 85
6.3 Approach 89
6.4 Evaluation 95
6.5 Benefit Estimation 100
6.6 Future Work 106
6.7 Summary 106

7 Natural Language Test Smells 109

7.1 Definitions and Terminology 111
7.2 Approach: Natural Language Test Smells 116
7.3 Automated Detection of Natural Language Test Smells 121
7.4 Evaluation 125
7.5 Benefit Estimation 136
7.6 Summary 145

8 Summary 147

8.1 Summary of the Problem and the Contributions 147
8.2 Outlook and Future Work 149

Bibliography 155

Chapter 1

Introduction

Testing is a central activity of quality assurance in software development. The goal is to find errors as well as to gain confidence that a system works as intended. However, this confidence has to be paid with a high price: Testing consumes huge parts of the overall software development effort. In *The Art of Software Testing*, Myers and Sandler [2004] state that testing is a major effort factor not only in recent software projects, but ever since:

> *"In 1979, it was a well-known rule of thumb that in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed. Today, a quarter of the century later, the same is still true."*

Those numbers are supported by earlier work, too. Beizer [1990] estimates the effort for testing at 50% labor expenses minimum. More recent work confirms the role of testing as major effort driver: Ramler and Wolfmaier [2006] support the estimation that testing accounts for at least 50% of project costs. In the year 2000, Harrold [2000] states similar estimations and furthermore adds that, in some application domains such as avionics systems, testing expenses are even higher. Blackburn et al. [2004] go one step further by not only supporting those well-known effort estimations, but also by adding that those estimations are independent from the experience and development competence of software development companies.

Software Testing

Testing comprises different parts, each takes place at different stages of the development process and has different goals: *Unit testing*, for example, verifies that units of source code are correctly implemented, whereas *integration testing* checks the interaction of several (not necessarily all) software units and therefore verifies the implementation of software designs. In contrast, *system testing*, tests completely integrated systems as a whole verifying whether a completely integrated software system corresponds to its functional requirements.

Focus of this Work:

To make the contributions of this thesis more specific, we make the following restrictions:

System Testing/Functional Testing:
We focus on *system testing*: Testing conducted on a complete, integrated system to evaluate if it complies with its specification. Hence, we use the terms *test* and *testing* synonymously for *system test* and *system testing*. We focus on tests verifying mostly functional aspects of systems (called *functional testing*). In Section 2.1 *Functional System Testing*, we introduce terms and definitions of software testing in more detail.

Application Domain:
We focus on testing of interactive systems, such as business information systems. More specifically, we focus on those types of software systems that can be tested in both ways, manually as well as using automatic testing techniques. This covers software systems that are primarily operated by human users having dedicated human-computer interfaces (e.g., graphical user interfaces) and therefore are also tested using these interfaces. We explicitly exclude software systems that have primarily technical user interfaces (such as programming interfaces or web services) since they can only be tested in automated ways. For the same reason, we also exclude systems having strict timing constraints, such as real-time systems.

System Testing Effort

In industry, system testing is considered as an extraordinary effort driver. However, little scientific evidence of the amount of expenses for different types of testing exist. Most published reports are rather imprecise and report data without going into detail for which test activities time and money is actually spent.

A survey of the U.S Department of Commerce, for example, shows that the effort spent on integration and system testing grew from the 1960s and 70s to the 1980s from 10% up to 20% of the overall development costs [Tassey, 2002]. However, that survey has been published in the year 2002 and has not been updated with new data since. Nor does it explain on what specific system testing activities effort is spend.

Based on industrial experiences, Ramler and Wolfmaier [2006] give an explanation for high system testing effort in industry. They state that practitioners often miscalculate the abilities and efforts of automated system testing:

> *"Practitioners frequently report disastrous failures in the attempt to reduce costs by automating software tests, particularly at the level of system testing."*

While performing the case studies of this thesis, we were able to gather some own anecdotal evidence about testing effort in industry. We got a coarse-grained split-up of development effort (in terms of project budget) of a business information system of our industry partner Munich Re (see Section 1.5 *Case Study Partners*). The software system is 12 years old and is still actively used and maintained. Testing is performed by engineers of Munich Re and of external subcontractors. The data we got is reconstructed using internal time tracking and billing information of the project teams.

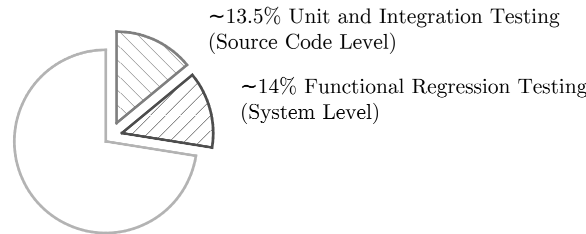


Figure 1.1: Testing share of the overall budget of a software project of Munich Re.

In the mentioned project, ~27.5% of the annual project budget is spent on testing the developed software. Those testing expenses split up into two different types of testing: Approximately half of the testing effort (~13.5% of the overall project budget) is spent on unit and integration testing on the source code level. The other half (~14% of the overall project budget) is spent on functional regression testing on the system level (see Figure 1.1). However, this data represents just one project of Munich Re. Based on interviews with engineers of Munich Re, the testing expenses of the mentioned project is on the lower border of what typical projects of Munich Re spend on testing. In other projects of Munich Re, a larger share of the project's budget is spent on testing.

1.1 System Test Cases

To understand how system testing effort arises as well as how to find ways to reduce it, we focus on the central artifact of system testing, namely *(system) test cases*. In the following section, we discuss the activities that are applied to system test cases (*system test case life-cycle*). Additionally, we describe one particular characteristic of test cases: Commonality in test procedures. Finally, we give a preview of the results of our study (Chapter 4 *Clones in Manual System Tests*) and illustrate the consequences of commonalities, which increase the effort for the system test case life-cycle activities.

1.1.1 The System Test Case Life-Cycle

To understand the influence of test cases to testing expenses, we describe how they are used in system testing. Concretely, we describe the activities that are applied on them while testing software systems. Test cases, the key artifacts in system testing, undergo a certain life-cycle of the activities: They are created, executed and maintained (see Figure 1.2).

Figure 1.2: The system test case life-cycle.

Test Creation: First, test cases are created. In literature, there is often no clear distinction between *test derivation* (identifying test ideas) and *test implementation* (writing test artifacts). In the context of this work, we focus on the latter, the transition of abstract to executable concrete test cases such as automated test scripts or test descriptions used for manual execution.

Test Execution: Second, tests are executed. This covers the execution of tests either manually by human testers or automatically by running test scripts. During the actual test execution, the strengths and weaknesses of the chosen execution techniques pay off.

Test Maintenance: Third, over time, tests are adapted to changed project needs such as changed testing goals and new or modified functionality of the system under test.

1.1.2 Commonalities in Test Procedures

The goal of testing is to execute a software system extensively in order to find errors. This leads to one particular characteristic of test cases: The inherently high commonality in the flow of test steps (their *test procedures*). In the following, the cause for this characteristic is explained in more detail by giving some reasons explaining how test commonality originates:

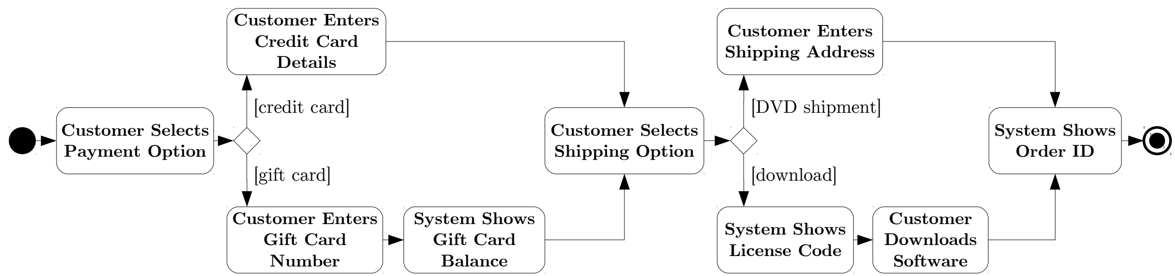
Test Coverage: The creation of test cases is often aligned to test coverage criteria, such as the execution of all possible (exceptional) cases of the specified functionality. The test derivation technique *equivalence class partitioning* [Spillner et al., 2007], for example, aims at finding relevant classes of test data to create test cases. *Boundary-value-analysis* [Spillner et al., 2007], as another example, builds up on these equivalence classes and proposes additional test cases that test the system's correct behavior at the boundaries of the equivalence classes. All these techniques lead to large numbers of test cases, which are very similar but differ in minor aspects such as their input data, the expected response of the system, or minor parts of test procedures.

Commonality of Functionality: Another reason for commonality in test procedures might be the functionality that is tested. Often, functionalities contain parts that occur in several functions throughout a system, such as handling errors or loading default data records. This leads to test procedures which will inevitably contain similar parts while actually testing different functionalities.

Function Dependencies: Similarly to identical parts in functionality, dependencies between functionalities (such as preconditions or inheritance of functions) also lead to commonality in test procedures. For example, if a system requires users to authenticate first, all test cases have to run through the login functionality first to reach their desired functionality.

Example

We briefly illustrate this phenomenon of commonality in test procedure using the example of an online software shop. Figure 1.3a shows the functionality *cart checkout* outlined in form of an activity diagram. Customers can choose between two ways of payment. By credit card or using a gift card. Furthermore, they can decide whether they want to get the product shipped by mail or download it immediately.



(a) The functionality *cart checkout* outlined as activity diagram.

Credit Card + DVD Shipment
Precondition: Cart contains a product.

Credit Card + Download
Precondition: Cart contains a product.

Gift Card + DVD Shipment
Precondition: Cart contains a product.

Gift Card + Download
Precondition: Cart contains a product.

```
1  *** Test Cases ***
2  Card Checkout with Credit Card and DVD Shipment
3      Open Web Shop
4      Search For Product        12345
5      Add Product to Cart
6      Checkout cart
7      Select Payment Option     creditcard
8      Enter Credit Card Details VISA, 4263...
9      Select Shipment Option    dvd shipment
10     Enter Shipment Address    TUM, Munich, ...
11     Verify Order ID
12
13 Card Checkout with Credit Card and Download
14     Open Web Shop
15     Search For Product        12345
16     Add Product to Cart
17     Checkout cart
18     Select Payment Option     creditcard
19     Enter Credit Card Details VISA, 4263...
20     Select Shipment Option    online download
21     Verify License Code
22     Verify Download Link
23     Verify Order ID
24
25 Card Checkout with Gift Card and DVD Shipment
26     Open Web Shop
27     Search For Product        12345
28     Add Product to Cart
29     Checkout cart
30     Select Payment Option     giftcard
31     Enter Gift Card Number    123...
32     Verify Remaining Balance
33     Select Shipment Option    dvd shipment
34     Enter Shipment Address    TUM, Munich, ...
35     Verify Order ID
36
37 Card Checkout with Gift Card and Download
38     Open Web Shop
39     Search For Product        12345
40     Add Product to Cart
41     Checkout cart
42     Select Payment Option     giftcard
43     Enter Gift Card Number    123...
44     Verify Remaining Balance
45     Select Shipment Option    online download
46     Verify License Code
47     Verify Download Link
48     Verify Order ID
```

(b) Four test cases covering different paths through the *cart checkout* functionality. The left side shows the basic idea of each test case on a conceptual level. On the right side, each test case is manifested in form of an automated test.

Figure 1.3: An example of an online software shop demonstrating commonality in tests.

Figure 1.3b shows four test cases, each testing a different combination of payment and shipping option. The left part of the figure outlines the test cases on the conceptual level, by showing which parts of the functionality each tests runs through. On the right part of the figure, each conceptual test case is materialized as an automated test case[1].

This example shows that test cases contain commonality on both, the conceptual as well as on the artifact level. On the conceptual level, all four test cases are very alike since they are testing just different variants of the same functionality. They furthermore have the same precondition since they all require that a product has been added to the cart before. We call commonality on this level *conceptual commonality*. However, in our example, this commonality has been carried over to the artifact level: The automated tests contain parts that are the same. We call commonality on this level *artifact redundancy* or *cloning* (we discuss both in Section 2.4).

1.1.3 Consequences of Commonality to the System Test Case Life-Cycle

Based on evidence and observations of practitioners and researchers [Marick, 1999; Persson and Yilmazturk, 2004; Ramler and Wolfmaier, 2006] as well as on our own experiences and studies (see Chapter 4 *Clones in Manual System Tests*), we see that commonality in test procedures leads to the following consequences, negatively influencing the system test case life-cycle: First, commonality complicates the decision when to automate tests and when to rely on manual execution. Second, commonality causes cloning in test artifacts which hampers executing and maintaining them. In the following, both consequences are explained in more detail (see Figure 1.4).

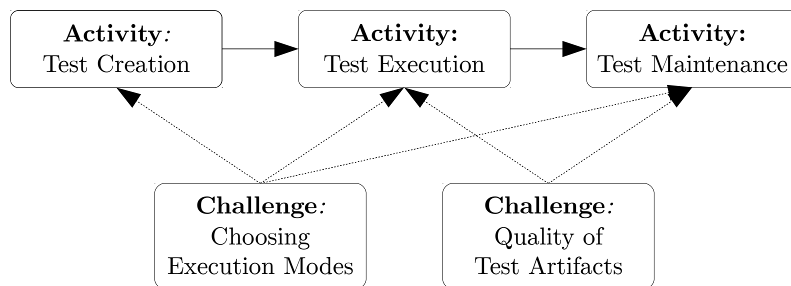


Figure 1.4: Consequences of commonality to the system test case life-cycle.

Choosing Execution Modes

To execute system tests, two basic execution techniques exist, manual and automated execution. Both techniques have advantages and disadvantages: In manual test execution, the initial costs are low, but the variable costs can grow tremendously the more often tests are executed. Automated test execution relies on test scripts which are expensive to create and maintain, but promise cheaper and faster test execution. For each test suite, one has to decide when to use either execution technique. We call this decision determining the *execution mode*. However, defining execution modes is not a trivial task and involves risks: If they are chosen inappropriately, testing budgets can be wasted easily. Kaner et al. [2002] state:

[1] The tests are denoted in a keyword-driven style [Fewster and Graham, 1999] (cf. Section 2.2.2).

> *"[. . .], behind the decision to automate some tests is a cost and benefit analysis. If you get the analysis wrong, you'll allocate your resources inappropriately."*

The adequacy of execution modes depends on a variety of information [Dustin et al., 1999; Fewster and Graham, 1999; Linz and Daigl, 1998; Ramler and Wolfmaier, 2006; Schwaber and Gilpin, 2005] such as test script and test description development costs, or the expected changes to the system under test, but also the execution frequency of each part of a test suite. The latter does not only depend on the number of planned test runs, but also on how often parts of test scripts are reused among test suites (commonality of the test procedures) [Greiler et al., 2012; Hauptmann et al., 2012b]. For example, the study presented in Chapter 5 *Choosing Execution Modes* revealed that by executing a test suite of 42 test cases once, some test steps were executed more than 150 times.

In practice, there are often no structured approaches to cope with this variety of information. Execution modes are determined based on best practices, experiences from similar projects, rules of thumb and gut feeling. Although experts make decisions mostly leading to tolerable results, there are still severe drawbacks: Determining execution modes in an ad-hoc style may result in solutions that might not be cost-efficient. Furthermore, a once determined selection of manual and automated tests can hardly be changed when a system or the development context evolves as it is unclear on what basis the decision was made.

To keep testing efforts low, we need methods and tools to support test engineers in choosing appropriate execution modes. This evaluation has to cover all relevant types of information including the commonality throughout test procedures.

Quality of Test Artifacts

In addition to test execution, the creation and maintenance of test artifacts are further effort drivers. Ramler and Wolfmaier [2006] state that those drivers are often misunderstood beforehand:

> *"In most cases, reasons for failed [software] projects include a gross underestimation of the effort required to develop and maintain automated tests."*

To reduce testing effort, test artifacts have to be of high quality. From other areas of software development, we know that the quality of artifacts strongly influence how expensive it is to maintain them. The same is true for system tests: If test artifacts are inappropriate to support execution and maintenance of test cases, testing effort increases.

Redundancy in software artifacts (also known as *cloning*) is considered as a major quality factor since it can lead to failures and increased maintenance effort [Juergens et al., 2009]. Due to the nature of testing, test cases often contain redundancy that is caused by conceptual commonality which has been carried to the artifact level (see Section 1.1.2).

To support the executability and maintainability of tests, we need ways to detect and eliminate quality defects such as clones in test procedures. Furthermore, we need ways to perform quality analysis continuously to prevent degradation and to maintain quality over time.

1.2 Problem Statement

Summarizing the consequences of commonality, we formulate the following problem statement which we address in this thesis:

Functional system testing is an effort consuming task in software development. We see two reasons for this high effort: (1) There are no structured methods to choose between manual and automated test execution, which leads to execution modes that are not cost-effective. (2) Poor artifact quality leads to test cases that are inappropriate to be executed and maintained, which increases testing effort.

1.3 Approach of this Thesis

This section first introduces two basic principles this thesis is based on. Afterwards, we describe the research methodology of this work.

1.3.1 Key Principles

While addressing the problem statement, we adhere to the following two key principles:

Principle 1: Non-Invasiveness

Based on our experiences, established processes and tools of organizations are hard to change. Software development and testing processes of companies have often been developed based on experiences of previous projects and are optimized to fit in existing project settings and development strategies. Furthermore, tools are often centrally selected and predefined for whole companies to fit to existing processes and surrounding tool chains. This makes it difficult to change company-wide standards, despite the existing evidence of other, more promising solutions.

To circumvent these obstacles, we design our contributions to be as little invasive as possible. To make them easier adoptable in practice, we try not to interfere with established processes and existing tools whenever possible. Instead, we complement existing processes and tools, compensate their disadvantages and support test engineers in performing established testing activities more efficiently.

Principle 2: Test Cases as Starting Points

Test cases are the key artifacts for all system testing activities and are therefore directly related to testing effort: They have to be created (mostly) manually by test engineers. They are key drivers for test execution and therefore determine the effort for test runs. For long living systems, the number of test cases and the way they are manifested determines the effort maintenance and adaptation to changed functionality makes.

We address the problem of high system testing effort by taking test cases and the activities that are related to them as starting points for our contributions. More specifically, we focus on the system test case life-cycle, namely the activities of test case creation, execution and maintenance, and aim at supporting test engineers in performing those activities.

1.3.2 Research Methodology

In this thesis, we apply the *industry-as-laboratory* approach as suggested by Potts [1993]: Following this approach, *"researchers identify problems through close involvement with industrial projects, and create and evaluate solutions in an almost indivisible research activity"*.

Phase 1: Problem Analysis

Following this philosophy, we first gain better understanding of the problem we want to address. Together with one of our industry partners, we investigate the phenomenon of commonality in real-world test suites and its impact to system testing effort (see Chapter 4 *Clones in Manual System Tests*). Performing this initial study closely with companies does not only lead to a deeper understanding of the problem domain but also helps to ensure, that the identified challenges are relevant and worth to be addressed (parts of the results have already been mentioned in Section 1.1.3 *Consequences of Commonality to the System Test Case Life-Cycle*).

Phase 2: Addressing the Problem

In the second phase, we pick up the challenges that have been identified in the first phase and propose three ways to reduce system testing effort (Chapter 5 *Choosing Execution Modes*, Chapter 6 *Test Refactoring Using Grammar Inference*, and Chapter 7 *Natural Language Test Smells*). Each of the proposed contributions follows our key principles by being as less invasive as possible to existing processes and tools (principle 1) and by tackling the problem at the artifact level (principle 2). Having the daily work of test engineers in mind, we develop ways of reducing testing effort by optimizing and supporting existing testing tasks. Each contribution is developed closely together with our industry partners. Thereby, we make sure that our contributions are realistic to implement and apply in practice. We use the close contact to industry to evaluate our approaches in real-life settings. According to Mcgrath [1995], research evidence should always try to maximize the following three criteria: Generalizability, precision and realism. We address those criteria by mixing different research methods: We combined techniques, such as expert interviews, controlled experiments, and experimental simulation and applied them together with two companies.

1.4 Contributions

This section summarizes the contributions of this thesis (see Figure 1.5).

Relevance and Understanding: Clones in Manual System Tests

To get a better understanding of commonality in system tests and their influence on the system test case life-cycle, we present an empirical study analyzing redundancy in manual system test case documents (so called *clones in test artifacts*). We performed clone detection on seven test suites from our industry partners (72 – 1800 manual test cases each test suite). All test suites contained 43% to 86% recurring text passages, some of which appeared more than 30 times within a test suite. The study lets us draw the following conclusions:

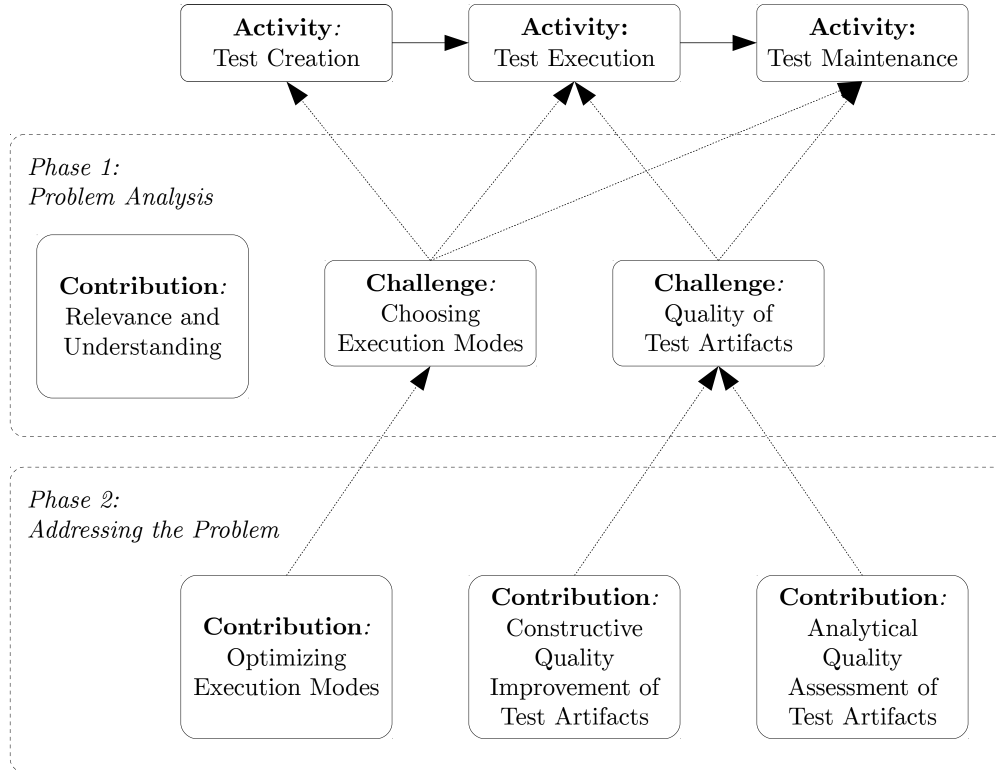


Figure 1.5: Contributions of this thesis.

Consequences to Test Execution: The more often a certain part of a test is cloned, the more often it will be executed and the earlier test automation will pay off. In our case study, we found large portions of cloned parts indicating that test automation will pay off here. But we also found many uncloned parts for which automation will not pay off. This demonstrates the challenge in choosing between manual and automated test execution.

Consequences to Test Maintenance: Cloning can negatively affect the maintainability of software artifacts [Juergens et al., 2009]. Since all analyzed test suites were affected by cloning, it demonstrates the need for methods and tools for quality control.

Optimizing Execution Modes

We propose an effort estimation model that can be used by test engineers to support choosing execution modes. Our model estimates testing effort in form of abstract costs and is based on static analysis of the structure of a test suite as well as on expert estimations. Goal of the model is to compute the estimated total costs for creating, executing and maintaining a given execution mode. Our model allows test engineers to balance advantages and disadvantages of execution modes at hand and thereby enables a more solid decision making process. With this contribution, we address the challenge of choosing execution modes (see Figure 1.5).

To demonstrate the consequences (w.r.t. costs of execution modes), we applied our estimation model to an industrial test suite (containing 41 system tests) and compared the

overall estimated costs of several common execution modes, such as performing all test cases manual, or performing all test cases automatically, or mixing both execution techniques. The results of our case study show that the overall estimated efforts differ strongly based on the chosen execution mode. In our case, costs differed by the factor of four. Although having the smallest overall costs in some situations, the combination of manual and automated test execution, if applied in the wrong way, can lead to costs that are far above all other execution modes. Only a small window exists where this strategy is considerably cheaper than fully manual or fully automated test suites. This leads to the conclusion that finding an appropriate execution mode is a challenging task that, if done wrong, can have time and money wasting consequences. This underlines the need for a structured way of choosing them.

To understand the benefit of our approach, we perform a coarse benefit estimation. For our industrial test suite, choosing an adequate execution mode would reduce the overall system testing costs up to ∼20% – ∼30% within two years compared to the originally planned pure manual execution.

Constructive Quality Improvement of Test Artifacts

To support test maintenance, we present an approach to constructively improve the quality of test cases. The presented approach builds upon the clone findings from our first study. However, instead of just identifying commonality in test cases, we now support test engineers in improving the maintainability of tests by removing clones. Using grammar inference techniques, we identify parts of tests that are suitable for extraction to reuse components. Our approach considers overlapping clones and suggests how to cut them into parts which are suitable for extraction and thereby leverages reuse mechanisms best. With this contribution, we address the challenge of poor quality of test artifacts (see Figure 1.5).

We surround our approach by two industrial studies: Each of the studies is applied to two test suites (97 test cases in total). A pre-study shows that test clones often overlap and can be complex to understand. Up to 31.7% of all test steps that were affected by cloning were overlapped by up to five clones at the same time. This makes it challenging to find good strategies to remove clones by refactorings. A evaluation study reveals that our approach helps test engineers in refactoring clones to avoid duplicated maintenance in cloned parts. In 16 out of 18 cases, test engineers would implement our refactorings (as is or slightly modified). Furthermore, our refactorings uncovered further conceptual problems of the test suite leading to more far-reaching quality improvements of the test suite.

To find out if and when it pays off to remove test clones, we perform a coarse benefit estimation of our approach: For our two study objects, implementing our refactorings has a positive benefit if the test suite is subject to maintenance for at least 2 – 3 times. If maintained more often (5 – 10 times), the overall system testing costs will shrink up to ∼10%.

Analytical Quality Assessment of Test Artifacts

In addition to constructive quality improvement methods, we also introduce an analytical quality assurance method for test artifacts. We identify quality defects in tests w.r.t. their executability and maintainability, such as redundancy or ambiguity in test procedures. Being able to uncover quality defects enables improving a test suite's executability and maintainability and, if applied continuously, helps to maintain both in the long run. We introduce *Natural Language Test Smells*, a technique to identify quality defects concerning executabil-

ity and maintainability of manual tests in natural language. Furthermore, we present tool support to automatically detect test smells. With this contribution, we address the challenge of poor quality of test artifacts (see Figure 1.5).

We evaluate our approach in an industrial case study using 9 test suites (5,433 test cases in total). To demonstrate the quality of our automated smell detection techniques and the relevance of smell findings, we manually assess smell findings and perform interviews with test engineers. The study reveals that our smell detection techniques have an average precision of 73.8% and were able to uncover ∼80,000 correct quality findings in total. The interviews reveal that test engineers consider 75% of the findings relevant enough to fix them either immediately or at next opportunities.

We perform a coarse benefit estimation to find out if and when the costs for removing smell findings are paid off. For our 9 study objects, removing smell findings will provide a positive benefit, if the test suite is executed at least ∼40 – ∼100 times and is subject to maintenance for more than 2 – 4 times. Removing smell findings can reduce the overall system testing costs up to ∼10%.

1.5 Case Study Partners

This section gives an overview of the companies that participated in our studies:

Munich Re: Munich Re is one of the world's leading reinsurance companies and employs about 43,000 employees throughout the world. For their insurance business, they operate a variety of individual supporting software systems.

Airbus Defence & Space: Airbus Defence & Space (formally Cassidian, an EADS company) is a worldwide leader in global security solutions and systems, providing products and services to civil and military customers around the globe. The systems we analyzed here stem from ground support systems used, for example, for mission planning.

Both companies are not software producers in the first place, they are neither software vendors nor is software development part of their main business. However, both develop a variety of business information systems to support their own core business processes. In many cases, they employ sub-contractors to develop the software. System tests are used to verify the correct implementation of the software. Furthermore, system testing is used for regression testing after feature improvements and bug fixing. Both companies have experience in system testing for many years.

1.6 Outline

The remainder of this thesis is structured as follows:

Chapter 2 ***Fundamentals, Terms, and Definitions*** introduces the terms and concepts that are used in the following chapters.

Chapter 3 ***State of the Art*** presents related work from the area of test execution, effort estimation modeling and test quality. It summarizes the state of the art and the state of the practice.

Chapter 4 ***Clones in Manual System Tests*** presents a study on clones in manual system tests. This study motivates that finding an appropriate execution mode is a common problem in software testing and is not limited to certain test suites. Furthermore, it demonstrates the need for methods and tools to improve the quality of test artifacts.

Chapter 5 ***Choosing Execution Modes*** introduces a cost estimation model to estimate the overall effort for test execution modes. We furthermore present an industrial case study on consequences of test execution modes. Our study motivates that inappropriate test execution modes can have tremendous effort wasting consequences.

Chapter 6 ***Test Refactoring Using Grammar Inference*** introduces an approach to constructively improve the quality of test cases by removing clones from test artifacts. We furthermore present a case study applying our constructive quality improvement approach to industrial test cases demonstrating its effectiveness and applicability.

Chapter 7 ***Natural Language Test Smells*** introduces a technique to identify quality defects concerning executability and maintainability of manual system tests. Furthermore, we present a case study demonstrating its effectiveness and benefit in industry.

Chapter 8 ***Summary*** resumes this thesis by presenting its contributions, their limitations and directions for future topics.

Previously Published Material

Parts of the contributions presented in this thesis have been published in:

[Hauptmann et al., 2012a] Benedikt Hauptmann, Veronika Bauer, and Maximilian Junker. *Using edge bundle views for clone visualization.* Short paper in *Proceedings of IWSC'12*, 2012, pages 86 – 87. IEEE.

[Hauptmann et al., 2012b] Benedikt Hauptmann, Maximilian Junker, Sebastian Eder, Elmar Juergens, and Rudolf Vaas. *Can clone detection support test comprehension?* Full paper in *Proceedings of ICPC'12*, 2012, pages 209 – 218. IEEE.

[Hauptmann et al., 2013] Benedikt Hauptmann, Maximilian Junker, Sebastian Eder, Lars Heinemann, Rudolf Vaas, and Peter Braun. *Hunting for smells in natural language tests.* NIER track in *Proceedings of ICSE'13*, 2013, pages 1217 – 1220. IEEE.

[Hauptmann et al., 2014] Benedikt Hauptmann, Maximilian Junker, Sebastian Eder, Christian Amann, and Rudolf Vaas. *An expert-based cost estimation model for system test execution.* Short paper in *Proceedings of ICSSP'14*, 2014, pages 159 – 163. ACM.

[Hauptmann et al., 2015] Benedikt Hauptmann, Sebastian Eder, Maximilian Junker, Elmar Juergens, and Volkmar Woinke. *Generating Refactoring Proposals to Remove Clones from Automated System Tests.* Full paper in *Proceedings of ICPC'15*, 2015, pages 115 – 124. IEEE.

Chapter 2

Fundamentals, Terms, and Definitions

This chapter introduces the fundamentals of this thesis. The first part introduces terms and definitions of system testing and integrates this work into common testing terminology. In the second part of this chapter, different ways of system test execution are introduced. We furthermore introduce the system testing process this work is based on. The later part introduces basic concepts and notions of similarity in test cases.

2.1 Functional System Testing

We integrate our perspective on the field of software testing into the terminology of the *IEEE Standard 829 for Software and System Test Documentation* [IEEE, 2008], the *International Software Testing Qualifications Board (ISTQB)*[1] and its *Standard Glossary of Terms used in Software Testing* [International software testing qualifications board, 2011], and the *ISO Standard 29119 Software and Systems Engineering — Software Testing* [ISO/IEC/IEEE, 2013a,b].

Our definition of *system testing* is based on the IEEE Standard 829 defining it as *"testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements"* [IEEE, 2008]. We accord with this definition, however, we focus on system testing verifying mostly functional aspects of systems (called *functional system testing*). Furthermore, since the definition of IEEE is still vague, we concretize it by clarifying the *test level*, the *test type*, and the *tests' formality*:

2.1.1 Test Level – Software Development Activities

Testing is commonly divided into *test levels* (also called *test phases* [ISO/IEC/IEEE, 2013a]), which are structured along basic software development activities [International software testing qualifications board, 2011; ISO/IEC/IEEE, 2013a; Pol et al., 2001]. The V-model [Boehm, 1979; Sommerville, 2010] illustrates the relationships of software development activities and their corresponding testing activities (see Figure 2.1).

[1] `http://www.istqb.org`

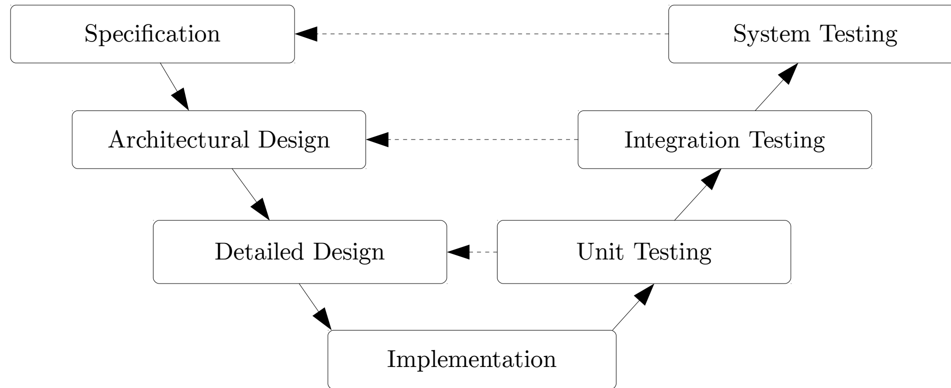


Figure 2.1: Development and testing activities illustrated in the V-model [Boehm, 1979; Sommerville, 2010] (simplified).

In this work, we neither focus on software development activities nor on responsibilities of certain roles within these activities. Instead, we focus just on how far the system under test is assembled: Our definition of system testing addresses all tests that are executed on a complete, integrated system independently from the development life-cycle activity or for what purpose they are conducted.

Our definition of system testing may also include *acceptance testing* which *"determine[s] whether or not a system satisfies the acceptance criteria and [...] enable[s] the user [...] to determine whether or not to accept the system"* [IEEE, 1990] as long as it is conducted on a complete, integrated system.

Furthermore, decoupling the definition of system testing from test levels and hence from development activities enables its application in development settings in which strict separations of development activities do not exist, such as agile software development (see [International software testing qualifications board, 2014a]).

2.1.2 Test Type – The Objective of Tests

Testing is divided into *test types* based on test objectives (or *quality characteristics* [ISO/IEC/IEEE, 2013a]), such as functional test, usability test, portability test, etc. [International software testing qualifications board, 2011; ISO/IEC/IEEE, 2013a; Pol et al., 2001]. System testing is commonly considered as *functional testing*, which aims at testing the implementation of *"[...] functions and features (described in documents or understood by the testers) [...]"* [International software testing qualifications board, 2011]. According to this definition, we focus but explicitly do not limit ourselves to functional testing.

Furthermore, we expect functional testing to be applied for both, to verify the correctness of newly implemented functionality and for *regression testing* aiming at testing *"[...] a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result to changes made"* [International software testing qualifications board, 2014b].

2.1.3 Formality – Formal Testing vs. Informal Testing

Testing is furthermore divided into formal and informal testing: In formal testing (also *scripted testing* [ISO/IEC/IEEE, 2013a]) all test cases are *"conducted in accordance with test plans and procedures that have been reviewed and approved by customer, user, or designated level of management"* [IEEE, 1990]. This requires test cases to be specified in form of *test case specifications*, which specify test objectives, inputs, test actions, expected results, and execution preconditions [International software testing qualifications board, 2014b].

On the contrary, informal testing (also called *ad hoc testing* or *exploratory testing*) [International software testing qualifications board, 2014b; ISO/IEC/IEEE, 2013a] is *"simultaneous learning, test design, and test execution"* [Bourque and Fairley, 2014]. This means, no detailed pre-specified test cases exist [Itkonen and Rautiainen, 2005] but manual testers spontaneously designs and execute tests.

Following our key principles (see Section 1.3.1 *Key Principles*), we take test cases and the activities that are related to them as starting points for our work. Therefore, in the remainder of this thesis, we focus on formal testing only.

2.2 System Test Execution and Corresponding Artifacts

The contributions presented in this work are based on the concept of separating fundamental techniques to execute test cases (*test execution techniques*) and how those techniques are applied to execute a test suite (*test execution modes*). In the following, we introduce both concepts and the required types of test case representations (artifacts).

2.2.1 Test Execution Techniques

To execute system tests, two fundamentally different *test execution techniques* exist: manual and automated test execution.

Manual Execution

In *manual execution*, tests are performed by humans (the *tester*). This means that all inputs, the analysis of the output as well as the evaluation are performed manually without any significant tool support. Basis for manual test execution are *test descriptions* (see next section). Test descriptions act as manuals that are guiding the tester and describing how to stimulate the system and how to verify whether the system responds correctly. The specific information and the level of details that is contained in the test descriptions depends on the domain knowledge of the manual tester.

Automated Execution

In contrast to manual execution, *automated execution* refers to performing test cases automatically, without manual interaction. This requires tests to be implemented as autonomously executable *test scripts* (see next section). In order to be automatically executable, test scripts have to be very detailed, containing precise information on how to stimulate the system (e.g., by having detailed information of the user interface).

2.2.2 Test Case Representations

A test case is the primary artifact that drives test execution. It describes how to stimulate a software system to reach a test objective (e.g., to execute a certain functionality), and how to verify that the system under test reacts appropriately. The goal of a documented test case is to make test execution reproducible, deterministic, and plannable (in contrast to ad hoc testing where manual testers spontaneously design tests along the way while performing it – see Section 2.1.3 *Formality – Formal Testing vs. Informal Testing*). We describe test cases by the following model (see Figure 2.2):

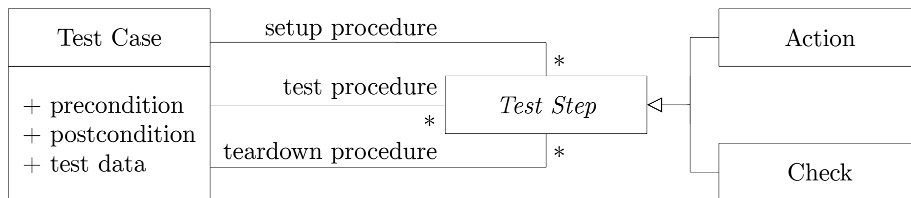


Figure 2.2: Parts of test cases.

Test Steps: Test steps are commands that describe how to interact with the system under test. They are commonly divided into *actions* that are describing how to stimulate the system under test and *checks* that describe how to verify whether it responds correctly.

Test Procedures: The main part of a test case is its test procedure, a sequence of test steps (actions and checks).

Test Data: If necessary, test cases are equipped with test data, which is used as input values for the software system (in actions) or as reference values to verify the response of the system under test (in checks). Test data may be embedded in test steps (in manual test cases, for example, in the text of a test step) or kept as attachments of test cases, which are referenced in test steps.

Pre-/Postconditions: Test cases may have preconditions and postconditions, which describe the state of the system (or context of the system) before the test can be performed, respectively after the test finished. If preconditions (or postconditions) are defined, it is job of the test engineer to make sure that the conditions hold.

Setup/Teardown Procedures: Alternatively, preconditions can be described in form of setup procedures (prefixes of test procedures), which ensure that all necessary preconditions hold. For example, by bringing the system in a certain state. Those prefixes of test procedures are also called setup procedures. Accordingly, teardown procedures describe postfixes of test procedures, for example, to return the system to its original state.

The aforementioned aspects are parts that exist in almost all test cases. However, their specific representation in test artifacts strongly depends on the way test cases are executed. For each execution technique, fundamentally different description techniques for test cases are required. In the following, we present two specific test case formats: *Test descriptions* for manual test execution and *test scripts* for automated test execution.

Manual Execution: *Test Descriptions*

We call test case representations that are used for manual execution *Test descriptions*: Test descriptions are the primary artifact for manual test execution (see Section 2.2.1 *Test Execution Techniques*). They act as manuals to guide testers step by step through the execution of a test case. Test descriptions tell the tester how to stimulate the system under test and how to verify whether it responds correctly. Since test descriptions are meant to be read by humans, they are often written in natural language or other, for humans easy understandable notation.

The level of detail of test descriptions may differ and depends on many properties such as the experience or domain knowledge of the tester. Descriptions of test steps may consist of just a few keywords indicating the rough idea of a test step (leaving it to the tester to decide what specifically to do) up to a detailed explanation of what exactly the tester has to do referencing precise details of the user interface. Checks can also be on different levels of abstractions: Decisions whether the response of the system under test is correct may be vague, in form of plausibility checks that are performed by the manual tester, up to very detailed containing exact values.

Table 2.1: Example: Test description.

Wrong PIN Entered Twice		
	Step Description	**Expected Result**
Step 1	Put the ATM card into the card-reader.	The ATM asks for the card's PIN.
Step 2	Enter the wrong PIN '1234'.	The ATM responds that the PIN you entered is wrong and that you have only two more attempts left. Afterwards, the ATM asks for the card's PIN again.
Step 3	Enter the wrong PIN '1234' again.	The ATM responds that the PIN you entered is the same wrong PIN again and that now you have only one attempt left. Afterwards, the ATM asks for the card's PIN again.
...	...	...

Table 2.1 shows a fictive example of a manual system test for an ATM. Goal of the test case is to verify whether an ATM grants just three attempts to authenticate with the card's PIN, despite the same wrong PIN is used for all attempts. The test procedure is denoted in a table notation. Each line represents one test step, consisting of two parts. The column *step description* tells what to perform with the system under test. On the contrary, the column *expected result* describes how to verify the system's correct response.

In this example, it is expected that the manual tester has sufficient domain knowledge to be able to operate the ATM. The test steps are just indicating the actions to perform (e.g., entering a PIN) without specifying what exact interactions the tester has to do to perform the action. Furthermore, test data is embedded in the test description which is on different levels of abstraction: Input values are embedded in the text as precise values (e.g., `PIN '1234'`) whereas the expected response of the ATM is described in an abstract way (e.g., `The ATM asks for the card's PIN`).

Automated Execution: *Test Scripts*

Test scripts are the primary artifact for automated test execution (see Section 2.2.1 *Test Execution Techniques*). Similarly to test descriptions, test scripts define the individual actions that have to be performed to execute a test. However, the goal of a test script is to perform a test case automatically without manual intervention. Therefore, test scripts must automatically interact with the system interface of the system under test and must be able to observe its reactions. Test scripts are denoted in a machine readable format to be executed directly or to be interpreted by a testing framework. All information that is necessary to execute a test case, such as user interface details, have to be available during runtime, while executing the test script.

Listing 2.1 shows an example of a test script written using the open source testing framework *Robot Framework*[2]. The example is taken from the official Robot Framework Demo Test Suite[3] and shows a test case testing the login functionality of a demo web application (test case `Valid Login`, lines 12 – 18). The Robot Framework test definition language follows a keyword-driven approach [Fewster and Graham, 1999]. Each test step (lines 13 – 18) is either a reference to a *keyword* (a reuse component for test procedures, e.g., `Open Browser To Login Page` at lines 21 – 25) or to a base function of Robot Framework (e.g., `Open Browser`, line 22). The example defines reusable test data at the beginning of the test (lines 5 – 9), which is referenced within the test procedure to parameterize test steps. Furthermore, the test cases and keywords contain precise values about user interface elements, for example the technical ids of the text fields for username and password (lines 32 – 36).

2.2.3 Test Execution Modes

We consider a *test execution mode* as the decision which execution techniques are applied to execute which parts of a test suite. More concretely, it defines which test cases or test steps of a test suite are executed manually and which are executed automatically. Execution modes can range from executing all test cases manually to any combination of manual and automatic test execution up to fully automated test suites. In *semi-automated test execution*, as an example, both execution techniques are mixed even within executing single test cases. This means that test scripts as well as manual testers execute test cases collaboratively.

In Figure 2.3, we summarize the concepts and relations between execution techniques, artifact types and execution modes.

2.3 A Generic System Test Process

This section describes a generic system test process as we have seen it in industry. It is based on our experiences of five years of close cooperation with companies developing and maintaining business information systems. Although we do not address each activity in our work, the process shows the big picture describing which system testing activities are performed and which artifacts are created thereby.

[2] `http://www.robotframework.org`
[3] `https://bitbucket.org/robotframework/robotdemo/wiki/Home`

Listing 2.1: Example test script written using the testing framework *Robot Framework*.

```
*** Settings ***
Documentation                      A test suite with a single test for valid login.

*** Variables ***
${SERVER}                          localhost:7272
${BROWSER}                         Firefox
${DELAY}                           0
${LOGIN URL}                       http://${SERVER}/
${WELCOME URL}                     http://${SERVER}/welcome.html

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username                 demo
    Input Password                 mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]                     Close Browser

*** Keywords ***
Open Browser To Login Page
    Open Browser                   ${LOGIN URL}        ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed             ${DELAY}
    Login Page Should Be Open

Login Page Should Be Open
    Title Should Be                Login Page

Input Username
    [Arguments]                    ${username}
    Input Text                     username_field      ${username}

Input Password
    [Arguments]                    ${password}
    Input Text                     password_field      ${password}

Submit Credentials
    Click Button                   login_button

Welcome Page Should Be Open
    Location Should Be             ${WELCOME URL}
    Title Should Be                Welcome Page
```

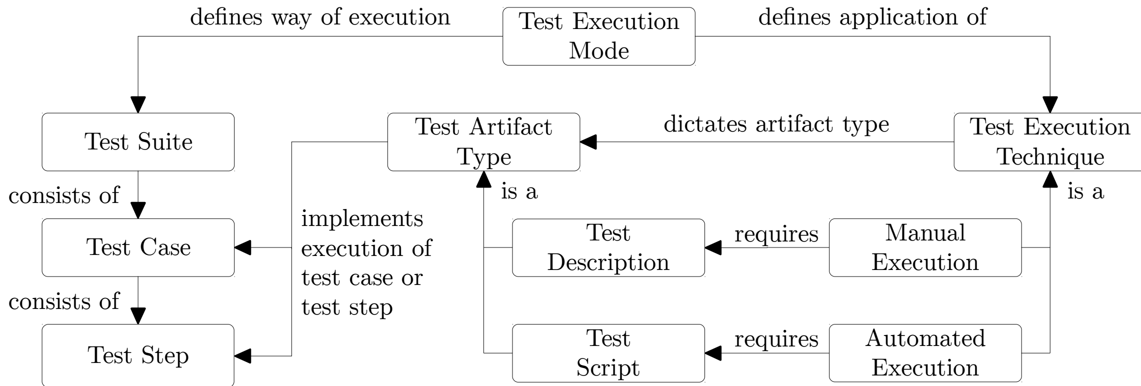


Figure 2.3: An ontology showing concepts and relations of system test execution.

We aligned our terminology on the *fundamental test process* and its terminology, which has been introduced by the International Software Testing Qualifications Board (ISTQB) [International software testing qualifications board, 2011], and the *dynamic test process* defined in the ISO Standard 29119 [ISO/IEC/IEEE, 2013b]. We build up on terms and definition of both sources and complement them with concepts that are important for our work. Figure 2.4 gives an overview of our generic system test process. In the following, all steps are described in detail and we give a running example of all steps:

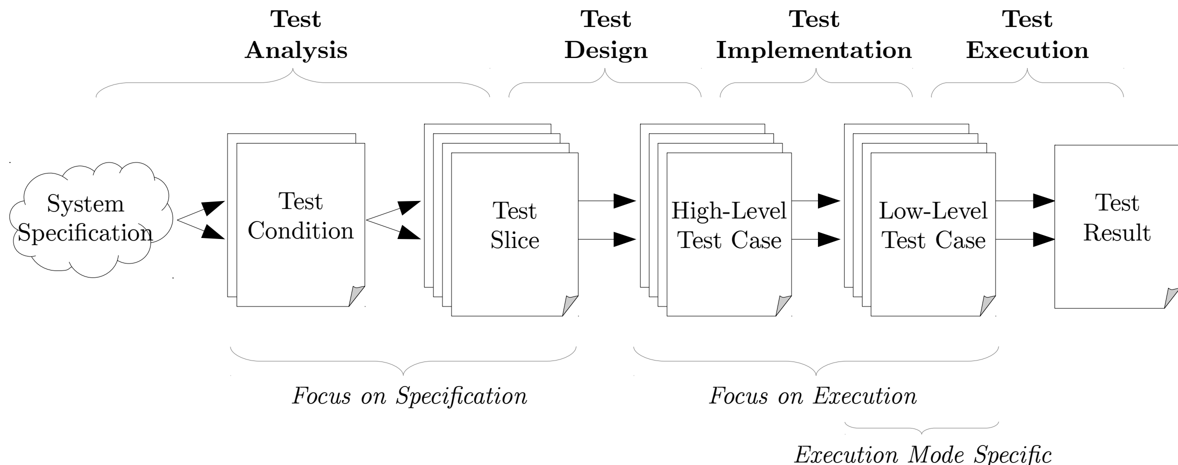


Figure 2.4: The generic system testing process.

2.3.1 Test Analysis

Goal of *test analysis* is to identify *test conditions* [International software testing qualifications board, 2011; ISO/IEC/IEEE, 2013b] created by the *test analyst* based on analysis of the system specification. A test condition is a *"testable aspect of a component or system, such as a function, transaction, feature, quality attribute, or structural element identified as a basis for testing"* [ISO/IEC/IEEE, 2013b]. Once test conditions are defined, those parts of the specification can be identified that are relevant to satisfy the test conditions: *Test slices*.

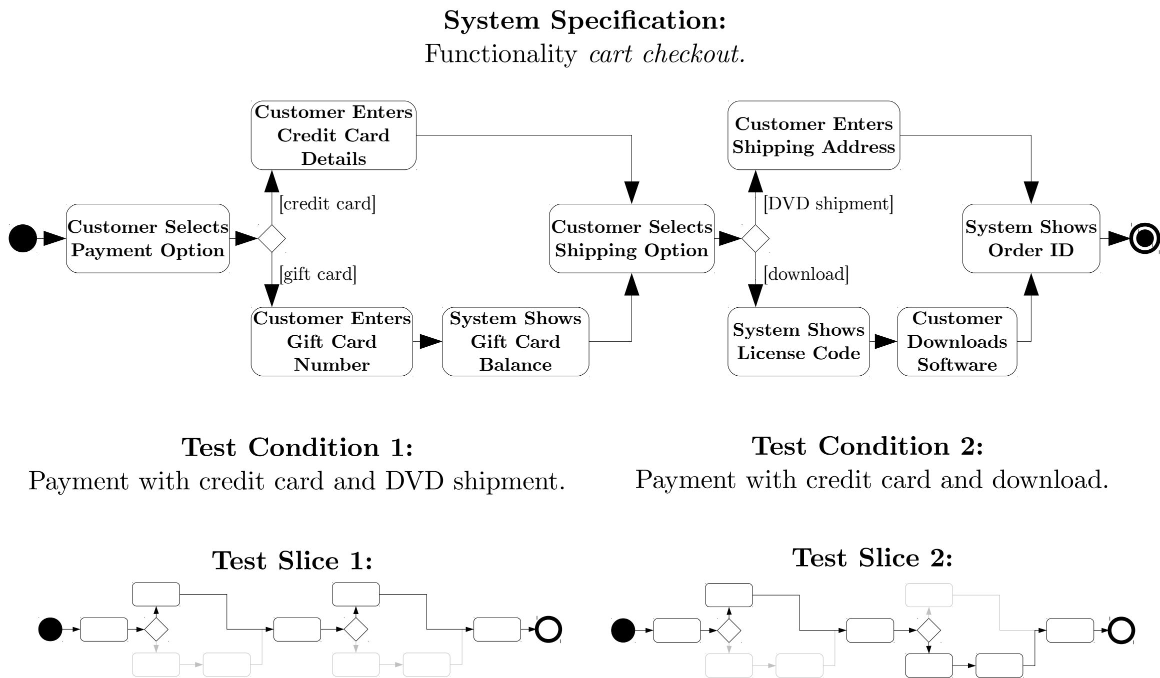


Figure 2.5: Running example of the generic system test process: Test analysis.

Running Example: Figure 2.5 shows an example of a fictive online shop software (same as in Section 1.1.2 *Commonalities in Test Procedures*). The upper part of the figure shows the specification of the functionality *cart checkout* in form of an activity diagram: The online shop allows users to choose from two different payment and two different shipment option. The middle of the figure shows two test conditions covering different parts of the cart checkout functionality: Test condition 1 aims at testing the combination of credit card payment combined with DVD shipment. The second test condition combines credit card payment with online download of the chosen software. Applying both test conditions to the system specification will lead to the two test slices indicated at the bottom of the figure.

2.3.2 Test Design

In the next step, the test analyst extends each test slice to *high-level test cases* [International software testing qualifications board, 2011] by enriching them with further information such as test procedures and test data. The goal of high-level test cases is to focus on the operationalization of test slices and to define the conceptual structure of test cases especially their test procedures. However, high-level test cases are just outlines of test cases and are still abstract and lack implementation aspects, which are necessary for test execution.

Remark: Based on our experiences, test conditions and high-level test cases are often just intermediate artifacts and are not necessarily written down and persisted. In many cases, high-level test cases are just intermediate artifacts that exist just in the head of test engineers during test analysis but are not written down in form of (textual) documents.

Running Example: Figure 2.6 sketches how high-level test cases for our running example could look like. For both test slices from Figure 2.5, operationalized test procedures have been outlined in form of abstract actions that have to be performed in order to walk through the corresponding slices. Additionally, preconditions have been satisfied: Before both test slices can be verified, at least one product has to be added to the cart of the web shop.

High-Level Test Case 1: Cart Checkout with Credit Card and DVD Shipment	**High-Level Test Case 2: Cart Checkout with Credit Card and Online Download**
Satisfy precondition:	*Satisfy precondition:*
1. Open application	1. Open application
2. Search for a random product	2. Search for a random product
3. Add product to cart	3. Add product to cart
Test:	*Test:*
4. Checkout the cart	4. Checkout the cart
5. Select credit card as payment option	5. Select credit card as payment option
6. Enter credit card details	6. Enter credit card details
7. Select shipment of DVD as shipping option	7. Select download as shipping option
8. Enter address for shipment	8. Verify the license code
9. Verify that order has been accepted	9. Verify the download link
	10. Verify that order has been accepted

Figure 2.6: Running example of the generic system test process: Test design.

2.3.3 Test Implementation

Since high-level test cases just give a coarse outline of the steps that have to be performed in a test case, they have to be enriched with execution specific information such as user interface details or concrete test data. Those enriched and executable test cases are called *low-level test cases* [International software testing qualifications board, 2011]. At this point, a decision for a test execution mode has to be made since manual and automated test execution require different types of low-level test cases (see Section 2.2.2 *Test Case Representations*): For manual test execution, *test descriptions* are created to guide manual testers (see Section 2.2.2). These test descriptions contain instructions for manual testers and form the basis for manual test execution. For automated test execution, high-level test cases serve as the basis for *test programmers* to program *test scripts* (see Section 2.2.2).

Running Example: Table 2.3 gives an example for low-level test cases for our running example. In this example, both test cases have been realized as manual test cases in form of test descriptions (see Section 2.2.2).

2.3.4 Test Execution

The activity of *test execution* covers the actual test execution by means of running each test case against the system under test. The concrete proceeding of this activity strongly depends on the chosen execution mode since each execution technique executes differently and is based on different types of artifacts: In *manual test execution*, test cases are performed by human beings: the *manual testers*. Test descriptions form the basis for manual execution and guide the manual tester to execute the test procedures and evaluate the system's responses

Table 2.3: Running example of the generic system test process: Test implementation (manual test descriptions).

Test 1: Cart Checkout with Credit Card and DVD Shipment		
	Step Description	**Expected Result**
Step 1	Open the webshop application in a new web browser. Use the URL `http://test-server/webshop`	The welcome screen of the web shop is shown.
Step 2	Use the search field to search to search for a random product.	The system shows a list of found products.
Step 3	Select a random product.	The system shows the product details page for the selected product.
Step 4	Add product to shopping card.	The system shows the shopping card containing the selected product.
Step 5	Trigger the checkout of the shopping cart (use the `checkout` button).	The system offers different payment options.
Step 6	Select credit card as payment option.	The system requests credit card details.
Step 7	Enter credit card details. Use random credit card number startin with `VISA, 4263 ...`	The system asks for shipment option.
Step 8	Select DVD shipment.	The system asks for shipping address.
Step 9	Enter random address, e.g., `TUM, Munich, ...`.	The system confirm, that the order has been received and shows the order ID.
Step 10	Verify that an order ID is shown (ten-digit and numeric).	(nothing to do)

Test 2: Cart Checkout with Credit Card and Online Download		
	Step Description	**Expected Result**
Step 1	Open the webshop application in a new web browser. Use the URL `http://test-server/webshop`	The welcome screen of the web shop is shown.
Step 2	Use the search field to search to search for a random product.	The system shows a list of found products.
Step 3	Select a random product.	The system shows the product details page for the selected product.
Step 4	Add product to shopping card.	The system shows the shopping card containing the selected product.
Step 5	Trigger the checkout of the shopping cart (use the `checkout` button).	The system offers different payment options.
Step 6	Select credit card as payment option.	The system requests credit card details.
Step 7	Enter credit card details. Use random credit card number starting with `VISA, 4263 ...`	The system asks for shipment option.
Step 8	Select online download as shipping option.	The system shows a page listing the download link, the license key for the selected product, and the ID of the order.
Step 9	Verify that the download link is shown.	–
Step 10	Verify that the license key is not null.	–
Step 11	Verify that the order ID is between 0000000000 and 9999999999.	–

(see Section 2.2.2). In *automated test execution*, tests are programmed as autonomously executable test scripts. Those test scripts stimulate the user interface of the system under test to process the test procedures. Test scripts run autonomously without intervention by manual testers (see Section 2.2.2).

2.3.5 Maintaining System Test Artifacts

In long-living systems, requirements will likely change over the system's life-span. This leads not only to modifications of the specification and the system under test but also to adaptions of system tests. Reasons for changes are manifold: If the requirements change, high-level as well as low-level test cases have to be adapted to be able to verify the changed implementation as well as to verify that no other functionality has been altered accidentally (regression testing). But also changes, such as simple modifications of the user interface, might cause adaptions of low-level or even high-level test cases.

All this is not only leading to repeated test analysis, design, and implementation (including adaptions of existing tests) but also to repeated test execution. We call this sequence of test implementation, test execution and test maintenance the *system test case life-cycle* (see Section 1.1.1 *The System Test Case Life-Cycle*).

2.4 Similarity of Test Cases

This section gives an overview of similarity in system testing artifacts. Using a running example, we explain step by step how similarity arises in test slices and is carried over to semantical and syntactical similarity in test cases.

2.4.1 Similarity in Test Slices – Commonality of Test Procedures

In the previous section, we presented a running example of an online shop and exemplarily showed how two test cases are derived for the functionality *cart checkout*: One test case shall test the cart checkout functionality with DVD shipment, the other shall test the same functionality with online download as shipment option. During test analysis, two test slices have been derived, showing those parts of the specification that are affected. At this stage, the actual test case documents do not exist since the tests are still on the conceptual level in form of test slices. Nevertheless, similarity between the future (high-level and low-level) test cases is already visible: They will execute identical parts of the functionality, which will result in similar sequences of test steps.

Figure 2.7 shows both test slices of our running example and highlights the overlap between them (gray highlighted). In this example, the overlap results in a common prefix and suffix in both paths through the activity diagram.

In the remaining of this work, we call similarity of test cases that origins from overlap in test slices as *commonality of test procedures*. With this term, we refer to similarity on the conceptual level of test cases that is independent from any artifactual representation (see Figure 2.8).

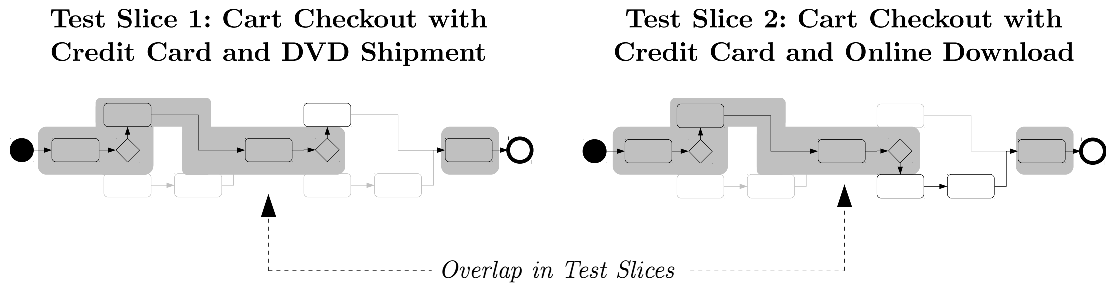


Figure 2.7: Example: Overlap in test slices.

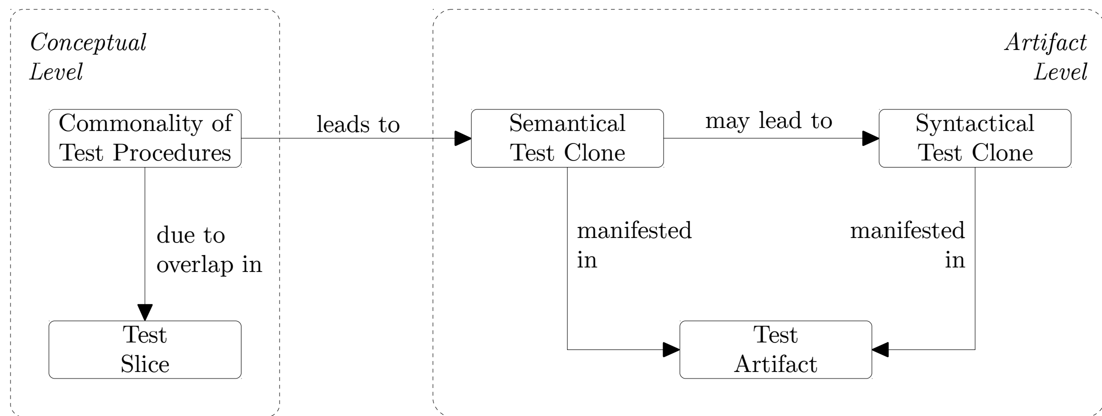


Figure 2.8: Test similarity on the conceptual and artifact level.

2.4.2 Similarity in Test Cases – Semantical Test Clones

Figure 2.9 shows two manual test cases from the running example, which we introduced in the previous section. The test cases realize the test slices shown in Figure 2.7. Since the first parts of both test slices are similar (highlighted parts in Figure 2.7), both test cases will execute similar parts of the functionality. Hence, the beginning of the test case's test procedures will be similar too. More specifically, the test steps 1 to 7 and the last test step of both test cases (see Figure 2.9) describe identical tasks: The description and the expected result guide testers to perform the same tasks (respectively validations) with the system under tests.

In the remaining of this work, we call sub-sequences of test procedures (test steps or sequences of test steps), that lead to identical behavior of manual testers respectively test scripts as *semantical test clones*[4]. With this term, we refer to similarity on the artifact level of test cases (see Figure 2.8).

2.4.3 Similarity in Test Cases – Syntactical Test Clones

When test engineers create test artifacts that have common parts, they often reuse parts of already existing test cases, for example, by copying and pasting them. This leads to test procedures that are partly not only semantically but also syntactically similar. Looking at our running example in Figure 2.9, Step 1 to 7 of both test cases are syntactically identical: Each test step has exactly the same text within its step description and expected result. On

[4]For source code, semantical clones are also called *simions* [Deissenboeck et al., 2012; Juergens et al., 2010b]

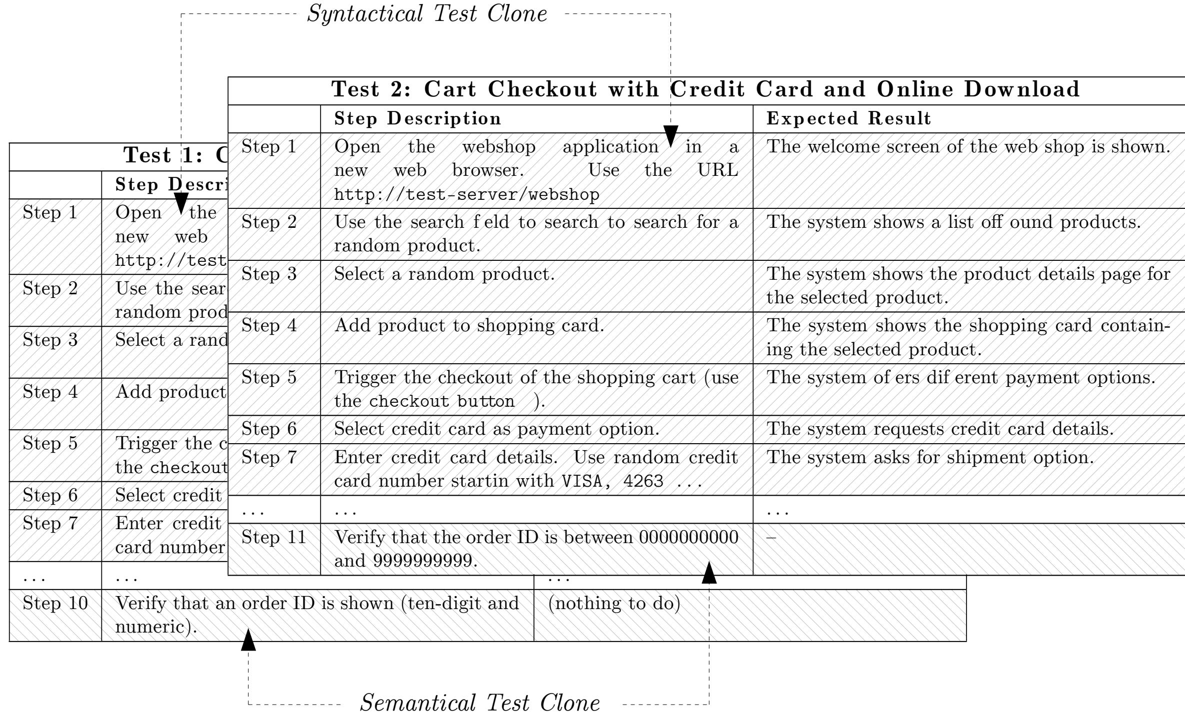


Figure 2.9: Example of semantical and syntactical test clones.

the contrary, the last steps of both test cases are semantically identical too, however, both steps have been phrased differently (their syntactical representation differs).

In the remaining of this work, we call sub-sequences of test procedures (test steps or sequences of test steps), that are syntactically identical as *syntactical test clones*. With this term, we refer to similarity on the artifact level of test cases (see Figure 2.8).

More formally speaking, we define a non-empty sub-sequence *sp1* of a test procedure as a syntactical test clone if another sub-sequence *sp2* of any test procedure exists for that a given similarity predicate $syntacticalsimilarity(sp1, sp2) \rightarrow \{true, false\}$ is true. However, the characteristic of the similarity predicate $syntacticalsimilarity(sp1, sp2)$ strongly depends on the description technique that is used: For manual test cases, for example, two sub-sequences of test procedures might be similar if the natural language text of the test steps is syntactically identical. For test scripts, for example in keyword-driven style (as the example in Listing 2.1), two sub-sequences might be similar if they consist of identical sequences of similar keyword-calls and parameters.

2.4.4 Clone Terminology and Metrics

In the following, we introduce common terminology and metrics that are used in our studies.

Clone Pairs and Clone Classes

We define the terms *clone pair* and *clone class* for testing artifacts by adapting the corresponding definitions from the survey of Roy and Cordy [2007]: *"A* clone pair *is a pair of portions or fragments of artifacts which are identically or similar to each other"*. A *clone*

class (often called *clone group*) is *"the maximal set of artifact fragments in which any two of the fragments form a clone pair"*.

Ungapped and Gapped Test Clones

Koschke [2006] and Roy and Cordy [2007] introduce four levels of similarity predicates, which define similarity on different levels of rigorousness. However, both authors focus on source code clones. We adapt their idea of hierarchical levels of similarity to system test artifacts and define two types of test clones: *Ungapped test clones* and *gapped test clones.*

Ungapped Test Clones: We define *ungapped test clones* as identically sub-sequences of test procedures. However, in our definition, ungapped test clones may differ in formatting aspects that do not change the semantics of the artifacts. To what extent those parts differ depends on the language in which the artifacts are written, however, we expect that the meaning of artifact fragments is not affected by the changes.

For test descriptions in natural language, this may cover sentences, paragraphs or parts of both that are syntactically identical apart from formatting aspects such as white spaces or line breaks. For test scripts, this covers test scripts or parts of test scripts that are syntactically identical apart from aspects that do not influence the semantics of the test execution, such as different code formatting or added or removed comments.

Gapped Test Clones: We define *gapped test clones* as ungapped test clones that, additionally, may differ in changed, added, or removed parts of the clone, for example, changed, added, or removed test steps.

A common phenomena in test artifact is, that (parts of) test cases differ only in test data, which is embedded in test cases (see Chapter 4 *Clones in Manual System Tests* for more information). Using gapped clone detection techniques, those parts can be found.

Clone Metrics

In the following, we introduce common clone metrics that are used in our studies:

Number of Clone Classes: The number of clone classes detected within a set of artifacts.

Size of a Clone Class: The number of clones within a clone class.

Cloned Size: The size of those parts of artifacts that are affected by cloning. The unit of this metric can be, e.g., number of natural language words, number of text or script lines, or number of script statements.

Redundancy Free Size: The absolute size of a set of artifacts, in which cloned parts are counted just once[5]. Assuming that all clones can be removed perfectly, this metric indicates the minimal size a set of artifacts could have.

Clone Coverage: The probability that an arbitrarily chosen part of an artifact is affected by cloning. It is calculated by dividing the artifacts cloned size by its actual size.

Relative Blow-Up: The relative enlargement of a set of artifacts that is caused by cloning. It is calculated by dividing the actual artifact size by its redundancy free size.

[5]In code clone detection, this metric is often called *redundancy free source statements (RFSS).*

2.5 Summary

This chapter integrated this work into common terms and terminology of system testing. We introduced the system testing process this work is based on and defined all relevant artifacts of this process. The later part of this chapter introduced basic concepts and notions of similarity and cloning in test artifacts.

Chapter 3

State of the Art

In the following sections, we discuss existing approaches that address the same problem as we do in this thesis, namely, reducing the effort for the system test case life-cycle activities test creation, test execution, and test maintenance (see Section 1.1.1 *The System Test Case Life-Cycle* and Section 1.4 *Contributions*).

We start with approaches that follow the most obvious way by performing testing activities automatically and thereby making them more efficient. In Section 3.1 *Automating Testing Activities*, we describe the state of the art of automated test derivation and test execution techniques.

Automated testing techniques have advantages and disadvantages though. In Section 3.2 *Balancing Automation in Testing*, we present approaches that help balancing the costs and benefits of automated testing techniques. More specifically, we present the state of the art of effort estimation techniques, which form the basis for cost-benefit estimations.

However, additional to using automation to perform testing activities more efficiently, there are also approaches that follow a complementary approach to reduce testing effort, namely, by identifying and minimizing unnecessary testing tasks. In the third part of this chapter (Section 3.3 *Optimizing Goals of Test Suites*), we present the most influential approaches to optimize and reduce test suites by avoiding unnecessary testing tasks.

Once automated techniques are implemented, and unnecessary tasks have been reduced, at the end of the day, test cases have to be executed and maintained. In the last part of this chapter (Section 3.4 *Optimizing Representations of Test Suites*), we present approaches to optimize the representation of test artifacts so that they avoid unnecessary additional costs. We present the concepts of refactorings and smells, which help to reduce unnecessary overhead while executing and maintaining tests.

3.1 Automating Testing Activities

The following subsections present approaches aiming at making test activities more efficient by automating them. We distinguish those approaches based on the testing activity that are automated. Subsection 3.1.1 briefly summarizes approaches that automate the creation of high-level test cases by applying automated test derivation techniques. Subsection 3.1.2 summarizes approaches and design techniques that automate the execution of test cases.

3.1.1 Automated Test Derivation

Automated test derivation aims at applying techniques to derive test cases, test input data, or expected test results (output data) automatically. The most common approach is model-based testing:

Model-Based Testing

In their book *Practical Model-Based Testing*, Utting and Legeard [2010] describe model-based testing as *"automation of the design of black-box tests"*. It relies on explicit behavior models in which the intended behavior of the underlying software and its environment is encoded [Utting et al., 2012]. Common types of models are finite state machines or data-flow graphs. Using those models, a variety of algorithms exist that derive test procedures as well as input and output data from those models [Broy et al., 2005; Lötzbeyer and Pretschner, 2000]. The types of algorithms range from random input data generation up to search-based algorithms or model-checking approaches [Utting et al., 2012].

Using automated test derivation techniques, large numbers of test cases can be generated without spending much effort on each individual test case. The main effort is spend on the creation of the underlying data bases from which the test cases are generated (e.g., behavior models and coverage criteria). As a consequence, automatically generated test cases will never be subject to maintenance since they can simply be regenerated whenever the underlying models change. Using automated test derivation techniques, test maintenance shifts from the actual test cases to the underlying data bases from which tests are generated.

The goal of automated test derivation approaches is to efficiently derive (abstract) test cases. The contributions presented in this thesis are aiming at a different goal and do not change existing test derivation processes. Instead, we are supporting the following testing activities, such as, creating concrete test cases, as well as executing and maintaining them. However, we consider our contributions as complementary to existing and established test derivation processes.

3.1.2 Maintainable Automated Test Execution

Automated test execution aims at applying automated techniques to run test cases without manual interaction. Thereby, test execution effort can be reduced. However, a big challenge is to create automated test cases efficiently but also in a way that they are easy to maintain. Several approaches exist addressing this challenge in different ways:

Capture/Replay

Test scripts form the basis to execute test cases automatically. They consist of sequences of low-level interactions with a system's user interface, for example, clicking on buttons or typing concrete test data into a text field (see Section 2.2.2 *Test Case Representations*). *Capture/replay tools* [Hicinbothom and Zachary, 1993] aim at creating test scripts automatically by recording all low-level user interface interactions a tester is doing while performing a test case manually once. Afterwards, the recorded test scripts can be executed automatically as often as desired. Although this approach allows fast creation of test scripts, it reveals disadvantages on the long run: Generated test scripts are poorly maintainable and tend to be brittle.

Memon and Soffa [2003] report on a study in which they created a large number of capture/replay test scripts for two software systems. After switching both systems to newer releases, 74% of all generated test scripts did not work anymore due to changes in the graphical user interfaces.

Leotta et al. [2013b] performed experiments comparing creation and maintenance effort for manually programmed and capture/replay test scripts. They generated test scripts in both ways for six open-source software systems. Similar to Memon and Soffa [2003], they used the systems' release history to simulate the systems' evolution and reproduce realistic maintenance activities. The experiments have shown that capture/replay tools speed up test script development up to 112%. However, up to 51% of maintenance effort could be saved by programming test scripts manually. Furthermore, it turned out that the break-even point for creating test scripts manually is reached rather early: After one to three releases, creating test scripts manually pays off.

Page Object Pattern and Component Abstraction

A common design technique to ease maintenance of test scripts is the *page object pattern*. Goal of this pattern is to make test scripts flexible to user interface changes by encapsulating user interface details using object-oriented programming techniques. The pattern dictates to program *page objects* for each dialog of a user interface. Each of those page objects shall provide *services* that encapsulate the low-level test script code to interact with user interface elements. The actual test cases are thus free of low-level script code since they are interacting just with page objects.

Leotta et al. [2013a] report on a study comparing the impact of the page object pattern to the maintenance effort of test scripts. For a given web-application, the authors implemented a test suite twice, with and without applying the page object pattern. Afterwards, both test suites have been adapted to a newer release of the same software system. The results of the study imply that the adaption time could be reduced using the page object pattern by the factor of three.

Chen et al. [2008] enhanced the idea of page objects by introducing *component abstraction*. Instead of using page objects only to encapsulate low-level test script code, Chen et al. propose to use page objects also for higher level application functionality by creating hierarchically composed service objects. Therefore, not only low-level test script code, but also calls of object services can be reused.

Keyword-Driven Testing (KWD)

Another design technique for test scripts is *keyword-driven testing (KWD)* [Fewster and Graham, 1999]. Goal of this technique is to make test scripts efficient to maintain and ease creation of test cases.

The fundamental concept of keyword-driven testing is to add an additional level of abstraction in automated test cases. Test automation is separated into two parts: (1) A set of reusable snippets of test scripts. Each snippet is identified by a unique name, the so called *keyword*. (2) *Test cases* which do not contain test script code themselves. Instead, they are described by sequences of keywords. To execute a keyword-driven test suite, an interpreter reads each test case and executes the referenced keyword snippets.

Keyword-driven testing promises to have two major advantages: (1) Since test cases are free of implementation details, new test cases can be easily created by non-technical staff. (2) The concept of keywords acts as an explicit reuse mechanism for test script code. This allows reducing cloning in test scripts and therefore reduce unnecessary duplication of maintenance effort. However, key challenge of keyword-driven testing is to identify the right abstraction level for keywords [Fewster and Graham, 1999].

Test Cases as User Stories in Behavior-Driven Development (BDD)

In *behavior-driven development (BDD)* [North, 2006], a special kind of the software development philosophy *test-driven development (TDD)* [Beck, 2003], early and continuous testing is used to support short development cycles. Key aspect of behavior-driven development is to write acceptance test cases in natural language in a similar way as user stories are written. North [2006] proposes a semi-structured notation for acceptance test cases in which all test cases follow a similar structure. He proposes to use a template to construct test cases containing just three parts: *"Given [initial context], when [event occurs], then [ensure some outcomes]"*. To make test cases automatically executable, similar to keyword-driven testing, there is a test script snippet for each constituent of a test case.

Model-Based System Test Instantiation by Hauptmann and Junker

Hauptmann and Junker [2011] proposed an approach to constructively design test scripts that are easy to maintain. The presented approach strictly separates test logic from user interface details: System interfaces are modeled as *abstract system models* having *abstract system interactions* acting as input and output channels to exchange data with the environment. Consequently, test cases are denoted just as sequences of inputs and outputs using those abstract system interactions. Additionally, a test suite is equipped with a *GUI model* in which information about the user interface is encoded. The GUI model consists of several sub-models dealing with different aspects of graphical user interfaces, such as navigation between dialogs, the layout of dialogs or concrete user interface widgets. To execute a test suite, a virtual machine interprets each test case and uses the information from the GUI model to concretize abstract system interactions to concrete test script code.

By separating test logic and user interface information, both can be stored in different locations instead of being spread among both artifacts. Thereby, redundancy in test cases is avoided. Therefore, changes to test logic and adaptions of GUI design details can be performed efficiently.

3.1.3 Discussion and Relation to this Thesis

Automated test execution promises to save testing effort by automating the labor intensive activity of executing test cases. However, automated test scripts are expensive to create and maintain. Whether automated test execution will pay off depends on a variety of project specific factors. In Chapter 5 *Choosing Execution Modes*, we present a effort estimation model that can be used to find out whether automation will pay off for concrete project settings.

One of the major challenges of automated test execution is to design test scripts in a way that they are easy to adapt and to maintain. Most existing approaches address this by dictating certain design patterns to implement maintainability in a constructive way. The

contributions presented in this thesis follow a fundamentally different approach: We strictly do not change existing design patterns. Instead, we propose analytical methods to support creating high quality test artifacts: In Chapter 6 *Test Refactoring Using Grammar Inference*, we present an approach that helps test engineers in extracting clones from automated test cases. Goal of this approach is to enable efficient clone removal and thereby make test cases easier to maintain. In Chapter 7 *Natural Language Test Smells*, we present an approach to identify quality defects in natural language test cases regarding test maintenance and executability.

3.2 Balancing Automation in Testing

In the following section, existing approaches are presented that support test activities by applying structured techniques for test planning and effort estimation. These approaches can be used to balance different testing approaches (e.g., test automation). First, existing effort and cost estimation approaches for software development in general (Subsection 3.2.1) and for software testing (Subsection 3.2.2) are discussed. Afterwards, the state of the practice in choosing test execution modes is presented (Subsection 3.2.3).

3.2.1 Basic Software Effort Estimation Approaches

Several basic approaches exist to estimate overall software development effort: A widely known approach is the *function point analysis (FPA)* [Garmus and Herron, 2001], which measures the amount of business functionality a software system provides to users. Function points are used as unit of measurement by domain experts to estimate the complexity of each user requirement. This allows estimating relative differences of implementation effort between requirements.

Mohagheghi et al. [2005] introduce the *use case point analysis (UCP)* as an extension of function point analysis. Similarly to FPA, goal is to predict software development effort in a top-down way. The main difference to FPA is that UCP is applied on requirements being written as use cases. UCP is commonly used as part of the rational unified (development) process (RUP) [Kruchten, 2003] and is applied on specifications using the unified modeling language (UML) [Rumbaugh et al., 2004].

A conceptually similar method are *story points* [Cohn, 2004]. Similar to FPA and UCP, an abstract metric (story points) is used to determine implementation effort. Story points are a relative measure for user stories, which are commonly used in agile methods. However, story points are not used to be directly translated to actual effort, such as working hours. They are mainly used to plan sprints by comparing efforts of future stories against the historical data in past stories.

Another common approach is the *constructive cost model (COCOMO)* initially introduced by Boehm et al. [2000]. COCOMO uses parameters from historic projects and characteristics from the current project to estimate effort for software projects in form of costs.

3.2.2 Estimating System Test Effort

Several approaches exist to estimate test effort and costs using techniques very similar to function point analysis:

Execution Points by Aranha and Borba

Aranha and Borba [2007] propose a technique to estimate execution effort of manual test specifications similar to the ones specified in Section 2.2.2. Every step within a test suite is analyzed based on a set of characteristics to obtain its execution complexity. This complexity is expressed in form of *execution points*, which are assigned to the test cases. Having the overall number of execution points of a test suite and historical data or expert estimations of the effort to execute the whole test suite allows breaking down the overall effort and estimating execution effort for individual test cases.

Compared to our effort modeling approach (Chapter 5 *Choosing Execution Modes*), Aranha and Borba [2007] focus just on test execution and do not take into account efforts for creating and maintaining test cases. Furthermore, the approach does not allow comparing manual execution with automated execution or mixes of both techniques.

Similar Tests have Similar Costs by Zhu et al. and Xiaochun et al.

Zhu et al. [2008] and Xiaochun et al. [2008] estimate test execution effort based on the principle *"similar tests have similar costs"*. They propose to build up a database storing historical values about test runs grouped by test execution complexity and the skill levels of the corresponding testers. Using machine learning algorithms, they predict execution effort for new test cases.

Similar to Aranha and Borba [2007], Xiaochun et al. [2008]; Zhu et al. [2008] use historical test execution data to predict execution effort for other tests. However, also this approach does not consider test creation and maintenance effort.

Adjusted Use Case Points (AUCP) by de Almeida et al.

De Almeida et al. [2009] propose to estimate test effort based on underlying use cases. For each use case, the authors rank the number of actors of the use cases, the number of scenarios, as well as each use case's technical and environmental factors, each, based on a set of ranking schemes. Having this information, they calculate *adjusted use case points (AUCP)* for each use case on which overall testing effort can be predicted. The proposed method predicts testing effort for a whole test suite including test planning, test design, test execution and report as whole.

The effort estimation method we propose in Chapter 5 *Choosing Execution Modes* follows a similar approach, but focusses only on effort that influence the decision for or against execution modes, namely, expenses for creating test cases, executing them and maintaining them (see Chapter 1 *Introduction*). Furthermore, our approach relies on a simplified classification of the effort of a single test step. Thereby, we expect to achieve easier application.

G-O Model based Model by Pham and Zhang

The approach of Pham and Zhang [1999] relies on a different type of model. The authors propose an effort estimation model to determine when to stop testing and to release the

software instead. The proposed model is based on a *Goel-Okumoto NHPP model (G-O model)* as a software function to extrapolate the reliability of a software system. The approach is based on the idea that the more time is spent on testing, the more errors will be found and removed which in turn leads to a more reliable software system in terms of risk of failures. The proposed model considers testing costs, error removal costs and risks of failures, each based on empirical data of previous experiments and approximated by mathematical functions.

The presented approach considers testing effort on a rather coarse-grained level since it is just one factor among others. The effort estimation method we are presenting in Chapter 5 *Choosing Execution Modes* explicitly focusses on all system test execution life-cycle activities (see Chapter 1 *Introduction*).

3.2.3 Choosing Test Execution Modes

Several guidelines and best practices exist to decide how to apply manual and automation execution beneficially:

Lessons Learned in Software Testing by Kaner et al.

Kaner et al. [2002] agree, that the decision when to rely on manual and when to use automated test execution should be based on a cost benefit analysis. As a general idea, Kaner et al. propose to balance test preparation costs, such as test script development or preparation of manual test cases, with the actual test execution costs which grow linearly with the number of test runs. However, the authors emphasize the different capabilities of both execution techniques: Whereas manual testing has its strength in exploratory testing by uncovering unexpected flaws of the software system, automated test execution has the advantage of replicable test cases.

We agree on the conceptually different concepts of exploratory testing and formal testing (see Chapter 2). However, following our key principles (see Section 1.3.1 *Key Principles*), we take test cases and the activities that are related to them as starting points for our work. Therefore, in the remainder of this thesis, we focus on formal testing only. Furthermore, in our context, we consider both execution techniques as substitutable for formal regression testing.

Economics of Test Automation by Forrester Research

In the year 2005, Forrester Research published a report on experiences in automated functional testing tools (Schwaber and Gilpin [2005]). In this report, the authors proclaim to consider the overall costs for automated test execution as the sum of the costs of the testing tools, the labor to create and maintain test scripts. The authors furthermore proclaim to automate tests just if the costs for test automation are smaller compared to the overall costs of manually executing the same tests similarly often.

We believe that test automation opens up new possibilities in software development which are difficult to cover in an pure economic comparison. For example, to enable continuous deployment strategy, tests are required to run automatically after each build. Nevertheless, Schwaber and Gilpin support the general idea of weighing up costs for manual and for automated test execution in order to support decision making for execution modes. However, the authors are not proposing a structured process nor a tool (such as a cost model) to support reasonable and replicable decision making.

Cost Benefits of Test Automation by Hoffman

Hoffman [1999] reports on experiences introducing test automation in industry. Based on this report, to introduce test automation successfully, it is important to understand the potential benefit as well as the costs and risks of test automation before introducing them. Based on the experience of the authors, realistic expectations are key to success. The authors emphasize the difference in developing and maintaining automated and manual tests. Both execution techniques have positive and negative aspects that have to be considered. Hoffman proposes to consider the *return on invest (ROI)* to make decisions for or against test automation. The author divided testing costs into *fixed automation cost factors* and *variable automation cost factors*. The author furthermore introduces a simple cost model to calculate the ROI in order to choose manual or automated test execution.

In contrast to our approach, Hoffman [1999] suggests to automate the whole test suites, while we let the test engineer decide which parts of a test suite to automate and which additional parameters to take into account.

Opportunity Costs by Ramler and Wolfmaier

Ramler and Wolfmaier [2006] furthermore support that a gross underestimation of development and maintenance effort of automated test execution is the reason for failed projects. However, the authors state that oversimplified cost models as the only decision base for selecting execution modes are not suitable, since often, contextual project information is not included in the calculation. Instead, they propose to consider *opportunity costs* in selecting execution modes. Ramler and Wolfmaier introduce an alternative cost model based on linear optimization. The presented approach is based on the existence of a *fixed testing budget*, which covers costs for manual and automated testing. Based on the author's experience, testing budgets tend to be too small for both, to automate all test cases as well as to execute all test cases manually. The presented cost model finds the optimal trade-off between manual and automated test execution with the goal to get the optimal number of executed test cases from a given testing budget.

In our work, we do not aim at dictating the test engineer what to automate, since the goal of our approach is to provide a more solid basis for decision-making. Additionally, our cost modeling approach explicitly supports execution modes in which manual and automated execution can be mixed beneficially.

3.2.4 Discussion and Relation to this Thesis

Cost-estimation techniques are widely applied for planning and managing software projects. In the area of testing, many approaches exist to predict the effort of testing or to manage testing activities. For example, they are used as test-end criteria. However, to choose between manual or automated test execution, existing approaches focus on guidelines and best practices. None of them provide methods or tools that can be used by test engineers and support choosing execution modes.

In Chapter 5, we introduce an approach using cost-estimation techniques to support choosing between manual and automated test execution. We present a cost-model that enables comparing execution modes and can be used by test engineers as quantitative input to bal-

ance between different execution modes at hand. Additionally, compared to other approaches, we focus on all relevant costs of the system life-cycle activities test case creation, execution and maintenance.

3.3 Optimizing Goals of Test Suites

The following section presents approaches that reduce testing costs by optimizing test suites. We distinguish between two general approaches: *Test suite reduction*, aims at shrinking test suites by removing obsolete test cases. Thereby, test execution and maintenance effort can be reduced. Another common approach is *test suite optimization*, which aims at making test suites more effective by reordering test cases to satisfy test coverage criteria early. Both approaches share a number of common properties: They automatically transfer a given test suite in a new, optimized test suite.

Test Suite Reduction with Representative Sets by Harrold et al.

Harrold et al. [1993] and Rothermel et al. [1998] propose a heuristic approach to select a representative subset of test cases from a test suite that provide the same coverage as the initial test suite. The basic idea of the presented approach is to remove *unnecessary test cases* from the test suite, which are either obsolete or redundant. Harrold et al. consider test cases as *obsolete* once the reason for the test case's inclusion in the test suite has been removed. On the other side, a test case is considered as *redundant* if other test cases in the test suite provide the same coverage of the program. Both definitions are based on the existence of a test coverage criteria. Concretely, their approach is built on the existence of a set of *testing requirements* and on the information which test case satisfies which testing requirements. Harrold et al. propose a heuristic algorithm to categorize test cases based on their degree of *essentialness* and select test cases starting from most essential to least essential.

Test Suite Reduction with Representative Sets by Chen and Lau

Chen and Lau [1998, 1996] propose a technique to find subsets of test cases that are as powerful as the initial set of tests concerning their ability to satisfy testing requirements. Similarly to Harrold et al. [1993], their approach is based on the existence of the information which test case satisfies which testing requirement. They furthermore build up on the idea of *redundancy* of test cases by which they refer to test cases having testing objectives that are already satisfied by other test cases of the test suite. Chen and Lau propose a greedy strategy to find a subset of test cases that is still able to satisfy the same test requirements as the initial test suite. Similar to Harrold et al. [1993], this subset is called *representative set*. Their approach consists of three steps: First, all test cases of the initial test suite are selected which are considered as *essential*, which means, test cases that cannot be replaced by other test cases. Second, test cases are reconsidered having a 1:1 redundancy relation. Third, a greedy strategy is used to select test cases that satisfy a maximum number of requirements. The greedy algorithm follows the strategy of finding a global optimum by a series of local optima.

Vector-Based Test Suite Reduction by Yu et al.

Based on Yu et al. [2008], existing test suite reduction approaches are mostly *statement-based* and define test case requirements as coverage of statements. Yu et al. present an approach that reduces test suite by selecting a subset of test cases that are executing the same statements as the initial test suite. Yu et al. propose a *vector-based* approach to reduce test suites which uses *statement vectors* as basis to define test requirements. Hence, their test suite reduction approach selects subsets of test cases that are executing the same *statement vectors* as the initial test suite would. Yu et al. furthermore report on an empirical study in which they analyze the effect of test suite reduction techniques on fault localization algorithms.

Reducing and Prioritizing Mutation Testing by Offutt et al.

Offutt et al. [1995] propose test suite reduction and prioritization algorithms for *mutation testing*, which is a common technique to measure the ability of a test suite to identify changes of a software system: Variants of the system under testing are derived automatically, differing just in minor aspects. Goal of testing is to identify (*kill*) as many mutations as possible. Algorithms, such as *constraint-based test data generation (CBT)*, aim at generating test cases which explicitly address one mutation each. Using such approaches, the number of test cases is limited to the number of mutants. Offutt et al. propose an approach based on experiments to identify redundant test cases for test suite reduction. The authors propose to run tests in different orders with the goal to increase the difference by means of number of killed mutants. Offutt et al. [1995] furthermore report on an empirical study in which they applied their approach. In their study, test suites could be reduced by 33% in terms of number of test cases.

Reducing and Prioritizing MC/DC Test Suites by Jones and Harrold

Jones and Harrold [2003] propose test suite reduction and prioritization algorithms for test suites targeting at *modified condition/decision coverage (MC/DC)*. Existing test reduction and prioritization approaches cannot be applied directly for MC/DC test suites. Compared to conventional testing, the benefit of test cases for a test suite cannot be judged by looking at individual test cases; other test cases have to be included in this decision too: MC/DC is a coverage criteria, which analyzes conditions of branches. However, compared to common branch coverage, MC/DC requires every individual part of a condition to be covered by both truth values: true and false. Therefore, for each part of a condition, a pair of test cases has to be known leading to both truth values.

Jones and Harrold propose a test suite reduction and a test suite prioritization algorithm for test suites following MC/DC coverage. The presented algorithm splits a test suite into essential and redundant test cases. However, compared to existing approaches, the definition of essentialness is more complex: a test case is considered as essential if it uniquely covers a truth value of a covered condition after removing any uncovered MC/DC pairs. To reduce a test suite, the proposed approach identifies the *weakest test case* that is not essential (the test case that contributes the least to the testing requirements), removes it and identifies the next weakest test case. The algorithm halts so that the modified test suite still covers all test requirements. To prioritize test cases, the approach recomputes the contribution of test cases after each test case is selected. However, compared to the test suite reduction algorithm, this time the *strongest* test case is selected.

Effectivity of Test Case Prioritization by Elbaum et al. and Rothermel et al.

Elbaum et al. [2001, 2000] and Rothermel et al. [2001] report on empirical studies in which they evaluate the effectivity of a variety of test case prioritization strategies. In their study, the authors evaluate the quality of the analyzed strategy to sort test cases based on their fault detection rate: The goal is to arrange the order of test cases to reach high fault detection rates early. Elbaum et al. distinguish between *general prioritization* approaches, which sort test cases to reach good results over multiple versions of a software system and *version-specific prioritization*, which focuses on effective ordering of test cases relative to versions that contain multiple faults. They furthermore distinguish *fine-granular* approaches, which base their prioritization on the level of source code statements and *coarse-granular* approaches which operate on the function level. To compare test case prioritization strategies, the authors introduce the metric *average percentage of faults detected (APFD)*, which ranges from 0 to 100 (higher numbers mean faster fault detection coverage). The metric relates the degree of fault detection coverage with the degree of test cases necessary to reach that coverage. The study the authors report on has been performed using the source code and test suites of eight industrial software systems. The results indicate that both types of strategies, fine-grained (on statement level) as well as coarse grained (on functional level), were able to improve the rate of fault detection of test suites. Fine grained strategies tend to be more effective, however, are more costly to realize.

3.3.1 Discussion and Relation to this Thesis

Test suite reduction and optimization are both common techniques to reduce test execution and maintenance effort. The basic idea of both approaches is to identify and exploit optimization potential within the test cases' sequences of test steps. Both approaches result in test suites in which test cases have been removed or merged.

The contributions presented in this thesis follow a fundamentally different approach by not trying to change test procedures or the order in which test steps are executed. Following this fundamental guideline, the contributions of this thesis are not leading to changed behavior of test suites. Instead, they are improving just the artifact representation of test cases. Thereby, we expect our contributions to be easier adoptable in practice since they are not in conflict with surrounding activities such as test derivation. More specifically, in Section 5, we propose an approach to optimize the way test cases are executed without changing the test procedures themselves. In Section 6 and Section 7, we are aiming at improving test artifact quality, such as maintainability or executability without changing the order of test steps during test execution.

3.4 Optimizing Representations of Test Suites

In the following subsections, existing approaches are presented that support test activities by improving the quality of testing artifacts. Well-known techniques to constructively improve the quality of software engineering artifacts are *refactorings*. For example, removing clones by extracting them to reuse components. Common techniques to identify opportunities to apply refactorings are *bad smells*. In the following section, we are discussing foundations and applications of refactorings and smells in software engineering artifacts in general as well as on specific test artifacts.

Origin of *Restructuring* and *Refactoring* Software Engineering Artifacts

Chikofsky and Cross [1990] define *restructuring* as transformation of software engineering artifacts from one representation form to another while preserving the same relative abstraction level. Refactorings furthermore preserve functionality and semantics of artifacts, however, may lead to improvements of the artifacts' representation. The term *refactoring* was originally introduced by Opdyke [1992] and refers to restructuring on object-oriented artifacts particularly source code in object-oriented programming languages. The basic idea is to rearrange classes, methods and variables to support future modifications [Mens and Tourwe, 2004].

In the context of software maintenance and evolution, both, restructuring and refactoring aim at improving the quality of software engineering artifacts to support engineering tasks, such as, extensions, reuse and understandability [Mens and Tourwe, 2004]. In software reengineering, restructurings and refactorings are used to bring a software system in a new form, for example, to reimplement a system in a different programming language or a different programming paradigm [Chikofsky and Cross, 1990; Demeyer et al., 2002; Fanta and Rajlich, 1999].

3.4.1 Refactoring and Smells of Source Code and Unit Tests

Refactorings and smells have initially been applied on source code and soon after been used to improve programmed unit tests:

Refactoring Source Code based on *Bad Code Smells* by Fowler

Refactoring is a well-known technique applied in the context of *software aging* [Parnas, 1994]. In this context, Fowler defines refactoring as *"a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"* [Fowler, 1999]. In his book *Refactoring* [Fowler, 1999], he defined 22 refactorings for Java source code. Fowler furthermore initially introduced the concept of *bad smells in code* as an indicator when (and where) to apply refactorings. We define smells in source code as *"certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring"* [Fowler, 1999]. Prominent examples of smells are *Duplicated Code* and *Long Methods.*

The definition of smells clearly distinguishes smells from defects, since defects cannot be removed by performing semantic preserving refactorings. Therefore, smells indicate internal quality problems of the source code such as maintainability, understandability or reusability. Furthermore, smells are not considered as absolute accurate but are treated as indications for quality problems. They are always subject to evaluation by experts.

Refactoring Unit Tests using *Bad Test Code Smells* by van Deursen et al.

Unit testing is one of the basic concepts of *extreme programming (XP)* [Beck, 1999]. The XP process dictates to write unit test cases for each piece of application code that is produced. In XP, unit tests are not only used to verify the source code's correct behavior but also to document the application by giving usage examples [Beck, 2000]. This leads to a huge number of unit test cases, which have to be maintained when the system under test evolves.

Van Deursen et al. [2001] state, that refactoring test code differs from refactoring application code since unit test code has a distinct set of bad code smells that differs to application code. Furthermore, existing refactorings are not sufficient to remove those smells. Consequently, van Deursen et al. introduce *bad [unit] test code smells*, a set of 11 bad code smells that are specific for unit test code. The smells cover quality aspects that are specific to automated and frequently executed unit test cases, such as, test resource handling or correct usage of test assertions. An example is the smell *general fixture* that identifies test cases that erroneously use the same setup methods (often referred to as *test fixture*) for the wrong reasons, for example, just because it is located in the same unit test class. For each of the smells, van Deursen et al. either reference an existing general purpose refactoring of Fowler [1999] or propose an individual unit test code specific refactoring. In overall, van Deursen et al. present 6 *test refactorings* for unit test code. Each of the introduced refactorings aims at improving the understandability and maintainability of unit test code.

The *Test Automation Manifesto* by Meszaros et al.

In their paper *The Test Automation Manifesto* [Meszaros et al., 2003], Meszaros et al. build up on bad unit test code smells introduced by van Deursen et al. [2001]. The authors classify each smell of van Deursen et al. into the two categories: *Code smells*, which are based on the source code of the unit test and *behavior smells*, which appear when executing the tests. The first class is similar to the original code smells by Fowler [1999] and represents parts of the unit tests' source code that might affect maintenance costs, such as the smell *hard coded test data* or *conditional test logic.* The latter class stands for smells that have negative effect while running tests, such as *fragile tests* or *interdependent tests.* Later, in his book *xUnit test pattern* [Meszaros, 2007], Meszaros furthermore adds the category of *project smells*, which indicate the state of the overall health of a project.

Meszaros et al. proclaim the *test automation manifesto*, which defines 12 properties each automated unit test should fulfill, such as *conciseness*, *robustness* or *traceability.* The authors furthermore state that smells may help to improve the quality of unit tests that already exist, however, do not help to guide creating new test cases having high quality. Therefore, the authors introduce 22 rules for test automaters to create high quality artifacts from the beginning. The rules are separated into 4 categories of patterns covering aspects such as *readability*, *robustness*, or *reuse.*

3.4.2 Refactoring and Smells of System Tests

The concept of bad smells has also been transferred to other types of artifacts such as system test cases:

Refactoring TTCN-3 Tests using Metrics by Zeiss et al. and Neukirchen et al.

Zeiss et al. [2006] and Neukirchen et al. [2008] propose refactorings to improve the quality of test suites denoted in the *testing and test control notation (TTCN-3)* [ETSI, 5 06; Grabowski et al., 2003], a programming language for test scripts which is primarily used in the domain of communication systems. In their papers, the authors introduce 48 refactorings to improving the readability, usability, and maintainability of TTCN-3 test cases. 28 of which are adapted from Fowlers initial refactorings for Java source code. Another 21 refactorings are specific

to TTCN-3 language characteristics. Each of the proposed refactorings focuses either on the test's behavior, test data descriptions or on the improvement of the overall structure of the test suite.

Similar to Fowler [1999], Zeiss et al. and Neukirchen et al. also define measures to identify opportunities for refactorings. However, instead of defining a smell for each refactoring, *code metrics* are used to find out where to apply refactorings. Metrics are divided into *size metrics*, such as the number of operators or operands and *structural metrics*, such as the McCabe complexity [Watson et al., 1996] and object-oriented code metrics. In addition to the metrics and the refactorings, Zeiss et al. and Neukirchen et al. report on an implementation of the tool *TTCN-3 refactoring and metrics (TRex)*, which automatically calculates the presented metrics and provides automatic refactoring features.

Similar to our approaches (Chapter 6 *Test Refactoring Using Grammar Inference* and Chapter 7 *Natural Language Test Smells*), Zeiss et al. and Neukirchen et al. do not suggest to perform refactorings automatically and unattended. However, they propose to use both in a semi-automated approach, which guides testers to perform refactorings.

Refactoring TTCN-3 Tests using Code Smells by Neukirchen and Bisanz

Neukirchen and Bisanz [2007] extend the work of Zeiss et al. [2006] and Neukirchen et al. [2008] and build up on their refactorings to remove internal quality problems of TTCN-3 tests. However, Neukirchen and Bisanz state that the basic metrics introduced by Zeiss et al. [2006] and Neukirchen et al. [2008] are not sufficient to detect problematic test code that is error-prone or may lead to quality problems. Instead, they introduce 38 *TTCN-3 code smells* as a more powerful approach to identify where to perform refactorings.

They implement the proposed smells in the tool *TTCN-3 refactoring and metrics (TRex)* just as Zeiss et al. [2006] and Neukirchen et al. [2008]. The authors' ambition is to provide tooling to perform *push-button detections*, which enables an *"everyday usage of an automated issue detection"* to testers. [Neukirchen and Bisanz, 2007]

Refactoring Keyword-Driven Tests (and Component Abstraction) by Chen et al.

Chen et al. [2008] address quality issues in automated system tests (referred to as *GUI test scripts* [Chen et al., 2008; Chen and Wang, 2012]). They focus on test cases following the test script design pattern *component abstraction (CA)* [Chen et al., 2008].

To identify poor quality of test scripts, Chen et al. [2008] introduce a set of 11 *bad test smells*. Some of the smells are taken from Fowler [1999] and transferred to the context of test scripts, for instance, the java code smell *long method* was used as inspiration for the test script smell *long keyword*. Other smells have newly been introduced by Chen et al. [2008], for example, the smell *lack of macro events*, which indicates absence of assertions in test procedures. 6 out of the 11 presented smells are specific to CA test scripts, the remaining 5 can be applied to common KDT as well. Chen et al. [2008] furthermore introduce 16 refactorings to correct quality defects identified by smells. Similar to the smells, the refactorings have been inspired by the Java code smells of Fowler [1999]. Half of the presented refactorings are limited to test scripts using component abstraction. The refactorings have been implemented using the testing tool *GTT* and are automatically executable. However, the tooling does not provide techniques to detect smells automatically in test cases. Therefore, the approach can only be applied in a semi-automated quality improvement setting.

3.4.3 Refactoring and Smells Beyond Source Code and Tests

The concepts of bad smells and refactorings have also been applied to a variety of other types of software engineering artifacts:

Smells in Requirements Engineering Artifacts

Wilson et al. [1997] proposed a technique to analyze requirements specifications in natural language for quality defects. As a model for good quality, the authors define 8 *quality characteristics* that shall be fulfilled by requirements. The characteristics are based on the IEEE Standard 830-1993 [IEEE, 1993] and cover quality aspects, such as, completeness, correctness, modifiability, and unambiguousness. To identify violations of these characteristics, the authors propose to search for terms that indicate quality defects, such as weak phrases. Those *quality indicators* are grouped on their manifestation in *individual specification statements*, such as imperatives, weak phrases, or continuances, or *entire requirements document*, such as the document's size or text structure. Furthermore, the authors present a study investigating the appearance of each quality factor in a set of requirements documents at NASA.

Fabbrini et al. [2001] define a quality model against which requirements specifications in natural language can be checked to identify quality defects regarding the high-level properties *testability*, *completeness*, *understandability*, and *consistency*. Similar to the approach of Wilson et al. [1997], those properties can be evaluated automatically using tangible *quality indicators*, which are syntactical or structural aspects of the requirements documents. Furthermore, Fabbrini et al. developed a tool to find appearances of quality indicators based on techniques from the research field of *natural language processing (NLP)* [Jurafsky and Martin, 2000].

Hussain et al. [2007] follow the idea to use NLP techniques to automatically asses the quality of textual requirements documents. However, instead of programming a tool manually, Hussain et al. apply machine learning algorithms to find quality defects in requirements documents. The authors trained a machine learner to identify ambiguity based on manually classified text samples. The authors implemented their approach in form of the tool *Requirements Specification Ambiguity Checker (ReqSAC)*.

Femmer et al. [2015] propose an activity-based requirements model. The authors propose to tailor quality models context-specific by adapting it to the individual needs of software project. Thereby, stakeholders and the activities performed by them are used to derive quality factors for requirements artifacts. Those quality factors can then be used to identify bad smells in requirements documents by using NLP detection techniques [Femmer, 2013; Femmer et al., 2014].

Domann et al. [2009] and Juergens et al. [2010a] analyzed requirements specifications searching for appearances of cloning, which is a smell appearing in many software engineering artifacts. They adapt existing clone detection techniques from the area of source code clone detection to find duplicated text passages in natural language requirements specifications. The authors report on a study in which they apply their clone detection techniques on 28 requirements specifications from three different companies. The study revealed that in all analyzed specifications, the probability that an arbitrary sentence is duplicated is greater than 10%. Furthermore, cloning is not limited to a specific kind of information since all parts of the documents were equally affected by cloning.

Smells in Spreadsheets

Hermans et al. [Hermans et al., 2012a,b, 2014, 2013] proposed *bad smells for spreadsheets.* Spreadsheets are commonly used by non-professional programmers and, in contrast to software, are rarely subject to quality control. Initially, the authors adapted existing code smells by Fowler [1999] to spreadsheets. Starting with translating Fowlers inter class smells to inter worksheet smells [Hermans et al., 2012a], more smells followed [Hermans et al., 2012a, 2013]. Furthermore, Hermans et al. introduced an approach to detect smells creating data-flow graphs from spreadsheets. The detection techniques have been implemented in the spreadsheet analysis toolkit *Breviz* and are automatically performable. However, Hermans et al. do not propose how to deal with detected spreadsheet smells, for example, by listing refactorings that remove the quality problem.

Cunha et al. [2012] introduce a methodology to define *catalogs* of bad smells for spreadsheets. Besides the actual smell definition, the method furthermore contains steps such as validation and evaluation. In contrast to Hermans et al., Cunha et al. propose to group smells in different categories inspired by Mäntylä and Lassenius [2006]. Cunha et al. define *statistical smells* such as standard deviations between related spreadsheet cells, *type smells* such as cells that do not follow a certain pattern, *content smells* such as typographical errors, and *functional dependencies based smells*, which can be detected using data mining techniques.

Smells in Software Models

Deissenboeck et al. [2008] adapted existing clone detection techniques to identify cloned parts (a common smell in software engineering artifacts) in Matlab/Simulink models, which are used in model-based development of control systems. The presented approach extracts the underlying graph structure of Matlab/Simulink models and applied clone detection techniques on them. In a study, the authors analyzed models having 20,000 elements in total. The results of the study showed that over a third of the models were affected by cloning.

3.4.4 Discussion and Relation to this Thesis

Smells are a widely applied technique to identify potential quality defects in a variety of artifact types. They point at concrete locations in artifacts where quality defects are manifested. Smells and refactoring are often applied together: Whereas smells are applied to uncover quality defects, refactorings are used to remove those quality defects. In many cases, a concrete set of refactorings exists to remove certain quality defects identified by smells. However, since most smell detection techniques are inherently imprecise, their results are just indicators for potential quality defects. Therefore, smells are not sufficient to trigger automated smell removal by performing refactoring automated but are just starting points for manual assessment and removal.

In the following chapters, we make use of smells and refactorings in two different ways: In Section 6, we propose a technique to support test engineers in performing refactorings to remove clones in automated test cases. Concretely, we are using grammar inference techniques to propose ways to design reuse components to extract overlapping clones. In Section 7, we are defining smells for manual test cases written in natural language for the first time. Our smells are sufficient to identify quality defects in test cases written in natural language.

Chapter 4

Clones in Manual System Tests

To reduce system testing life-cycle effort effectively by exploiting commonality in test procedures, we need a better understanding of the phenomenon of commonality in system tests. We need to know to what extent commonality exists in system tests as well as how it affects system testing life-cycle activities. Commonality in test procedures is reflected in the corresponding test artifacts. A common symptom of commonality to test artifacts is cloning: Parts of test scripts or test descriptions that are syntactically identical.

The study presented in this chapter investigates the phenomenon of cloning in system test artifacts. We report on an empirical study in which we analyze clones in seven industrial system test suites containing more than 4000 manual tests. Parts of the content of this chapter have been published in Hauptmann et al. [2012a,b].

4.1 Research Goal

We define the goal of our study using the goal definition template of Wohlin et al. [2000]:

We analyze	*cloning in manual system tests*
for the purpose of	*characterization and understanding*
with respect to	*impact on system test life-cycle activities*
from the viewpoint of	*test engineers*
in the context of	*industrial software projects.*

4.2 Research Questions

The study is structured by five research questions: RQ 1 to RQ 3 investigate the extent and nature of cloning in manual system tests. Answering these questions enables to better understand the phenomenon of cloning in manual test cases. RQ 4 and RQ 5 investigate if clone detection can be used to support test life-cycle activities. We focus on the activities of execution and maintenance by using the examples of test suite optimization and test automation.

RQ 1: To What Extent Exists Cloning in Manual System Tests?

The goal of this question is to assess if cloning is actually a significant phenomenon in the current practice of manual system testing. The more cloning exists, the greater is the maintenance effort of a test suite. However, the quantity of cloning is not enough to determine the nature and severity of the problems caused by cloning. Hence, RQ 2 and RQ 3 investigate the characteristics of clones in tests.

RQ 2: What Kind of Information is Cloned in System Tests?

Tests contain several types of information, e.g., instructions for the tester or test data. Cloning different types of information might lead to different challenges. To better understand test cloning and estimate its problems, we investigate what type of information is actually cloned.

RQ 3: Are Clones in System Tests Locally Restricted?

Use cases are usually tested by several tests, for example, under different conditions, using different parameters or checking different exceptional cases. Tests of the same use case are usually more alike and share common parts. Test management tools allow structuring test suites hierarchically storing tests in folders and subfolders. If tests which exercise the same use case are stored closely together (e.g. in the same folder), it is more likely that maintenance tasks will be performed consistently in all cloned parts since it is obvious what other tests to change too. The aim of this research question is to find out if tests share common parts only with closely related tests or also with tests testing different use cases.

RQ 4: Do Clones Indicate Candidates for Test Suite Optimization?

If several tests contain large common parts, it might be beneficial to combine those test cases to a single test that accomplishes the goals of all combined tests. Clones in test cases can indicate candidates for such a test suite optimization. Combining test cases reduces the overall size of the test suite and thereby reduces the amount of tests that have to be maintained. Furthermore, merging tests together to fewer shorter tests will also make test execution more efficient.

However, test suite optimization is a complex topic that has to be performed carefully considering domain knowledge. Therefore, this research question just estimates the potential of test suite optimization. The results of this research question can be seen as an upper bound of test suite optimization by combining tests having similar beginnings.

RQ 5: Do Clones Indicate in Which Order Tests are Automated Most Efficiently?

The success of test automation depends on the effort it takes to create test scripts. This effort can be reduced by reusing test scripts for identical test steps. In this RQ, we quantify the benefit of test script reuse by determining how often identical test steps appear: The *execution frequency* of (identical) test steps. Based on these execution frequencies, we extrapolate the growth of overall test suite automation by creating test scripts for steps with high execution frequency first. This research question is based on the idea of *semi-automatic test execution*, in which automated test execution and manual test execution are combined.

4.3 Study Design

Whereas RQ 1 to RQ 3 can be answered directly using the results of the clone detections, we will not provide quantitative answers to RQ 4 and RQ 5. Instead, we use our findings to enable a discussion about how manual test suites can be optimized and efficiently automated based on clone analysis results.

4.3.1 Study Objects and Case Study Context

Our study objects are manual system tests of seven projects from Munich Re (see Section 1.5). The systems of the analyzed tests provide substantially different functionality, ranging from damage prediction, over pharmaceutical risk management to credit and company structure administration. They support between 10 and 150 users each.

We chose tests written in different languages (German and English), by different teams (internal employees and external suppliers), and testing different functionalities to increase the generalizability of the study results. We included systems with web front-end, Windows fat client interface, and SAP systems. The analyzed tests included regression tests, tests for planned features, and tests of change requests. All systems as well as the corresponding test suites are in productive use. For non-disclosure reasons we named the systems, from which we took the tests, system A to G (respectively test suite A to G). Table 4.1 summarizes the sizes of the test suites. The number of tests differs from less than 100 (test suite C) to more than 1,800 (test suite E) per test suite. In total, we analyzed 741,138 lines of natural language test description containing more than 1.4 million words.

Table 4.1: Study objects.

		Length of all Tests	
Test Suite	#Tests	(#lines)	(#words)
A	266	37,027	79,114
B	1,059	165,547	346,135
C	72	12,918	27,450
D	180	67,598	102,991
E	1,804	307,760	529,122
F	135	22,903	34,136
G	605	127,385	317,205
total	4,121	741,138	1,436,153

4.3.2 Data Collection

In this section we describe how we collected the data in order to answer the research questions. Most data was collected by automated analyses on the manual system tests. We performed the analyses using the open source quality assessment toolkit ConQAT[1] [Juergens et al., 2009], which includes, among others, a rich toolkit for clone detection.

[1] `http://www.conqat.org`

RQ 1: To What Extent Exists Cloning in Manual System Tests?

To answer RQ 1, we performed clone detections on all our study objects. In the beginning, we transformed the tests into plain text, which is easier to analyze by our tooling. Using whitespaces and line breaks, we tokenized the tests into sequences of words. ConQAT has been used to find clones of at least 30 sequential words (in natural language) in these token streams. To find clones which differ slightly (gapped clone detection. See Chapter 2) to cover test data or ignore inconsistent typos. The overall volume of the gaps were allowed to be up to 10% of the length of a clone.

We performed additional manual inspections of the clone detection results to find and exclude false positives such as different versions of the same test or stereotypical parts of tests (e.g., description templates). Those inspections were performed based on visualizations of the structure of the clones, manual inspections of clones, and interviews with project members. False positives were then excluded by configuring filters in ConQAT.

Apart from qualitative results about the structure and distribution of clones in the test suites, the main metric we gained here was the *clone coverage*, which is the percentage of the test suite that is subject to cloning. Furthermore, we calculated the *relative blow-up* of the tests introduced by cloning as well as the *length of clones* and the *cardinality of clone classes* (see Section 2.4 for details on the metrics).

RQ 2: What Kind of Information is Cloned in System Tests?

To answer RQ 2, we manually classified the content of the detected test clones. We randomly selected samples of 10 complete tests and 20 clones of each test suite. We developed a simple categorization scheme which is based on the IEEE-829-2008 standard [IEEE, 2008] for system test documentation. It describes, among others, the type of information that should be included in system tests. We designed the scheme to be detailed enough to capture the most important types of information as well as simple enough that it can be applied to a large sample of text in acceptable time. The categories of our scheme are:

Actions: Description how the tester has to stimulate the system. Actions are usually located in the *step description* parts of tests. The granularity ranges from abstract (e.g., *Execute function XY*) to concrete (e.g., *Click on the button ABC*).

Checks: Expectation of the system's behavior which has to be verified by the tester. Checks are usually located in the *expected result* part of the tests. The granularity ranges from abstract (e.g., *The result is positive*) to exact values (e.g., *Expect message 'Everything has been performed successfully.'*).

Input Data: Input required to execute the test. The granularity ranges from abstract (e.g., *a date max. three month ago*) to exact values (e.g., *'03/23/1983'*).

Output Data: Output expected by the system. The granularity ranges from abstract (e.g., *between 1 and 2*) to exact values (e.g., *'1.4'*).

References to Other Tests: References within the same test or between different tests such as prerequisite (e.g., *Test XY has to be executed before*) or jumps to other test steps (e.g., *execute step 3-5 from test XY*).

Environmental Needs: Test environment needed for the execution such as hardware, software, or test data.

The aim of the inspections was to compare the categorized information from both groups of samples. Using the scheme, we went through the samples (tests and clones) and counted what types of information has been covered. To compare both values, we normalized the assignments to a per word base. We calculate the *absolute appearance* of each category for both sample groups as well as the *relative difference* between both sample groups per category.

RQ 3: Are Clones in System Tests Locally Restricted?

To figure out if cloning only exists between tests of the same use case or also appear between tests of different functionalities, we analyzed the relationships of clones found in RQ 1.

The tests of our study objects are organized in a hierarchical structure similar to the structure of the functionality of the system under test. Starting by main function groups, the tests are further grouped by their sub-functionalities. The lowest hierarchy level always represents a specific use case or scenario. Therefore, tests which are stored in the same directory always test the same functionality whereas tests in different directories test different functionalities. We categorize clone dependencies into two categories:

Intra Use Case Dependency: If tests that are testing the same functionality are in a clone relationship, we classify this relationship as an *intra use case dependency.*

Inter Use Case Dependency: On the contrary, *inter use case dependencies* are clone relationships between tests which are testing distinct functionality.

To investigate if clones between tests are locally restricted, we calculate the *absolute* and the *relative number* of *intra* as well as *inter use case dependencies* for each test suite.

RQ 4: Do Clones Indicate Candidates for Test Suite Optimization?

This research question investigates whether clones indicate tests which can be combined. During manual inspections of the tests, we noticed that cloned parts are often located at the beginning of tests and cover actions to bring the system into a certain state, for example, by creating a defined set of test data.

In order to obtain an estimation of the minimization potential of clone information, we adapted our clone detection tooling to find just clones at the beginning of tests having a significant length (at least 50 words). We inspected such clones, as well as corresponding tests, to assess if the tests could be merged.

To quantify how many candidates for test suite optimization can be found using clone detection, we calculate the metrics *#clone groups* and *clone group size (max. and avg.)*. The metric *clone length (max. and avg.)* furthermore shows the size of text parts which are affected. We furthermore calculated the potential *saving* in artifact size that could be achieved by combining all found clones[2].

[2]In source code clone detection, a similar metric is used: *RFSS - redundancy free source statements.*

RQ 5: Do Clones Indicate in Which Order Tests are Automated Most Efficiently?

To answer RQ 5, we performed an adapted clone detection to determine the execution frequency of *identical test steps*. We modified our clone detection tooling to group test steps which are completely cloned and thereby identical. The number of clones of each group tells how often the same test script can be reused for all members of the corresponding group.

However, in manual inspections, it turned out that not all clones span over complete test steps. Some clones affect just parts of a test steps which is usually the case if test steps are created by merging or cutting test steps. As a consequence, these test steps are not suitable to directly create test scripts for. To solve this problem, we adapted our clone detection tooling again to find clones of either whole test steps or at least significantly large parts of test steps. We use the results of the clone detection to suggest how to break up test steps into parts which are more suitable for automation.

To quantify this research question, we compute two metrics: First, based on the results of the clone detection, we compute *test step execution frequencies* for each group of identical test steps. This tells how often identical test steps are executed by running the whole test suite once. Second, based on the execution frequencies, we create an *extrapolation of automation efficiency*: We sort all groups of identical test steps based on their execution frequency. Our extrapolation shows to what degree the complete test suite will be automated by creating test scripts one by one sorted by execution frequency.

This extrapolation indicates the efficiency of a stepwise implementation of semi-automatic test execution. It does not only specify the order of steps to automate, but also indicates the 'sweet spot' of semi-automatic test execution at which the proportion between the number of automated test scripts and the degree of test automation is optimal.

4.3.3 Study Execution

The case study was performed on PCs with Windows and Linux operating systems equipped with Intel Core 2 Duo CPUs with 2.4 GHz, and 4 GB of RAM each. The time for the clone detection was between a few seconds up to one hour per test suite.

4.4 Results

This section presents the results based on the research questions.

RQ 1: To What Extent Exists Cloning in Manual System Tests?

The first research question investigates the extent of cloning. The quantitative results are summarized in Table 4.2.

Clone Coverage: The clone coverage varies from 43.3% (test suite C) to 85.9% (test suite G). 6 out of 7 test suites have a clone coverage over 50% (every test suite except C).

Relative Blow-Up: The relative blow-up varies from 50% (test suite C) to 258% (test suite G). 4 out of 7 test suites have a relative blow-up over 100% which means that the test artifacts have more than double the size they could have. The tests containing the most clones are more than two and a half times longer as they could be (test suite G with a relative blow-up of 258%).

Table 4.2: Clone detection results. (RQ 1)

Test Suite	#Clone Groups	#Clones	#Cloned Words	Clone Coverage	Blow-Up
A	674	2,113	51,749	65.4%	114%
B	2,513	7,774	201,575	58.2%	75%
C	175	563	11,890	43.3%	50%
D	804	2,797	57,362	55.7%	78%
E	4,424	24,816	381,120	72.0%	138%
F	309	1,028	20,146	59.0%	82%
G	1,754	11,594	272,628	85.9%	258%

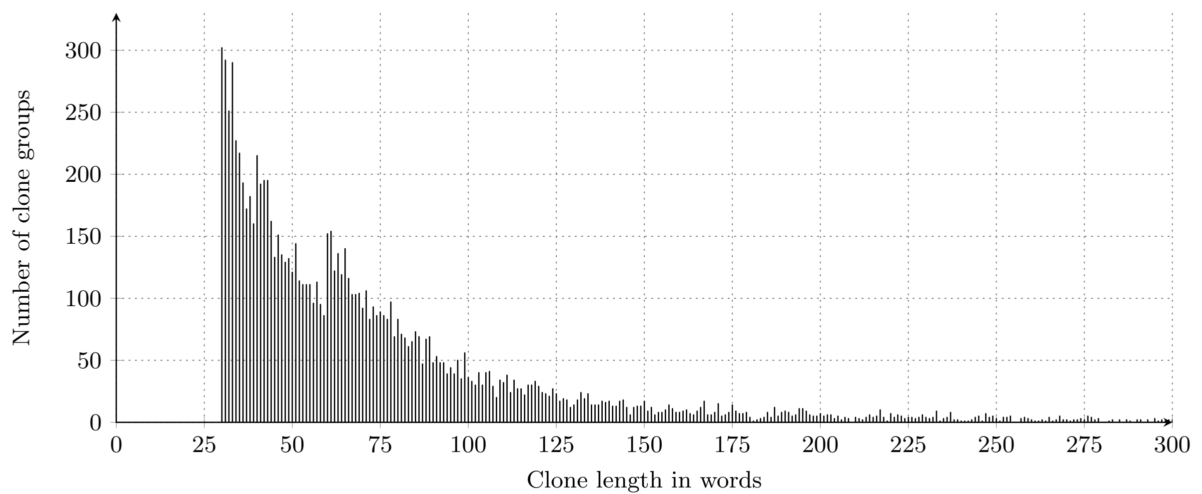


Figure 4.1: Distribution of clone lengths. (RQ 1)

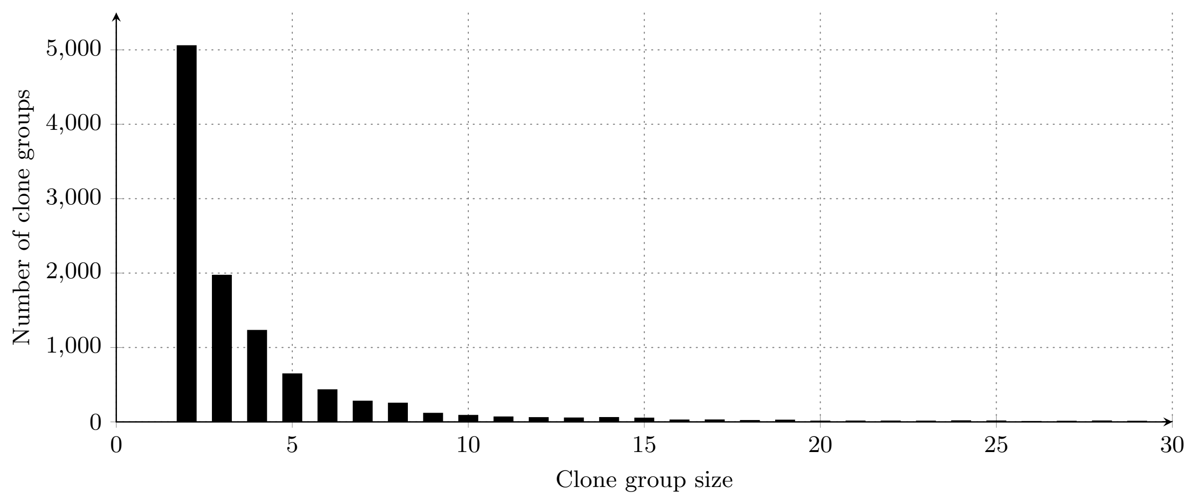


Figure 4.2: Distribution of clone group sizes. (RQ 1)

Length of Clones: Figure 4.1 shows the distribution of lengths (in words) of all clones found. Short clones are more frequent than long clones. However, 503 clone groups have a length greater than 250 words. The longest clone group has a size of 1019 words spanning 8 test steps.

Cardinality of Clone Classes: Figure 4.2 shows the distribution of the number of clones per clone group. Small clone groups with two clones are more frequent than clone groups with the size of 3 or higher. However, 104 clone groups containing more than 30 clones were detected.

RQ 2: What Kind of Information is Cloned in System Tests?

This question investigates which type of information is affected by cloning. We subjected the sample of 70 test cases (24,880 words) and 140 clones (6,998 words) to our categorization scheme. Since the size of both samples differed, we report on the relative frequencies. Table 4.3 shows the assignment rates per 10,000 words.

Absolute Appearance: *Checks* occurred the most often (446.9 for all tests and 694.5 for the clones) followed by *actions* (445/557). *Inputs* and *outputs* have been rated as 176.8/137.2 and 121.8/101.5. The categories with the least number of cases are *environmental needs* (32.2/50) and *references* (28.5/5.7).

Relative Difference: The highest relative difference between the two samples is in the category *references* with 80% less cases in the clone sample as in the overall sample. The numbers for *checks* and *environmental needs* are slightly more than 50% higher for clones as for the overall sample (+55.4% and +55.3%) whereas *actions* have 25% more cases in the clone sample. *Input* and *output values* have been assigned 22.4% and 16.7% less to cloned as to the overall sample.

RQ 3: Are Clones in System Tests Locally Restricted?

This question investigates if test clones are restricted to tests of the same functionality. Table 4.4 shows the quantitative results for all test suites.

The relative amount of inter use case dependencies varies between 4.26% (test suite E) and 90.93% (test suite C). 2 out of 7 test suites have inter use case dependencies around 5% whereas 4 out of the remaining test suites are between 80% and 90% in the same category. Just one test suite had almost as much intra as inter use case dependencies (49.17% to 50.83%). The average of inter use case clones (based on the sum of the absolute values) is 87.75%.

Additionally to Table 4.4, we created a visual representation of the structure of the test suites and their clone dependencies as *edge bundle view* [Hauptmann et al., 2012a; Holten, 2006]. Figure 4.3 exemplarily visualizes the dependencies of test suite A. The rings on the outside reflect the hierarchical structure of a test suite. The innermost level of rings represents the directories in which the tests are stored whereas every pin on the inside of the ring represents a single test. Dependencies between tests are indicated by lines.

Table 4.3: Classification of information (per 10,000 words). (RQ 2)

	Absolute Appearance		Relative Difference
Category	Overall	Clones	
Actions	445.7	557.3	+25%
Checks	446.9	694.5	+55.4%
Input Values	176.8	137.2	-22.4%
Output Values	121.8	101.5	-16.7%
Environmental Needs	32.2	50	+55.3%
References	28.5	5.7	-80%

Table 4.4: Clone dependencies between tests. (RQ 3)

	Intra Use Case		Inter Use Case	
Test Suite	(absolute)	(relative)	(absolute)	(relative)
A	290	94.77%	16	5.23%
B	1,300	19.97%	904	80.03%
C	90	95.74%	4	4.26%
D	792	19.97%	3,173	80.03%
E	4,237	9.07%	42,475	90.93%
F	118	49.17%	122	50.83%
G	248	10.76%	2,056	89.24%
total	6,805	12.25%	48,750	87.75%

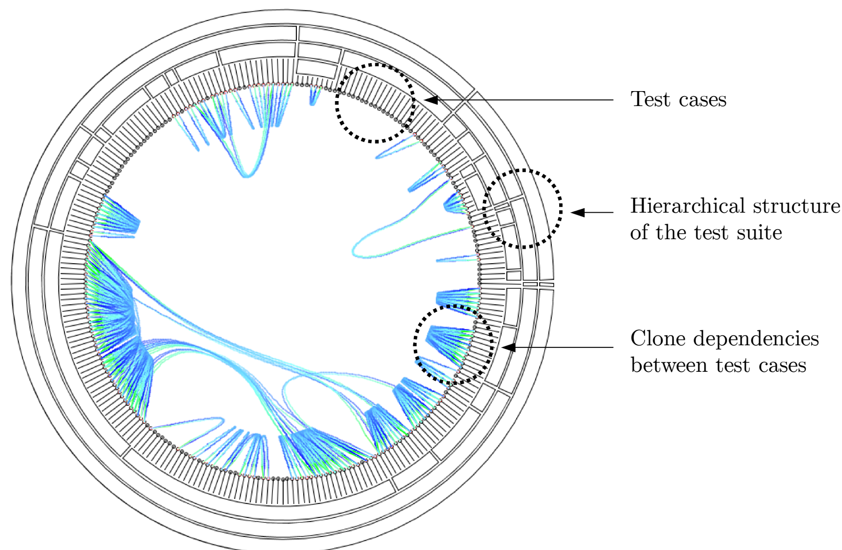


Figure 4.3: Clone dependencies between tests (test suite A). (RQ 3)

RQ 4: Do Clones Indicate Candidates for Test Suite Optimization?

This questions investigate if clone detection supports test suite optimization. Table 4.5 shows the results of the adapted clone detection we performed to find clones as indicator for test consolidation.

Table 4.5: Clone detection results for test suite optimization. (RQ 4)

	#Clone	Clone Group Size		Clone Length		Savings
Test Suite	Groups	(max)	(avg)	(max)	(avg)	(relative)
A	25	4	2.24	224	93	2.88%
B	1154	14	2.98	2943	117.05	24.63%
C	3	3	2.33	146	45	0.99%
D	10	13	4.3	127	44.7	1.19%
E	377	49	4.56	1391	73.35	10.78%
F	4	2	2	86	36.886	0.82%
G	191	13	2.96	6303	218.6	13.29%

#Clone groups and clone group size: The number of clone groups varies from 3 (test suite C) to 1154 (test suite B). The maximum cardinality of the clone groups varies between 2 and 49 clones (test suite F and E) whereas the average cardinality is between 2.24 and 4.56 (test suite A, E).

Clone length and relative savings: The possible relative artifact size reduction indicated by these results varies between 0.82% (test suite F) and 24.63% (test suite B). The relative savings of 3 test suites are around 10% and 25% whereas the remaining 4 test suites' relative savings were around 3% and lower.

RQ 5: Do Clones Indicate in Which Order Tests are Automated Most Efficiently?

This question investigates if clone detection supports efficient test automation. To answer this question we performed a tailored clone detection that finds clones of whole test steps or significant parts of test steps. The size of the clone groups indicate how often the according test step would be performed by executing the tests.

Test Step Execution Frequency: Figure 4.4 exemplarily shows the potential test steps of test suite A sorted based on their execution frequency. We found 2761 test steps of which 39.3% are occurring twice or more often. The most frequent test step occurs 52 times.

Extrapolation of Automation Efficiency: Figure 4.5 exemplarily shows the extrapolation for test suite A. The x-axis lists the test steps sorted based on their execution frequency (similar to Figure 4.4). The y-axis shows the share of overall test suite automation. In this example, by automating the 5.3% of the most frequently executed test steps, 25% of all test executions could be performed automatically. 50% of the test executions could be performed automatically after automating the top 18.3% of all

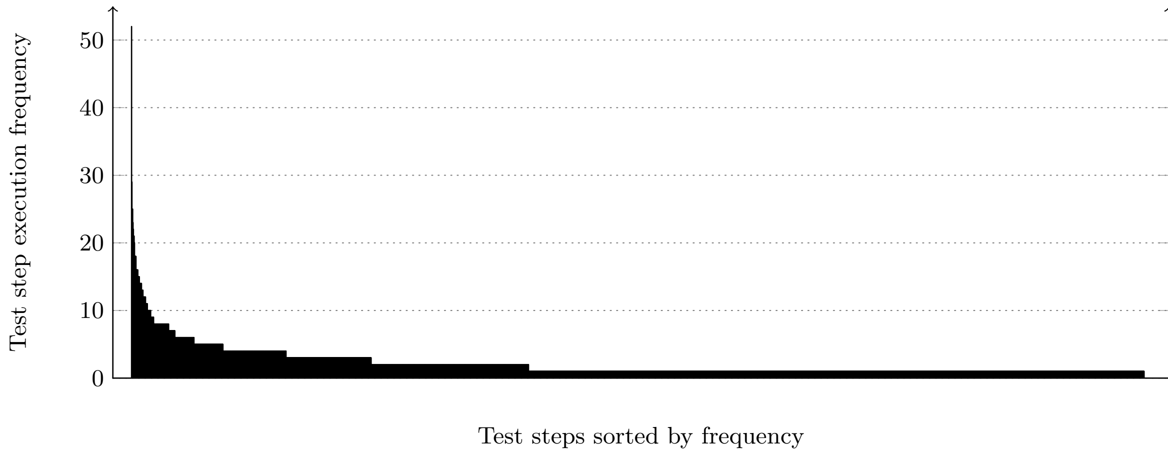


Figure 4.4: Distribution of the test step frequency for test suite A. (RQ 5)

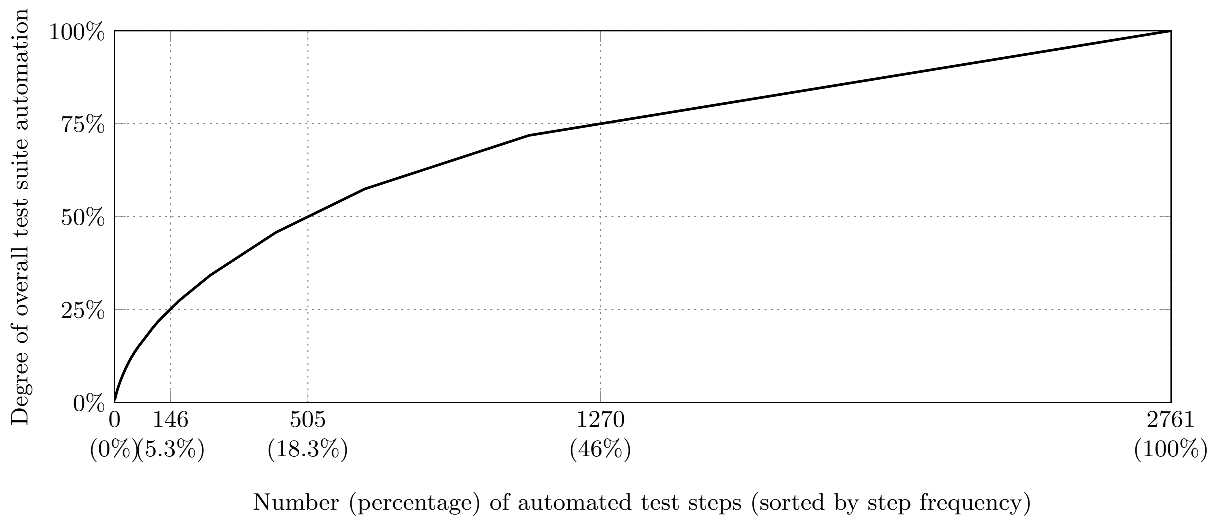


Figure 4.5: Extrapolation of the automation efficiency for test suite A. (RQ 5)

Table 4.6: Extrapolation of automation efficiency. (RQ 5)

	Percentiles			Start of lin. growth
Test Suite	(25%)	(50%)	(75%)	(at steps automated)
A	5.3%	18.3%	46%	71.9% (39.3%)
B	3%	13.5%	39.7%	78.5% (43.3%)
C	4.3%	15%	45.2%	69.8% (33.8%)
D	1.4%	9.7%	42.6%	68.2% (72%)
E	0.6%	5.9%	26%	48.5% (43%)
F	4.1%	15.4%	44.5%	71.5% (36.6%)
G	2.5%	10.8%	31.3.%	87.9% (53.4%)

steps. To automatically perform 75% of all necessary step executions, 46% of all test steps have to be automated. Since 60.7% of all test steps are unique and therefore have to be executed just once, the growth of automatically executed test steps starts to grow linearly after 39.3% of all test steps have been automated.

Table 4.6 summarizes the same metrics for all study objects. The columns *Percentiles (25%)*, *Percentiles (50%)* and *Percentiles (75%)* show the relative amount of test step executions automated by automating 25%, 50% or 75% of the test steps. The column *Start of linear growth* shows at which relative test step automation degree *(at steps automated)* the additional value of test step automation will start to grow linearly.

The 25% percentile differs from 0.6% to 5.3% of the test step automations. The 50% percentile (the median) ranges from 5.9% to 18.3%. The 75% percentile of test step execution automation is reached between 26% and 46% (test suite E and A each time). The start of linear growth ranges from 48.5% automated test step executions (at 43% steps automated of test suite E) to 87.9% (at 53.4% steps automated of test suite G).

4.5 Interpretation and Discussion

This section discusses the results based on the research questions.

RQ 1: To What Extent Exists Cloning in Manual System Tests?

The results show that cloning is a common phenomenon in real-world system tests. Although the amount of cloning differs, clones appeared in all analyzed test suites in a significant amount (clone coverage >40%). Furthermore, we found many clone groups with a size greater 5 (see Figure 4.2). As a consequence, when maintaining one of those cloned parts, at least four other parts need to be changed, too. Finding out where to perform changes can be challenging if clones are spread across multiple tests (see RQ 3). Therefore, depending on the locality of the cloned regions, clones can be a threat for test comprehension.

In the interviews we conducted to adjust the clone detection results, we noticed that the tools with which the tests have been created and maintained have a substantial influence on the amount of cloning. In several test suites, the tools did not provide the necessary abstraction capabilities to avoid cloning or the test creators were not trained to use these mechanisms properly.

RQ 2: What Kind of Information is Cloned in System Tests?

Action and check commands appear significantly more often in clones than in tests in general. In contrast, inputs and outputs appear less often in clones. The reason for this is that test data is frequently covered by the gaps of clones. Hence, clones of one clone group often differs in the input and output data whereas the sequences of commands is mostly alike. We draw the following two conclusions from that: First, it shows the need for suitable reuse mechanisms for test sequences. However, those reuse mechanism have to provide proper parameterization facilities to handle differences in test data. Second, test data is not explicitly managed but embedded in test procedures. This indicates potential maintenance and comprehensibility problems since it is difficult to assess what a test is actually testing.

Our results show further differences in the cloning of different types of information. For example, the category checks appear noticeably more often in clones than the category actions. However, we could not find an explanation for that.

Furthermore, environmental needs appear frequently in clones. This indicates a similar problem as with the test data. For a consistent handling of such needs across tests, this information should be managed separately from the test step description.

The fact that references are cloned rarely is not surprising, as common information has already been externalized in such cases.

RQ 3: Are Clones in System Tests Locally Restricted?

In twice as much test suites, clones are not limited to tests of the same functionality but are most often between tests which are testing different functionality. This fact complicates performing maintenance tasks in cloned parts consistently since it is not obvious to find all clones of a clone class.

In interviews with domain experts, we figured out that many inter use case dependencies in our study objects can be traced back to dependencies in the requirements of the corresponding use cases. If there was a dependency between two use cases (e.g., use case A must be executed before use case B), there were also clones between the corresponding tests. This supports our hypothesis that commonality and dependencies within requirements inherit to their corresponding tests.

Furthermore, clones that are spread across many tests often addressed the creation of test data. Many cloned parts of tests guide the tester in creating a certain type and amount of data which is used in the subsequent test steps. This implies that, if tests require the same type of test data, it is very likely that they have cloned test steps for the data creation.

The same is true for test steps dealing with the system interface. Independent of the use case under test, many clones that affected more than one use case contained system interface related information, such as navigation within the user interface. Therefore, if several tests require the usage of the same parts of the system interface, they are likely to have clones.

RQ 4: Do Clones Indicate Candidates for Test Suite Optimization?

In at least three test suites, we found significant portions (>10%) of cloning at the beginning of test cases which indicate that the test suite might be optimized by consolidating the affected test cases. However, not in every case such an optimization is possible or reasonable. There might be good reasons why specific tests are kept separate and redundancy and inefficiency are tolerated. For example, those tests might be part of different test sets which are executed at different times. On the other hand, there might be several other opportunities for test suite optimization that are not captured by our analysis (e.g. because the common part is not at the beginning).

In the remaining four test suites, the amount of cloning at the beginning of tests was rather low (<3%). The tests of those test suites were also smaller than the others. A possible explanation for the discrepancy in the results is that those tests are more diverse and do not include as many tests per use case as the other test suites. Thus, they do not have as many similar starting sequences.

Nevertheless, at least for the three test suites B, E, and G, the clone detection can be helpful in identifying many possible candidates for an optimization. The cloning information is especially helpful in the context of large test suites where information about redundancies and optimization potentials is otherwise hard to retrieve.

RQ 5: Do Clones Indicate in Which Order Tests are Automated Most Efficiently?

The execution frequency of test steps varies considerably among different steps. By executing test suite A, some steps are executed just once, others over 50 times.

Assuming that all tests are executed equally often, the overall degree of test suite automation can be quickly increased by automating frequently executed steps before infrequently executed steps. Although the benefit is different across the test suites, to reach 50% of automation no more than 18.3% of the steps need to be automated (in test suite E only 5.9%). To reach 75% test automation of test suite E, just 26% of the steps need to be automated. However, to benefit from such an approach, a testing framework which allows implementing semi-automated tests is required.

Since our study objects are written in natural language, not all semantically identical test steps might be found using syntactic clone detection. Thus, the values have to be regarded as a lower bound.

4.6 Threats to Validity

In this section, we discuss threats to the internal and external validity of the study and describe how we mitigated them.

Internal Validity:

The results of the clone detection might be biased by individual experiences and preferences of the researcher that tailored the clone detection. We mitigated this risk by inspecting random samples of some results by at least two researchers.

The categorization in RQ 2 has not been performed on all tests but just on samples of the tests and clones. Selecting samples can potentially introduce inaccuracy. However, we addressed this issue by selecting the samples randomly.

Since the categorization in RQ 2 has been performed manually, it is subjective to some degree. We addressed this risk by performing the categorization by three researchers which worked in close cooperation. Borderline-cases have been discussed between all researchers to compensate deviant interpretations.

To validate the results of RQ 4 and RQ 5, random samples of the findings have been manually inspected by researchers. However, the researchers have not been particular domain experts nor have they been users of the systems under tests. To substantiate the results, the findings have to be inspected by dedicated domain experts. Therefore, we consider the results of RQ 4 and RQ 5 as basis for domain experts to target their work.

In RQ 5, we calculated the automation efficiency based on the execution frequency of test steps. However, the execution frequency is just one factor to determine the automation efficiency. There are additional factors that have to be considered, such as the execution time of a test step or its individual automation effort. Therefore, we consider the results of RQ 5 as a first step towards a comprehensive effort estimation model for test automation.

External Validity:

All tests we analyzed originated from the same company testing applications of the same application domain, namely business information systems having web or rich client interfaces. It is possible that tests look different in other application domains or other companies because they use different tools and processes to create and maintain their tests. To make our results more generalizable, the study has to be repeated using tests of different application domains created by different organizations. To make our study repeatable, we used a publically available clone detection tooling and described the configuration we used (see Section 4.3 *Study Design*) so our study can be repeated by other researchers in different companies.

4.7 Summary

This chapter presented an industrial study analyzing clones in manual system test artifacts. We analyzed more than 4000 manual system tests written in natural language. The study revealed significant amounts of clones in all seven analyzed test suites. All test suites have a clone coverage between 43.3% and 85.9%. Many clones occurred even more than 30 times within a test suite. Detailed inspections of the cloned parts revealed that cloning does not affect all type of content the same way. Clones often cover just test procedures but differ in test data which is embedded in test procedures. Furthermore, cloning does not only occur between tests of the same functionality. Tests significantly share parts with other tests testing different functionality. 5 out of 7 test suites have far more (>50%) clone dependencies to tests of other use cases as to tests of the same use cases. In manual inspections, those dependencies could often be traced back to dependencies between the corresponding requirements. Using clone detection to support test suite optimization does not work for all test suites. In 3 out of 7 test suites, candidates for test consolidation could be found. However, in those cases, the results are promising. Our data indicate a potential reduction of the overall test suite between 10.78% and 24.63%. Finally, clone detection can provide helpful information to efficiently mix manual and automated test execution techniques. Using information about clone cardinalities, 50% of all test suites could be executed automatically by automation just the most frequent 5.9% to 18.3% test steps.

4.8 Conclusions to System Testing Life-Cycle Activities

In this study, cloning in manual system tests has been investigated as on example of commonality in system test procedures. The results of this study show the following consequences to system testing life-cycle activities:

Commonality in Test Procedures Can Be Used to Optimize Test Execution

Clones in test artifacts directly result in duplicated execution effort. To execute tests, two execution techniques exist: manual and automated execution, both showing their advantages and disadvantages in different settings. Duplicated execution effort strongly influences if execution techniques will play out their strengths. If not, execution techniques will not pay off and will lead to unnecessary implementation (creation) and execution expenses.

This fact leads to the consequence that information about commonality in tests can be used to optimize the selection of test execution techniques. Commonality in test procedures can be used to choose execution modes which reduce implementation and execution efforts.

Commonality in Test Procedures Hampers Maintenance of Test Artifacts

All analyzed test suites were strongly affected by cloning. Many tests contained considerable amounts of text passages which exist multiple times within a test suite. Cloning in test documents is not lead to problems per se. For a test case's understandability, for example, a self-contained test case without references to other documents might even be a good thing. However, when tests have to be modified for maintenance reasons, test clones lead to effort that has to be done unnecessarily multiple times. Furthermore, it increases the risk of inconsistently applied changes since clones are spread over whole test suites.

These facts lead to the conclusion that commonality in test procedures which is manifested in test artifacts in form of clones negatively influences maintenance costs.

Chapter 5

Choosing Execution Modes

The study in the previous chapter (Chapter 4 *Clones in Manual System Tests*) has shown that industrial test suites contain both parts (test cases or sections of test cases) for which manual test execution is the execution technique of choice and parts for which automated test execution will pay off easily. When setting up a system testing strategy, one must define an appropriate execution mode, which determines whether a test suite should be executed either fully manual, fully automated or partly manually and partly automated.

In every software development project, fixed testing approaches, strict quality goals and established development philosophies already narrow down the range of execution modes. However, most projects still provide a wide spectrum of reasonable execution modes from which a test engineer can choose.[1]

Based on our industrial experience, experts often choose execution modes in an ad-hoc style using rules of thumb, best practices or by relying on experience from similar projects. Although those decisions lead to tolerable results, we have perceived the following challenges in the state of the practice of finding and maintaining execution modes:

Finding Optimal Execution Modes:

Choosing execution modes in an ad-hoc style may result in solutions that are possibly not cost-effective and makes it hard to predict the effort of test-execution beforehand. Once determined, execution modes can hardly be altered as the basis of decision-making is often unclear.

Justifying Execution Modes:

Even though an economically optimal execution mode has been found, one might deviate from it for reasons that are economically not graspable, for example, because of company-wide testing strategies. In such cases, quantifying the impact of deviation from optimal execution modes is challenging.

[1]The previous study has been performed on interactive business information systems, which are built to be used by human users and therefore have dedicated graphical user interfaces. However, software systems that are having primarily technical user interfaces (e.g., programming interfaces or web services) can hardly be tested manually. Similarly, systems that have strict timing constraints (such as real-time systems) can also be tested only automatically. This chapter focusses on software systems that can (principally) be tested in both ways, manually as well as automatically.

Understanding the Impact of Factors:

The reason why decisions for and against execution modes are made ad-hoc is that there is no solid understanding of individual factors of impact. Knowledge of the impact of individual factors to the overall system test execution effort supports arguing for and against execution modes.

Predicting Changes:

Without a solid data-basis on that the determination of execution modes is grounded, predicting the effort of changes (e.g., because the functionality of a software system changed) is difficult.

Approach and Contributions of This Chapter:

In this chapter, we introduce a method to put the decision-making process on more solid ground. We propose an estimation model to predict the effort for the activities of test implementation, execution and maintenance. We use the concept of *costs* to model efforts of test activities. The model is based primarily on expert estimates and calculates the estimated overall effort for a given execution mode in form of total costs. The estimation model gives additional input for test experts for balancing pros and cons of execution modes under consideration and thereby enables a more solid decision-making process. Our approach furthermore documents decisions during the lifetime of a test suite. Additionally, we report on an industrial case study in which we evaluated the applicability of our estimation modeling approach. Parts of the content of this chapter have been published in Hauptmann et al. [2014].

5.1 Test Meta-Model

Our approach is not restricted to certain testing approaches, languages or tools. We nevertheless have some basic requirements on testing languages. In the following, we present our test meta-model (see Figure 5.1) illustrating those requirements to testing languages [2].

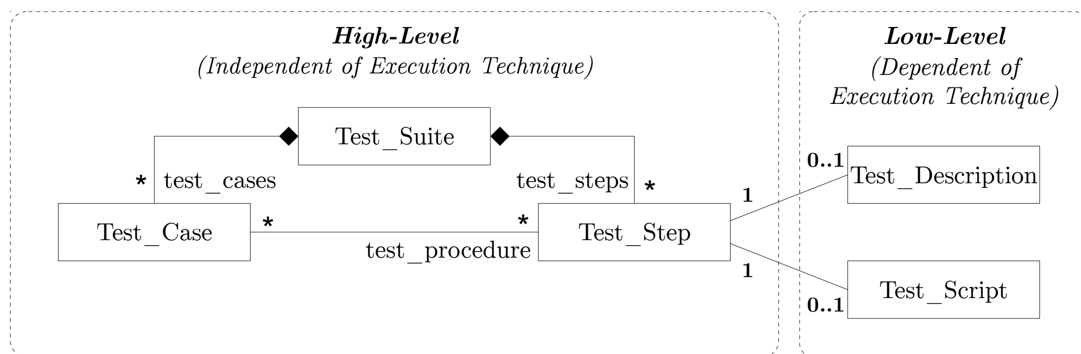


Figure 5.1: Test meta-model as UML diagram.

[2] This meta-model is a concretization of the ontology presented in Section 2.2 (Figure 2.3 on page 22) with a focus on those aspects that are important in this chapter.

Reusable Test Steps:

Similar to keywords in keyword-driven testing (KWD) [Fewster and Graham, 1999], we consider test suites to provide a set of reusable *test steps* (`Test_Step`), which act as basic building blocks to create test cases. Test steps act as placeholders for isolated activities and consist just of a unique name such as `'start the application'` or `'initialize calculation'`. Furthermore, to be integrable into several test cases, they may define parameters, for example, a test step named `'login to the system'` might have the two parameters `'username'` and `'password'`.

Test Procedures

The procedures of *test cases* (`Test_Case`) are hence created by assembling sequences of existing test steps (`test_procedure`). However, test steps are not copied but referenced at this point. This support maintainability of test steps. Furthermore, test cases might define concrete values to be passed to test steps.

Implementation and Execution:

To make test steps executable, each of them is realized using (at least) one execution technique. Therefore, each test step is linked to a *Test Description* (`Test_Description`) for manual execution and/or a *Test Script* (`Test_Script`) for automated execution.

Test steps, form the level on which execution techniques are defined. *Execution modes* can hence be realized on a per-step basis by determining which test step shall be executed manually and which automatically. Furthermore, this approach provides flexibility in implementing more complex execution modes: For example, if a test step is referenced by several test cases, which shall be executed using different execution techniques, several implementations can be provided.

5.2 Relative Cost-Benefit Analysis

We present an analytic estimation model that quantifies the impact of execution modes on testing efforts. It thereby provides a solid base for test experts to argue in favor or against execution modes. Our estimation model models effort by making use of an abstract concept of *costs*. However, costs do not necessarily have to be represented in form of a financial currency but can also be budget shares or time.

The goal of our estimation model is not to calculate the absolute effort of testing but to provide quantitative evidence to compare execution modes. Therefore, just the cost factors in which execution modes differ have to be considered. For example, efforts for deriving test conditions (see Section 2.3) do not have to be considered as they incur independently from a selected execution mode. Since we are also focussing on comparing different mixes of manual and automated test execution[3], initial costs, such as for setting up automation infrastructure or for populating databases, do not have to be considered in our calculation too. Since our estimation model only regards those factors in which execution modes differ, we consider it as a model for *relative cost-benefit analysis*.

[3]More about mixes of manual and automated test execution can be found in the case study in Section 5.4.2 *Study Object and Case Study Context*

Testing Activities:

Basis for our cost-benefit analysis are the system testing activities as defined in the *generic system test process* (Section 2.3): Test implementation, execution and maintenance.

Test Implementation

The activity of test implementation includes all tasks necessary to transfer test steps into executable low-level test steps, such as, test descriptions or test scripts.

Manual Test Execution: For manual test execution, this includes the effort for writing test descriptions for the manual tester, crosschecking the test descriptions with the system requirements, and testing them with the actual system under test.

Automated Test Execution: For automated testing, this includes effort for programming test scripts, quality assurance of the created source code, and trying out the test scripts with the actual system under test.

Since we just want to compare execution modes, tasks that are independent from a certain execution techniques, such as setting up testing environments do not have to be considered.

Test Execution

Executing addresses all tasks necessary to run a test suite against the system under test. This includes all test runs during a certain time period under observation.

Manual Test Execution: For manual test execution, this includes all efforts arising during the time period under observation for running test cases by a human tester as it is explained in the test description.

Automated Test Execution: In automated testing, this includes efforts for executing the test scripts during the time period under observation.

Similar to test implementation, the activities that do not differ between execution techniques can be ignored (e.g., preparing the system under test for execution).

Test Maintenance

Maintenance includes all changes of the test suite that can be predicted beforehand. This includes adaptions of the test suite because of planned changes of the system under test as well as estimated adaptions due to technical reasons. Similar to test execution, the efforts for test maintenance have to be approximated for a given time period under observation.

Manual Test Execution: For manual test execution, this includes approximations for the effort of adaptions of the test descriptions during the time period under observation.

Automated Test Execution: For automated test execution, this includes approximations for the efforts for maintaining and reprogramming test scripts during the time period under observation.

Similar to test implementation and test execution, tasks that have to be done equally for each execution techniques do not have to be considered.

5.3 Cost Estimation Model

In the following, we present our analytic cost estimation model to quantify the impact of execution modes on overall testing expenses. We align the basic structure of the cost model along the activities of the system test case life-cycle (see Section 1.1.1 *The System Test Case Life-Cycle*):

The *total costs* $C(test_suite)$ of the execution mode of a test suite $test_suite$ consist of the total costs of the activities implementation ($C_{\text{impl}}(test_suite)$), execution ($C_{\text{exec}}(test_suite)$) and maintenance ($C_{\text{maint}}(test_suite)$) of a test suite:

$$C(test_suite) = C_{\text{impl}}(test_suite) + C_{\text{exec}}(test_suite) + C_{\text{maint}}(test_suite)$$

Activity Costs – Implementation

The *total costs of implementation* $C_{\text{impl}}(test_suite)$ sums up the costs for implementing $c_{\text{impl}}(step, exectech(step))$ for each test step $step$ with an execution technique $exectech(step)$ as defined in the execution mode:

$$C_{\text{impl}}(test_suite) = \sum_{step \,\in\, test_suite.test_steps} c_{\text{impl}}(step, exectech(step))$$

Activity Costs – Execution

The *total costs of execution* $C_{\text{exec}}(test_suite)$ sums up the costs for executing each test step $step$ in the period under observation. The total costs of executing a test step consists of its costs for a single execution $c_{\text{exec}}(step, exectech(step))$ using a given execution technique $exectech(step)$ multiplied by the number of references to it within the test suite $\#refs(step)$ and the expected number of test runs $\#testruns$ of the test suite:

$$C_{\text{exec}}(test_suite) = \sum_{step \,\in\, test_suite.test_steps} \left(\begin{array}{l} c_{\text{exec}}(step, exectech(step)) \times \\ \#refs(step) \times \\ \#testruns(test_suite) \end{array} \right)$$

Activity Costs – Maintenance

The *total costs of maintenance* $C_{\text{maint}}(test_suite)$ sums up the costs for maintaining each test step *test* $c_{\text{maint}}(step, exectech(step))$ using a given execution technique $exectech(step)$ within the period of consideration:

$$C_{\text{maint}}(test_suite) = \sum_{step \,\in\, test_suite.test_steps} c_{\text{maint}}(step, exectech(step))$$

Table 5.1 summarizes all cost factors of our cost model.

Table 5.1: Basic cost factors.

Cost Factor		Depends on: Test Suite	Test Step	Execution Technique
Implementation	c_{impl}		X	X
Execution	c_{exec}		X	X
Maintenance	c_{maint}		X	X
No. of References	$\#refs$		X	
No. of Test Runs	$\#testruns$	X		

5.4 Evaluation

In the following, we report on an industrial case study in which we applied our cost estimation model to a real-world test suite having the following goals: (1) To evaluate the applicability of the estimation model to real-world test suites. (2) To quantify the consequences of execution modes on test execution efforts.

5.4.1 Research Goal

We define the goal of our study using the goal definition template of Wohlin et al. [2000]:

We analyze	*execution modes of system test suites*
for the purpose of	*characterization and quantification*
with respect to	*consequences to system testing efforts*
from the viewpoint of	*test engineers*
in the context of	*industrial software projects.*

Concretely, we want to find out how much the decision for a certain execution mode affects the overall testing efforts (with respect to the overall costs calculated by our cost estimation model). We show to what extent system testing effort can be reduced by choosing optimal execution modes.

5.4.2 Study Object and Case Study Context

Our study object is a test suite from Munich Re (see Section 1.5 *Case Study Partners*). The test suite consists of 41 manual system test cases testing a rich client-based business information system. The test cases have been created along the system's use cases. Besides some exceptions, every test case is testing one use case flow. However, for one flow, several test cases might exist. The test suite is designed following our test meta-model as we presented it in Section 5.1. All test cases are composed of 194 reusable test steps each covering parts of the business processes under test. Most test steps involve complex user interface handling covering simple user interface interactions such as clicking a single button up to performing more complex tasks involving several UI widgets and dialogs.

Originally, the test suite implements a full manual execution mode since all test steps are equipped with a test description in natural language. Goal of this case study is to find out whether it pays off to switch from the existing manual execution to a different execution mode. We took into consideration the following execution modes:

Execution Modes on the Level of Test Suites

We took into consideration two execution modes that define a single execution technique for the whole test suite:

Fully-Manual: Each test case is executed completely manual; no automated execution is involved (this reflects the currently implemented execution mode and works as a baseline for comparison).

Fully-Automated: Each test case is executed completely automatically executable. Each test step is realized by a test script. This execution mode is the opposite of fully-manual.

Execution Modes on the Level of Test Cases

We took into consideration another execution mode that defines individual execution techniques for each test case:

Manually-or-Automated: In this coarse-grained automation-decision, one must decide for each test case to execute it either completely automatically or completely manually. Since many solutions are possible, manually-or-automated actually represents a group of configurations of this execution mode.

Ignoring the two border cases *fully-manual* and *fully-automated*, in our case study, $2^{41}-2$ different configurations of this execution mode are theoretically possible (ignoring the two border cases *fully-manual* and *fully-automated*). As a simplification, we focus on the configurations that cause the minimal costs (*manually-or-automated-min*) and maximal costs (*manually-or-automated-max*).

Execution Modes on the Level of Test Steps

Finally, we took into consideration another execution mode that defines execution techniques for each test step individually:

Semi-Automated: In this fine-grained automation-decision, test cases are executed in a hybrid format since each test case may contain manual and automated parts. In this execution mode, the selection for an execution technique is made for each test step. If a test step is set to be automated, it will be executed automatically whenever it appears in a test case.

Similar to *manually-or-automated*, $2^{194} - 2$ configurations of this execution mode are theoretically possible (ignoring the two border cases *fully-manual* and *fully-automated*). However, this is just a theoretical number and not each of them may be feasible to implement (or reasonable). Furthermore, we focus on the solutions which causes minimal costs (*semi-automated-min*) and maximal costs (*semi-automated-max*).

The execution modes considered in this study represent a small selection of possible execution modes. Other conceivable execution modes might include different selections of cost factors, which have to be integrated in the calculation. For example, in our case study, we assumed that even in the execution mode *fully-automated*, tests are not executed unattended but a person must supervise the automated execution. Therefore, we include the automated execution time as labor time in the calculation. However, in other scenarios, automated test cases might be executed totally unattended without supervision.

5.4.3 Data Collection

To apply our cost estimation model to the test suite of our case study, we define a set of cost factors to capture the differences between the execution modes we want to compare. Furthermore, we define elicitation methods for each cost factor and perform the actual data collection.

Precise values for most cost factors can hardly be obtained in advance. Real values can only be obtained in retrospect. Therefore, we rely on expert estimation techniques to assess the value of those cost factors for which concrete measurement techniques cannot be applied. All estimations have been made by a trainee employed at Munich Re unless otherwise stated. The trainee has a solid background in programming and acts in this case study as test engineer, test script developer and manual tester. The trainee received training by domain experts of Munich Re to get familiar with the application domain and the system under test. Furthermore, he executed the whole test suite manually as well as exemplarily implemented test scripts for randomly selected test cases to get used to the role of a manual tester and test script developer.

In this study, we use working time to express the cost factors whenever possible. We assume that manual testers and test script developers are paid equally. If otherwise, working time can be corrected with factors representing the difference. In the following, we describe how we collected the data for each cost factor sorted by activity (Table 5.2 shows the data collection methods for all cost factors of the case study.):

Implementation Time:

The elicitation technique for the implementation time of test steps depends on the chosen execution technique. For automated execution, we estimate the time necessary to program test scripts. For manual test execution, we approximate the time for writing test descriptions:

Programming Test Scripts – $\mathbf{c_{impl}}$(step, $\langle \boldsymbol{AUTOMATED} \rangle$): The test script development strategy in our case study was a *capture-replay-plus-adaption* like style. This means, that each test step has been automated using capture-replay facilities of the test automation tool. If capture-replay was not sufficient to automate a test step, the test scripts that have been generated by the capture-replay tool have been manually modified. This was the case, for example, if test scripts had to be parameterized. Just in exceptional cases in that the capture-replay facilities of our testing tool were by far not sufficient, test scripts have been programmed manually. We estimated the automation time in a two-step process: First, we manually classified all test steps based on their

Table 5.2: Cost factors and their data collection methods within the case study.

Cost Factor	Data Collection Method ⟨*AUTOMATED*⟩	Data Collection Method ⟨*MANUAL*⟩
c_{impl}	Manual classification based on estimated test script development effort; experiments with random samples to determine average values.	Manual classification based on estimated creation time; expert estimation to determine the average development time for each class.
c_{exec}	Manual classification based on estimated execution time; experiments with random samples to determine average values.	Experiments to determine average execution time.
c_{maint}	Interviews with the test management: Same values as programming test scripts.	Interviews with the test management: Same values as creation of step descriptions.
#refs	Static analysis: Number of references to each test step.	
#testruns	Interviews with the project's test management.	

expected automation effort into the following classes: *capture-replay*; *capture-replay + minor adaptions*; *capture-replay + major adaptions*; *custom-time*. In a second step, we took several random samples from each class, created test scripts and tracked the development time. Since there were no outliers, we used the average development time of the random samples as representative values for each class.

Writing Test Descriptions – $\mathbf{c_{impl}}$(step, ⟨*MANUAL*⟩): Similar to the implementation time for automation, we manually classified each test step based on the expected effort to write test descriptions in natural language for them. We defined three classes based on how labor intensive we estimate the creation. Based on own experience in writing test descriptions, we defined representative values for each class.

Execution Time:

To estimate the execution time, we estimate the run time of test scripts and the time testers need to perform test steps manually.

Automated Execution Time – $\mathbf{c_{exec}}$(step, ⟨*AUTOMATED*⟩): To estimate the execution time for automated execution using test scripts, we classified each test step based on their test scripts' expected execution time. We created six classes based on how many UI elements and dialogs are involved in executing them. For each class, we took several random samples, exemplarily implemented test scripts, and executed them to measure their execution time. Based on the measured execution times, we took the average execution time over all samples of a class as its representative value.

Manual Execution Time – $\mathbf{c_{exec}}$(step, $\langle \boldsymbol{MANUAL} \rangle$): Compared to other cost factors, we were able to gather exact values for the manual execution time of each test step. We executed the whole test suite manually once and tracked the execution times for each step. Thereby, we received multiple measured values for each test step[4]. Subsequently, we used the average time for further calculation.

Number of Invocations per Step – #refs(step): We automatically counted how often each test step is referenced within the test suite. By this, we figured out how often each test step is executed during one test run.

Number of Planned Test Runs – #testrun: To receive the number of expected test runs during the system's life cycle, we performed interviews with the system's test management. The interviews revealed, that the test suite is planned to be executed 40 – 60 times within the next two years.

Maintenance Time:

Based on interviews with the test management, we did not find a universal way of reliably predicting how often and to what extent each test step will be affected by maintenance tasks. However, the test management agreed that each test step will be affected by maintenance tasks during the time of consideration. In agreement with the test management, we applied a simple heuristic to predict maintenance costs of test steps: We assumed that every test step is changed once within the time of consideration.

Maintaining Test Scripts – $\mathbf{c_{maint}}$(step, $\langle \boldsymbol{AUTOMATED} \rangle$): Since the test scripts are created in a capture-replay-plus-adaption like style, we assumed that maintaining them will most likely lead to creating them from scratch. Therefore, we took the script's creation time also as an approximate value for their maintenance effort[5].

Maintaining Test Descriptions – $\mathbf{c_{maint}}$(step, $\langle \boldsymbol{MANUAL} \rangle$): Similar to the maintenance costs for test scripts, we also used a step description's creation time as the value for maintenance time.

5.4.4 Study Execution

For *fully-manual* and *fully-automated*, calculating the overall costs is straightforward since just one configuration exists implementing those execution modes. However, for *manually-or-automated* and for *semi-automated*, multiple configurations exist since one must decide which parts of the test suite to execute manually and which automatically. We found configurations for each execution mode with minimal and maximal costs using the linear programming solver from GLPK[6]. We created solver models for each execution mode and thereby received the overall costs for an optimal configuration (*min*) as well as for the worst configuration (*max*). For each configuration, the solver also stated which test steps to execute automatically and which manually. However, as we were solely interested in the economic impact of each execution mode, we will only discuss the overall cost.

[4] The number of measured values per step depends on the factor $\#invoc(ts)$.

[5] If the test scripts would follow a different programming style, other estimation approaches can be applied.

[6] GNU Linear Programming Kit `http://www.gnu.org/s/glpk`

5.4.5 Results

Figure 5.2 illustrates the results of our calculations. We used the cost factor *#testruns* (total number of test suite runs) as a independent variable to get more insights on how costs of execution modes develop: We ran each calculation several times changing the value of *#testruns* from 0 to 300 test runs[7]. The x-axis of Figure 5.2 shows different values for *#testruns*. In this case study, the actual number of planned test runs is 40 – 60 (indicated by the gray area).

The y-axis of Figure 5.2 shows the total costs C for each execution mode. For *fully-manual* and *fully-automated*, the costs increase linearly based on the number of test suite runs. However, for manual-or-automated and semi-automated, several solutions exist (as explained in Section 5.4.2). For both types of execution mode, Figure 5.2 shows two values: One solution resulting in minimal and one resulting in maximal total costs C. The difference between the minimal and maximal costs (highlighted as patterned area between minimal and maximal values) indicates the continuum of possible solutions of execution modes.

[7] Although this case is unrealistic, we included it to see the fixed costs for each execution mode.

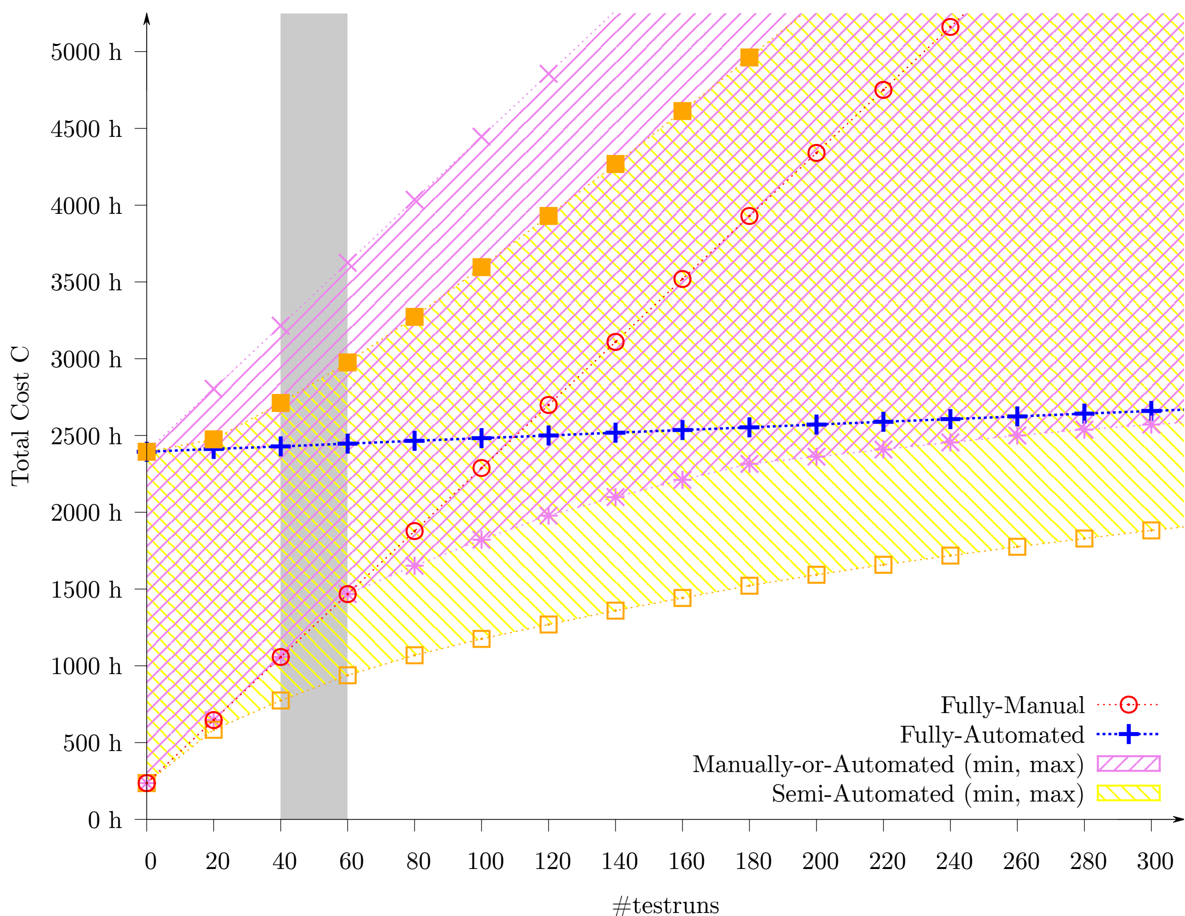


Figure 5.2: Total costs for each execution mode (based on the number of test runs).

Fully-Manual and Fully-Automated:

Although the *fully-manual* execution mode has rather low fixed costs, the total costs rapidly increase the more often the test suite is executed. In contrast, the *fully-automated* execution mode has high fixed costs, but the increase of costs is low the more often the test suite is executed[8]. Based on our calculation, the higher creation and maintenance effort of a full test automation will pay off, compared with *fully-manual*, after 110 test runs in this study.

Manually-or-Automated:

Looking at the optimal solutions for the execution mode *manually-or-automated*, separating a test suite into manual and automated test cases will not pay off until 60 test runs in comparison with full manual test execution. Starting at ∼80 test runs, there is a small range where the execution mode *manually-or-automated* causes fewer costs than *fully-manual* or *fully-automated* execution. However, if executed more often, the difference to *fully-automated* decreases quickly.

In contrast, if the separation into manual and automated test cases is done the wrong way, costs can be far above all other execution modes. Although it is unrealistic that an experienced test engineer would choose such unsuitable execution modes, it shows the range of consequences.

Semi-Automated:

Selecting the optimal allocation of test steps to execute manually and steps to execute automatically, *Semi-Automated* execution will start to pay off early (after ∼20 test suite runs). Similar to *manually-or-automated*, the costs converge to the costs for *fully-automated* if the test suite is executed more often. However, compared to *manually-or-automated*, the convergence is slower. Between ∼60 and 300 test suite runs (the maximum we calculated), the costs for *semi-automated* are ∼20% − 30% lower compared with all other execution modes.

Similar to *manually-or-automated*, if semi-automation is implemented the wrong way, the costs can be high. However, due to the experience of test engineers, we consider this upper bound as unrealistic to reach.

5.4.6 Interpretation and Discussion

With this case study, we wanted (1) to evaluate if the estimation model is applicable in a real world context and (2) to describe the impact of execution modes to the test execution expenses. Regarding (1), we found that data for the relevant factors can be collected with an acceptable effort. Where we had to estimate values, we used a combination of classification, sampling and experimentation. As we did not evaluate the accuracy of our estimation, we can only discuss (2) on a qualitative basis.

Figure 5.2 shows the estimated costs for multiple numbers of test runs. One can perceive that the costs for different execution modes differ strongly. Hence, selecting execution modes pays off: A common test execution strategy is to decide per test case whether to automate it or not (manual-or-automated). Surprisingly, only a small window (∼100 − ∼140 test runs)

[8] In this study, we included the execution time for automated test cases in the calculation. In case of a totally unattended automated execution of test cases, one could exclude this factor from the calculation. This would lead to no increase of the total costs the more often the test suite is executed.

exists where this strategy gives a considerable advantage over either fully manual or fully automated test suites. In our case study, semi-automated test execution would be more cost effective. The advantage of semi-automated test execution can be gained by deciding on automation on a per-step basis. This advantage can be reached from just a few test runs on (~20 test runs) and is maintained for all numbers of testruns (we calculated up to 300 runs).

However, in our calculation, we assumed that the number of test runs is known in advance (or at least a reasonable estimation). In software development settings with fixed testing plans, such estimations might be quite accurate. However, if system test suites are executed in an ad-hoc style without strict testing plans, getting a reliable estimation of the number of expected test runs is hardly possible.

Furthermore, all strategies that are neither fully automated nor fully manual bear the risk of automating the wrong parts, which can drive costs heavily. Hence, it is crucial to find out which parts to automate and which to execute manually. Additionally to the overall costs of a test suite, our calculation also reveals the exact configuration of the execution modes *manually-or-automated* and *semi-automated* in terms of detailed lists which test cases (or test steps) to automate and which to execute manually. To help test engineers in gaining more information of the results, our calculation can easily be extended to provide additional information about individual parts of test suites, for example, by sorting the list of test cases (test steps) based on their priority to be automated or manually executed. This information can be used by test engineers to tailor execution modes individually based on project needs.

5.5 Benefit Estimation

In the previous section, we applied our effort estimation approach to an industrial test suite and motivated the need to choose execution modes deliberately having the overall costs in mind. The results showed that the costs can differ strongly between execution modes, however, the actual influence to the overall system testing costs is unclear. In this section, we want to discuss the benefit that is gained by our approach. More specifically, we perform a rough benefit estimation to find out how much percent of system testing costs can be saved by our approach.

5.5.1 Benefit Estimation Model

In the following, we present a simple model that enables a rough benefit estimation of our approach. Goal of the model is to estimate how far the overall system testing costs can be reduced by our approach. Generally, the focus of this thesis is on the testing activities of the system test case life-cycle, namely test creation, test execution, and test maintenance (see Section 1.1.1 *The System Test Case Life-Cycle*). These activities cover just tasks that focus on test case artifacts, however, the overall process of system testing comprises more tasks than just test case creation, execution, and maintenance. Hence, we use the generic system test process presented in Section 2.3 *A Generic System Test Process* and extend our system test case life-cycle with the missing activities (see Figure 5.3).

Having this list of all activities that are involved in system testing, we assume that the overall costs of system testing calculates by summing up the costs of each individual activity:

	Test Analysis Costs
+	Test Design Costs
+	Test Creation Costs
+	Test Execution Costs
+	Test Maintenance Costs
=	Overall Costs of System Testing

Relations Between Test Activity Costs

Each of the contributions of this thesis focuses on directly reducing the costs of system test case life-cycle activities (see Section 1.4 *Contributions*). Hence, for each system test case life-cycle activity, we can give sufficient predictions of the average costs of these activities and the benefit of our contributions regarding these activities. Unfortunately, for the activities outside the system test case life-cycle (test analysis and test design), we have no empirical data to perform accurate effort informations.

To overcome this lack of information, and to make our benefit estimation model as simple and transparent as possible, we do not try to make precise cost estimations of each of those activities, nor do we calculate the actual overall costs of system testing. Instead, we estimate the relative size of each activity within our calculation (for example, *Activity A* is typically three times more expensive than *Activity B*). Having the difference in size between all activities allows making predictions without having exact values for each testing activity. Once we know in as far individual activities are affected by our approach, we can use our relative model to extrapolate the change of the overall costs. In the following, we describe how we use empirical data and estimations to determine the proportions between testing activities.

Costs for Activity *Test Analysis*: The goal of this activity is to analyze the specification of the system under test and to come up with test conditions (testable aspects of the system under test) that shall be validated by test cases. Furthermore, test slices are identified that satisfy the test conditions (see Section 2.3.1 *Test Analysis*).

Since none of our investigations or studies addressed these tasks in detail, we have no precise data on how much effort is spent on this activity in practice. To perform our calculation, we estimate the costs for test analysis by comparing it with other activities for which we have more precise data:

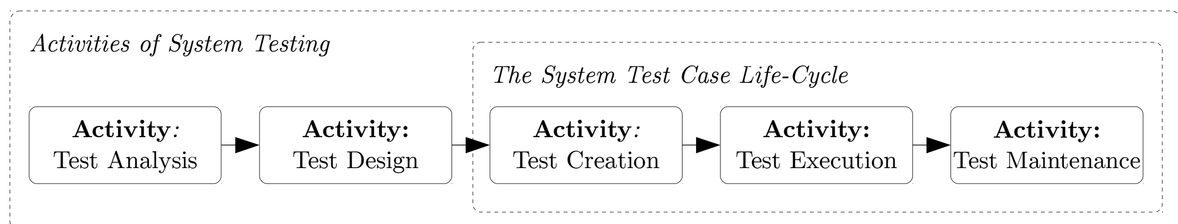


Figure 5.3: Activities within the generic system test process (see Section 2.3).

In the remainder of this thesis, we assume that the costs for test analysis are in the same order as for the implementation of test cases. We argue, that the effort of both activities scales linearly with the number of test cases that are developed. Furthermore, we expect that the effort of coming up with a test condition and identifying test slices from specifications (high-level test case) is similarly labor-intensive as implementing an executable low-level test case (activity *Test Creation*).

Costs for Activity *Test Design*: The goal of this activity is to extend each test slice to a high-level test case containing rough outlines of test procedures (see Section 2.3.2 *Test Design*).

Similarly to the test analysis, we lack detailed information on test design costs in practice. However, likewise to the activity of test creation, the effort for creating high-level test cases from test slices grow linearly with the number of test cases. Therefore, (similarly to test analysis) we assume that the costs for the activity test design is equal to the costs for test creation.

Costs for Activity *Test Creation*: The goal of this activity is to enrich high-level test cases with detailed information that is necessary to actually perform the test cases (see Section 2.3.3 *Test Implementation*).

Throughout the case study of this chapter, we gathered detailed information to estimate test creation efforts with reasonable precision. Therefore, we use the test creation costs as a reference activity to define the costs of the other activities relatively to it (*TestCreationCosts*).

Costs for Activity *Test Execution*: The goal of this activity is to actually run test cases. Therefore, manual test cases are performed by testers or, in the case of automated test execution, test scripts are started and supervised (see Section 2.3.4 *Test Execution*).

Similarly to the activity test creation, we gathered detailed information of test execution effort throughout the case study of this chapter. However, to simplify the calculation, we use empirical data from our studies to investigate average values of test execution costs. We calculate a test execution factor (*TextExecFactor*) that shows the average ratio between the costs of creating a test cases and executing it once. To estimate the overall costs for test execution, we multiply this estimated average value for one execution with the number of planned test runs (*#testruns*)

Costs for Activity *Test Maintenance*: The goal of this activity is keep test case artifacts up to date and adapt them to changes of the system's functionality or modifications of the system interface (that is used for testing) (see Section 2.3.5 *Maintaining System Test Artifacts*).

In the previous case study, interviews turned out, that there is no universal way of reliable predicting how often and to what extent each test step will be affected to changes. Therefore, we used a simple heuristic to perdict the maintenance effort of test steps: We assumed that every test step is changed once within the time of consideration (see Section 5.4.3 *Data Collection*). Building on this assumptions, we estimate that, for our benefit estimation, test maintenance is causing similarl costs as test creation. Furthermore, we multiply this costs by the number of expected maintenance tasks (*#mainttasks*).

Relative Benefit Estimation Model

Using the relations between the testing activities of our model, we can approximate our calculation of the overall system testing costs as follows:

$$
\begin{array}{llr}
 & TestCreationCosts & \text{(Test Analysis Costs)} \\
+ & TestCreationCosts & \text{(Test Design Costs)} \\
+ & TestCreationCosts & \text{(Test Creation Costs)} \\
+ & TestCreationCosts \times TextExecFactor \times \#testruns & \text{(Test Execution Costs)} \\
+ & TestCreationCosts \times \#mainttasks & \text{(Test Maintenance Costs)} \\
\hline
= & \text{Overall Costs of System Testing} &
\end{array}
$$

5.5.2 Quantitative Benefit

The contribution presented in this chapter addresses all three activities of the system test case life-cycle (namely test creation, execution, and maintenance) (see Figure 5.4). To find out in as far the overall system testing costs are reduced by applying our approach, we first calculate in as far the activities test creation, execution, and maintenance benefit from our approach. Having this information, we can use our relative benefit estimation model to calculate how much our approach reduces the overall costs of system testing.

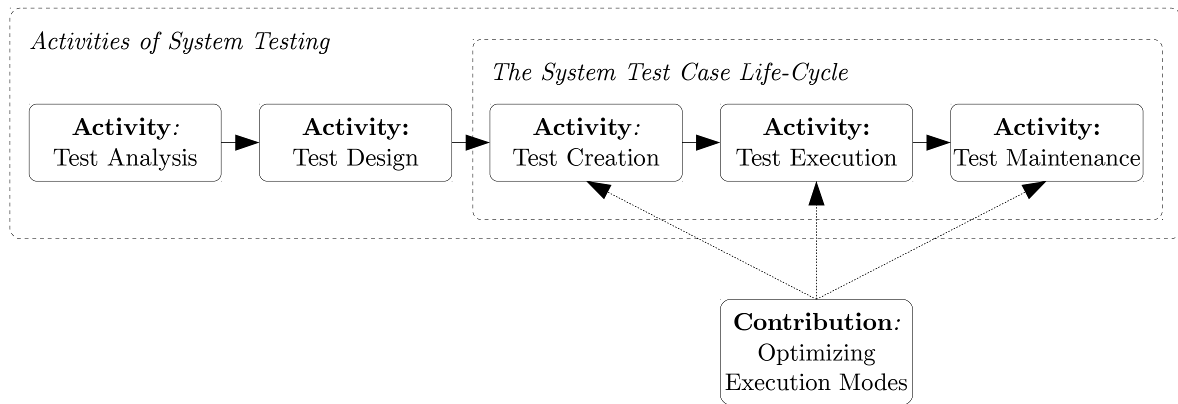


Figure 5.4: System testing activities that are addressed by our contribution.

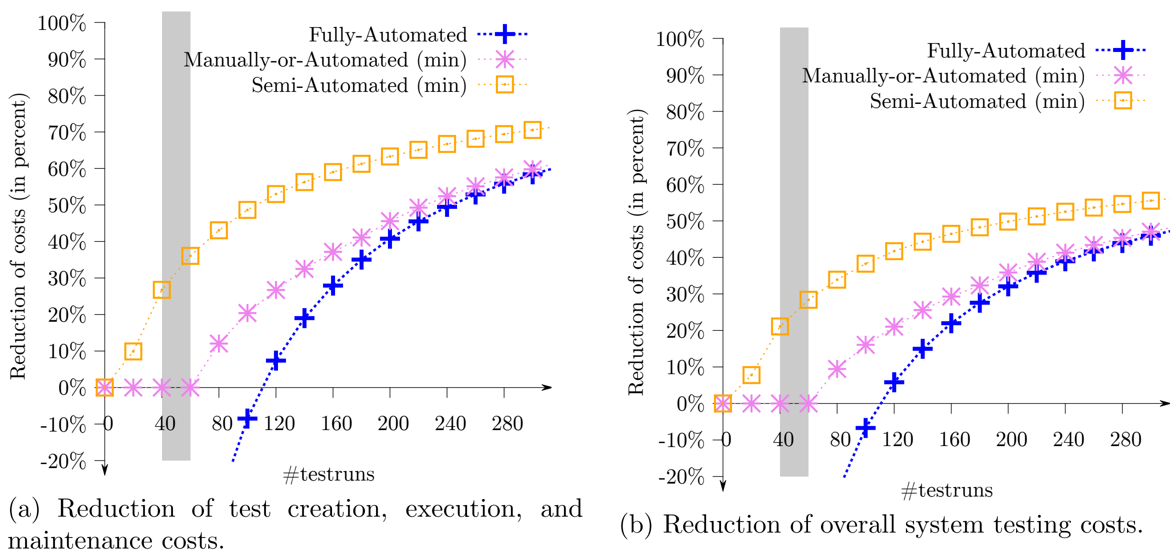
Reduction of Test Creation, Execution, and Maintenance Costs

To estimate the benefit for the system test case life-cycle activities that is gained by applying our approach, we compare the total costs for an optimal execution mode (as it has been proposed by our approach) with the total costs of the original execution mode (*fully-manual* or *fully-automated* in our case study). Furthermore, to consider the initial investment of our approach, we add additional costs for performing our approach to the costs of an optimal execution mode:

$$
\begin{array}{rl}
 & \text{Costs of optimal execution mode (system test case life-cycle activities)} \\
+ & \text{Costs of performing cost estimation analysis} \\
\hline
= & \text{Total costs of our approach (system test case life-cycle activities)} \\
 & \qquad\qquad\qquad\qquad\qquad \textit{vs.} \\
 & \text{Costs of original execution mode (system test case life-cycle activities)}
\end{array}
$$

Data Elicitation: We apply our estimation model based on the calculations of the previous case study. More specifically, we use the costs for optimal and for the original execution modes as we calculated them in Section 5.4 *Evaluation*. The effort of performing our cost estimation analysis splits up in two parts: Collecting all relevant data (manual classification, expert estimations, static analysis, and interviews. See Section 5.4.3 *Data Collection*) and for setting up the tooling to perform the actual calculation (see Section 5.4.4 *Study Execution*). Based on the time we spent on performing the case study, we estimate the effort for both parts to be ∼100 working hours.

Results: Figure 5.5a gives an overview of the costs that are saved for the system test case life-cycle activities (test creation, execution, and maintenance). In the figure, the number of planned test runs have been used as independent variable (x-axis). The y-axis shows the relative improvement (in percent) of test creation, execution, and maintenance costs compared with the original execution mode *fully-manual*: Different execution modes are compared with the original execution mode of our case study (*fully-manual*). However, this data is just an intermediate result on the way to calculate the overall system testing effort that is saved.



(a) Reduction of test creation, execution, and maintenance costs.

(b) Reduction of overall system testing costs.

Figure 5.5: Reduction compared to execution mode fully-manual.

Reduction of Overall Testing Costs

To estimate the benefit of our approach to the overall system testing costs, we use our relative benefit estimation model (see Section 5.5.1 *Benefit Estimation Model*) to extrapolate the costs for all system testing activities.

Results: Figure 5.5b shows the benefit for the overall system testing costs that is gained by our approach (as extrapolated by our benefit estimation model). Similarly to the previous figure, the number of test runs have been used as independent variable (x-axis). In this case study, the actual number of planned test runs is 40 – 60 (indicated by the gray area).[9] The y-axis of the figure shows the relative improvement (in percent) of the total system testing costs (as extrapolated by our benefit estimation model).

Based on this calculation, we can get the following information for the test suite of our case study: Similar to the results of the previous case study, the benefit of individual execution modes differ strongly. Compared to the original execution mode *fully-manual*, the execution modes *manually-or-automated* and *fully-automated* do not have a positive benefit until the test suite is executed more than ∼60 times resp. more than ∼110 times. The execution mode *semi-automated* has a positive benefit right from the beginning. In the relevant range of test runs (40 – 60, see Section 5.4 *Evaluation*), our calculation predicts that the execution mode *semi-automated* has a clear positive benefit. Implementing a *semi-automated* execution mode will lead to ∼20% – ∼30% reduction of the overall testing costs compared with the original execution mode *fully-manual*.

5.6 Summary

We addressed the problem of choosing execution modes for system test suites. To support this decision, we introduced a test effort estimation model that included expert estimations for creation, maintenance and execution of system test suites. The model can be used by test engineers to gain additional information about the test suite and helps to decide which parts of test suites to automate and which parts to execute manually.

In an industrial case study (41 test cases), we applied this estimation model for different test execution modes of the same test suite. We reported on how we estimated values for the cost factors of the model and calculated the total effort (in form of costs) for different execution modes and different numbers of test runs. The case study demonstrated that it is feasible to collect the data needed for the estimation model and that the efforts for different execution modes vary considerably. However, although our model uses the concept of costs to model testing efforts, we do not claim to forecast precise costs. Rather, our aim is to provide data that enables experts to compare different test execution strategies.

We furthermore performed a coarse estimation of the benefit that is gained by our approach. For our test suite of 41 test cases (40 – 60 planned test runs), performing our effort modeling approach and choosing an adequate execution mode would reduce the overall system testing costs by ∼20% – ∼30% (within two years).

[9] In this case study, one maintenance task is expected (see Section 5.4 *Evaluation*, #mainttasks = 1).

Chapter 6

Test Refactoring Using Grammar Inference

The study in Chapter 4 *Clones in Manual System Tests* has shown that industrial test suites contain many clones ($\sim$43 – 85%). Clones make tests complex to understand since they blow up test artifacts unnecessarily. Furthermore, they increase maintenance effort, since changes have to be performed multiple times.

Refactoring tests by extracting cloned parts into reuse components (e.g., subroutines that can be referenced) will avoid unnecessarily recurring maintenance tasks. In Chapter 4, we demonstrated that the size of manual system test suites is up to two times larger than necessary. However, test suites also have to be understandable in order to support maintenance activities. Therefore, the call structure between tests and reuse components has to be simple. Furthermore, reuse components have to represent meaningful parts of tests to be understandable. Extracting the right parts of tests to reuse components is crucial to support maintainability and understandability.

Example:

Figure 6.1 shows three automated test cases, which are testing the payment options of a web shop. The test cases are denoted in a keyword-driven notation[1] [Fewster and Graham, 1999] (a common notation for automated test). The test procedure of each test case consists of a sequence of (parameterized) references to keywords. For each keyword, a test script exists that will be executed automatically during test execution[2]. The idea of keyword-driven testing is to reuse keywords whenever possible. However, since all three test cases in the example are testing the same functionality, their test procedures contain large common (sub) sequences of references to keywords – test clones (colored in gray).

Figure 6.2 illustrates two different attempts to extract clones: Whereas Figure 6.2a shows the result of a naive clone extraction approach, the second attempt (Figure 6.2b), presents an approach in which overlapping clones have been considered to define reuse components. Both attempts result in clone free test suites, however, their quality differs: The second decomposition is not only easier to understand, since its call structure is simpler (1 vs. 2 levels of indirections), it is also easier to maintain since it is smaller (20 vs. 23 steps in total).

[1] This example does not follow the concrete syntax of a certain testing language or tool.
[2] The test scripts are not part of the example.

Tests:

Test: funds-transfer disabled

Step	Value
1. Login to System Admin Interface	
2. Set funds-transfer setting	*Disabled*
3. Logout	
4. Login to Webshop	*user1/pw*
5. Add Product to Shopping Cart	*1234*
6. Trigger Cart check-out	
7. Accept Terms & Conditions	
8. Verify funds-transfer payment is not available	

Test: funds-transfer just one order

Step	Value
1. Login to System Admin Interface	
2. Set funds-transfer setting	*just one*
3. Logout	
4. Login to Webshop	*user1/pw*
5. Add Product to Shopping Cart	*1234*
6. Trigger Cart check-out	
7. Accept Terms & Conditions	
8. Select funds-transfer payment	
9 Enter bank account	*account1*
10. Place Order	
11. Verify: System rejects order	

Test: funds-transfer default setting

Step	Value
1. Login to Webshop	*user1/pw*
2. Add Product to Shopping Cart	*1234*
3. Trigger Cart check-out	
4. Accept Terms & Conditions	
5. Select funds-transfer payment	
6. Enter bank account	*account1*
7. Place Order	
8. Verify: Order successfully placed	

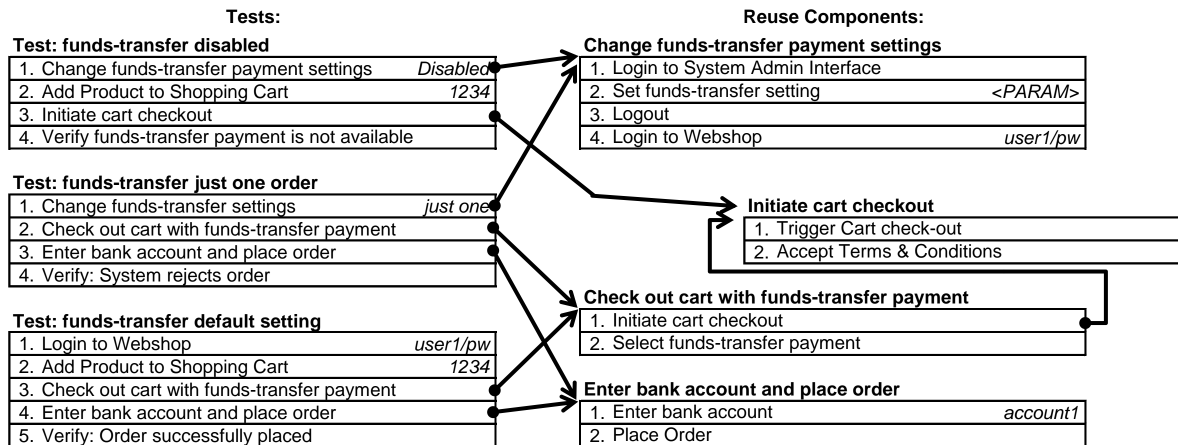
Figure 6.1: An example of three test cases containing clones (cloned parts are highlighted).

Problem and Consequences:

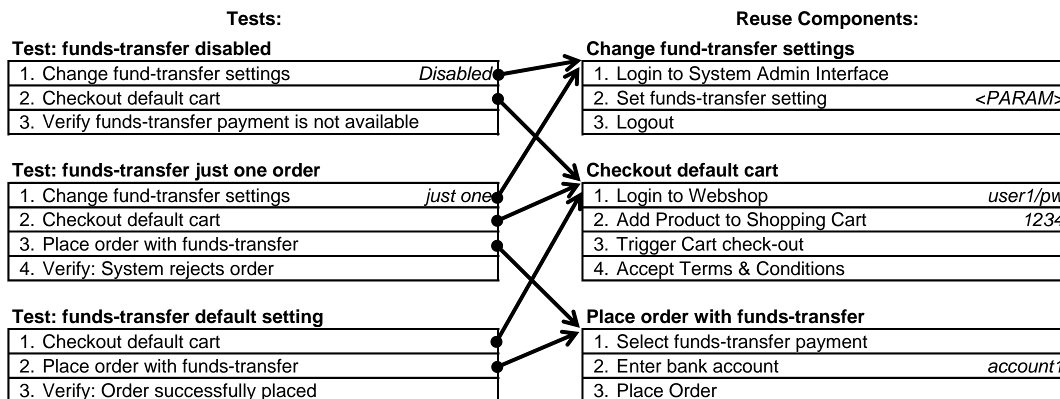
Finding the right abstractions to create reuse components is challenging. Unfortunately, knowledge about clones is not sufficient to remove them: Clones often overlap partly which makes it difficult to decide which sections of test procedures shall be extracted. If this decision is made wrongly, test suites will be difficult to understand since the call structure of test cases and reuse components will be unnecessarily complex. Furthermore, reuse potential will not be leveraged optimally which leads to test suites which will be larger than necessary and therefor costly to maintain.

Approach and Contributions of This Chapter:

In this chapter, we present a method to support test engineers in manually extracting clones from test cases to make test suites better maintainable and understandable. Our approach focuses but is not limited to automated test cases denoted in keyword-driven like notations. It complements existing clone detection techniques by presenting a proposal of how overlapping clones can be efficiently extracted to reuse components. Key concept of our approach is to apply grammatical inference algorithms to test procedures. We use the Sequitur algorithm [Nevill-Manning, 1996], which infers a hierarchical structure from a sequence of tokens. Sequitur creates a grammar in which recurring parts have been replaced by grammar rules. Using this grammar, we propose a decomposition of a test suite in which all (overlapping) clones have been efficiently extracted to reuse components. A visualization shows all test cases, reuse components, and how they are referenced by each other.



(a) Result of a naive approach to extract clones.



(b) A decomposition in which overlapping clones have been considered.

Figure 6.2: Two different ways of decompose a test suite to extract clones (the original test suite is shown in Figure 6.1).

We surround our approach by two studies: In a pre-study (Section 6.2), we first show the relevance of the problem by analyzing the extent and nature of overlapping clones in automated system tests. The study revealed that significantly large parts of the test clones overlap (up to 31.7%). In a evaluating study (Section 6.4), we challenge our approach and demonstrate that it helps test engineers in practice in performing refactorings. The study revealed not only that our approach is applicable in real world settings, but also that the results help test engineers in refactoring clones. Both studies have been performed in an industrial setting and were applied on real-world test suites. Finally, we present an estimation of the benefit that is gained by applying our approach to industrial test suites. Parts of the content of this chapter have been published in [Hauptmann et al., 2015].

6.1 A Conceptual Test Meta-Model

The approach we present in this paper is not restricted to certain test automation approaches, languages or tools. We nevertheless have some basic constraints on how automated test suites are designed. In the following, we present our conceptual test meta-model (see Figure 6.3) illustrating those constraints:

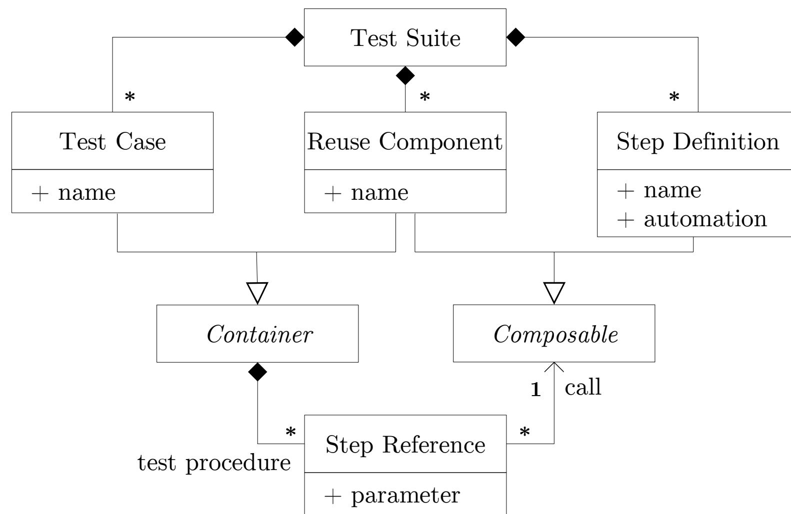


Figure 6.3: Conceptual test meta-model.

Test Procedures and Reusable Test Steps:

The essential part of *test cases* are *test procedures* which are sequences of steps. However, the fundamental concept of our test model is that steps are reusable within the test suite. Therefore, test procedures hold just references (*step reference*) to test steps, whereas the actual steps are defined for the whole test suite (*step definition*). Furthermore, step references can be equipped with parameters, which will be passed to the referenced steps during the execution of the test suite.

Reuse Components:

To make recurring sequences of test steps reusable by multiple test cases, they can be extracted to *reuse components*. Similar to test cases, reuse components have test procedures. In contrast, reuse components can be referenced by steps and thereby form a way to decompose test suites. The examples in Figure 6.2 follows our conceptual test model and make use of reuse components.

Test Automation for Test Steps:

To make test suites automatically executable, each step is equipped with an *automation*. A common approach is to define step automations declaratively in form of sequences of interactions with the system's user interface. Those sequences are not executable themselves, but

have to be interpreted by a testing tool. Many test automation tools provide reuse mechanisms to extract parts of interaction sequences (similar to reuse components for test procedures). Figure 6.4 shows an example of an interaction sequence in the test automation tool Ranorex[3].

#	Action				
1	Mouse	Click	Right	Center	NewDatabase
2	Mouse	Click	Left	Center	AddGroup
3	Key Sequence	Ranorex			(No item)
4	Key Sequence	{Return}			(No item)

Figure 6.4: Example of a step automation in Ranorex.

6.2 Significance of Overlapping Clones

Before we present our approach to support test engineers in refactoring clones, we investigate the significance of overlapping clones in the current practice of automated system testing.

6.2.1 Study Goal and Research Questions

We define the goal of our study using the goal definition template of Wohlin et al. [2000] and break it down to three research questions:

We analyze *overlapping test clones*
for the purpose of *characterizing and understanding*
with respect to *extent and complexity*
from the viewpoint of *test engineers*
in the context of *industrial software projects.*

RQ 1: To what extent are test suites affected by cloning?

To understand the relevance of overlapping clones, we first need to know to what extent test suites are affected by cloning in general. Therefore, the prevalence of cloning in test suites constitutes the basis for the following investigations.

RQ 2: To what extent do clones overlap?

This question analyzes if overlapping clones are a common phenomenon in system tests. The more often this phenomenon exists, the more extensive are its negative consequences.

RQ 3: How many clones overlap concurrently?

Whereas the previous research question analyzed to what extent clones overlap, this question investigates whether overlapping clones are also a complex phenomenon. The more clones overlap concurrently, the more complex it is to refactor those spots and extract them to reuse components.

[3] http://www.ranorex.com

6.2.2 Study Objects and Case Study Context

We answer the research questions by analyzing two automated system test suites of Munich Re (see Section 1.5), which we named MR1 and MR2[4]. Both test suites are created using the test automation tool Ranorex and consist of two conceptual parts, which we call test definition and step automation: (1) The *test definition*, which contains test cases, reuse components, and test procedures. (2) The *step automation* in which steps are automated in form of sequences of interactions with the user interface. Since those step automation procedures provide reuse mechanisms too, similarly as required by our conceptual meta-model, we consider them as individual study objects on which we apply our approach too. Table 6.1 summarizes all parts.

Table 6.1: Study objects.

	MR1		MR2	
	Test Definition	Step Automation	Test Definition	Step Automation
No. Elements	77 Tests	91 Step Auto.	20 Tests	27 Step Auto.
Min. Length	2 Steps	1 Step	2 Steps	1 Step
Max. Length	35 Steps	68 Steps	5 Steps	66 Steps
Total Length	391 Steps	959 Steps	71 Steps	314 Steps

6.2.3 Data Collection

To answer the research questions, we perform automated clone detection to all four study objects. We consider a *test clone* as a (consecutive) sequence of at least three steps within a test procedure appearing at least twice in a test suite. A *clone group* contains all test clones that have the same content. The detection of clones has been performed automatically using the open source quality assessment toolkit ConQAT[5] [Juergens et al., 2009].

RQ 1: To what extent are test suites affected by cloning?

To determine to what extent test suites are affected by cloning, we calculate the *clone coverage* CC for all study objects, which tells the relative amount of a test suite that is covered by cloning.

RQ 2: To what extent do clones overlap?

To figure out whether overlapping clones are a common phenomenon, we measure the portion of cloned sections that is affected by overlapping clones. We introduce a new metric *overlapping clone coverage* ($OCC[n \geq i]$) which computes the relative amount of a test suite that is covered by at least i clones concurrently. Figure 6.5 illustrates this metric OCC.

[4] For non-disclosure reasons, we are not allowed to publish the actual names.

[5] `http://www.conqat.org`

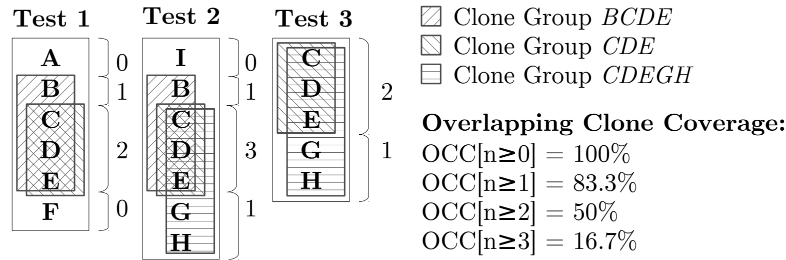


Figure 6.5: Example of the metric *overlapping clone coverage*. Clones are marked as hatched areas. The annotations next to the tests show the number of overlapping clones.

We first calculate the metric $OCC[n{\geq}2]$ for all three study objects which shows the relative sizes for a test suite that is covered by at least two clones concurrently. By dividing $OCC[n{\geq}2]$ by the clone coverage CC calculated in RQ 1, we get the share of by cloning affected test steps that is covered by overlapping clones. In the example of Figure 6.5, 60% of all cloned test steps are affected by overlapping clones.

RQ 3: How many clones overlap concurrently?

To figure out how complex the structure of overlapping clones is, we calculate the metric OCC for all occurring numbers of concurrently overlapping clones ($OCC[n{\geq}2]$, $OCC[n{\geq}3]$, $OCC[n{\geq}4]$, ...).

6.2.4 Calculation of Overlapping Clone Coverage

In the following, we describe how we measured the metrics clone coverage and overlapping clone coverage:

Step 1 – Transformation of tests to text files

As a first step, we transformed the test suite into a plain text format. We created a parser for the test suite format of Ranorex. Since the goal of the clone detection is to find candidates for clone refactoring, we ignored all parameters. Result of the transformation was a set of text files for each study object. Each text file represents a single test case. The content of each text file represents the test procedure, whereas each line represents a test step.

Step 2 – Perform clone detection

We performed automated clone detection and manually inspected random samples of clones from each test suite to calibrate the transformation into text files and the configuration of the clone detector. The result of the clone detection was, beyond some overview statistics and figures, an XML file containing detailed information on the findings. The XML file contained all detected clone groups and the included clones with their exact location.

Step 3 – Cutting tests into clone snippets

Based on the clone detection result, we cut each text file into disjoint text snippets along the boundaries of starting and ending positions of clones. Thereby, we got text snippets that are entirely covered by either no clone, one clone, or several overlapping clones. Furthermore, we annotated each snippet with the number of clones covering the snippet.

Step 4 – Calculate coverage

We grouped the text snippets of each test suite based on their number of overlapping clones. By summing up the number of lines of each snippet from each group, we got the size of the text bodies covered by each number of concurrently overlapping clones. The metrics clone coverage CC and overlapping clone coverage OCC can then be calculated by summing up the text body sizes of the corresponding groups.

6.2.5 Results

During the study execution, it turned out that the tests in MR2 do not contain meaningful test procedures, but just call a few steps. We did not find any clones in this study object, hence, we ignore MR2 Test Definition in the remainder of the discussion. Table 6.2 summarizes the clone coverage and the share of cloning that is affected by overlapping clones.

Table 6.2: Clone coverage and share of cloning that is affected by overlapping clones.

	MR1		MR2	
	Test Definition	Step Automation	Test Definition	Step Automation
Clone Coverage	48.3%	44.3%	–	36.1%
$\geq$ 2 Conc. Overl. Clones	31.7%	25.2%	–	18.0%
$\geq$ 3 Conc. Overl. Clones	2.6%	7.8%	–	5.8%
$\geq$ 4 Conc. Overl. Clones	1.1%	2.4%	–	–
$\geq$ 5 Conc. Overl. Clones	–	0.5%	–	–

RQ 1: To what extent are test suites affected by cloning?

The share of the test suite that is affected by cloning in general (clone coverage) ranges from 36.1% (MR2 Step Automation) to 48.3% (MR1 Test Definition).

RQ 2: To what extent do clones overlap?

The share of cloning that is affected by at least two overlapping clones concurrently ranges from 18% (MR2 Step Automation) to 31.7% (MR1 Test Definition).

RQ 3: How many clones overlap concurrently?

The complexity of overlapping cloning ranges from three (MR2 Step Automation) to five (MR1 Test Definition) concurrently overlapping clones.

6.2.6 Interpretation and Discussion

Based on the results, we draw the following conclusions:

Cloning is common in system test suites

We found considerable amounts of cloning in each test suite (up to 48.3%). This supports our hypothesis from earlier studies [Hauptmann et al., 2013, 2012b] that cloning in industrial system tests is a common phenomenon.

Clones commonly overlap in system tests

Significantly large parts of the test clones overlap (up to 31.7%). This supports our hypothesis that overlapping clones are a common phenomenon in industrial system test suites.

Overlapping clones in system tests can be complex

Although the maximum number of concurrently overlapping clones differed among the test suites, up to five test clones overlapped concurrently. This supports our hypothesis that overlapping clones in system tests can complicate clone removal since clone relations are difficult to grasp.

6.3 Approach: Supporting Test Engineers in Extracting Clones from Automated System Tests

In the previous section, we investigated the significance of overlapping clones in system test suites. In the following, we introduce an approach to support test engineers in extracting clones to make test suites easier to understand and maintain. Existing clone detection approaches indicate which parts of a test suite are affected by cloning. This helps to extract clones that do not overlap, however, if clones overlap, existing approaches reach their limits. Our approach complements existing clone detection techniques by giving constructive help for removing overlapping clones by extracting them to reuse components. We describe our approach in two parts. We first describe the workflow in which we apply our approach. Second, we describe in detail how we generate refactoring proposals, which is the key part in our approach.

6.3.1 Workflow

The workflow of our approach consists of the following three parts (see Figure 6.6):

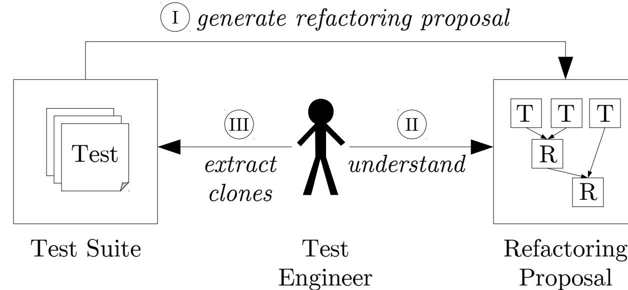


Figure 6.6: The workflow of our approach.

Part I – Generate refactoring proposal

We automatically generate a *refactoring proposal*, which contains an outline of the test suite that is free of cloning since all clones have been extracted efficiently to reuse components. It therefore depicts a clone free version of the given test suite (see Section 6.3.3 for details of the generation process).

Part II – Understand clone relations

The generated refactoring proposal provides insights for test engineers to understand clone relations within a given test suite. It shows how the cloned parts can be cut efficiently into reuse components, as well as how they are connected by call structures.

Part III – Extract clones

The idea of clone removal is to make test suites easier to comprehend and maintain. However, there are cases where adding reuse components introduces additional complexity that make test suites more difficult to understand. Therefore, we consider clone removal as a creative process that should not be done fully automatic. Instead, a test engineer should decide whether it pays off to perform a refactoring or not. More specifically, we suggest that test engineers use our generated refactoring proposal to gain better understanding about how the cloned parts of tests are connected with each other and to decide if and how to refactor tests. The size as well as the number of references of the proposed reuse components indicate the refactoring potential for cloned parts and point out starting points for manual clone removal by experts. We expect test engineers to build upon the refactoring proposal to develop own ways of clone removal which suites the context of the test suite.

6.3.2 Foundation: Grammar Inference and Sequence Modeling

Grammar inference aims at acquiring grammars based on examples of a language. Our approach makes use of *sequence modeling*, the subfield of grammar inference that examines a single sequence of tokens and forms a model of it, for example as state machine or grammar. This model can then be used to reproduce the original sequence [Nevill-Manning, 1996].

Several approaches for sequence modeling exist. For our work, we use Sequitur [Nevill-Manning and Witten, 1997], an algorithm and tool that infers hierarchical structures from a given sequence of symbols and produces a context-free grammar that is able to reproduce the original input sequence. Sequitur processes input sequences incrementally and replaces repetitive phrases with references to grammatical rules that generate that phrases.

Example of Sequitur: Figure 6.7 shows an example of a sequence S (*baabcabdabcabdab*) and a grammar that is generated by Sequitur. The input sequence S has been constructed using the terminal symbols a, b, c, d. Recurring phrases within the sequence S are extracted by Sequitur to production rules. For each production rule, a new nonterminal symbol (A, B) is introduced that replaces the extracted phrase whenever it occurs. Furthermore, the body of each production rule is subject to extraction, too: For example, the sequence *abcabd* appears twice within the input sequence S and has been extracted to the production rule A. However, both, the remaining input sequence as well as the newly created production rule A contain the phrase *ab* which consequently has been extracted to a new rule B.

Input:
S → baabcabdabcabdab

Output:

S → baAAB	(*generates* : *baabcabdabcabdab*)
A → BcBd	(*generates* : *abcabd*)
B → ab	(*generates* : *ab*)

Figure 6.7: A Sequitur example.

Goal and Strategy of Sequitur: The overall goal of Sequitur is to reduce the overall number of symbols within a context-free grammar. Unfortunately, finding the smallest (context-free) grammar is a well-known NP-complete problem. Sequitur is a greedy algorithm that is known to approximate good results [Nevill-Manning, 1996] and operates in space and time that is linear to the input size [Nevill-Manning and Witten, 1997].

Sequitur pursues this goal by eliminating duplication in (the right parts of) production rules. It processes input sequences symbol by symbol and updates the overall grammar whenever duplications are detected. The resulting grammar is uniquely defined by the following two constraints [Nevill-Manning, 1996]:

Digram Uniqueness: No sequence of two consecutive symbols is allowed to appear more than once in the grammar.

Rule Utility: Each production rule has to be used at least twice within the grammar.

Performance of Sequitur: To demonstrate the compressing effect of sequitur, we are comparing two examples: The first example (Figure 6.8a) shows a minimal input sequence and the result that is generated by Sequitur. In this example, a subsequence of the two symbols *ab* appear twice in a row (the input sequences has a length of four symbols in total plus one symbol for the name of the sequence). This subsequence is the smallest example that

violates the constraint *digram uniqueness*, which will lead Sequitur to extract appearances of this subsequence (production rule A). Furthermore, since the subsequence ab appears twice, it satisfies the constraint *rule utility*. Comparing the size (number of symbols) of the input and output shows, that, in this minimal example, sequitur does not only miss its aim to reduce the size of a given grammar, the size of the result is actually larger and grew by 20% compared to the initial grammar (6 symbols instead of 5). However, once we modify the input sequence slightly (the redundant subsequence is longer and appears more often), the Sequitur algorithm becomes more efficient resulting in an output grammar that is 20% smaller (the overall size of the grammar reduced from 10 to 8 symbols – see Figure 6.8b).

Input:
S → abab (*length* : 5 *symbols*)

Output:
S → AA (*length* : 3 *symbols*)
A → ab (*length* : 3 *symbols*)

(a) Number of steps: 6/5 = 120% (The resulting grammar is 20% larger)

Input:
S → abcabcabc (*length* : 10 *symbols*)

Output:
S → AAA (*length* : 4 *symbols*)
A → abc (*length* : 4 *symbols*)

(b) Number of steps: 8/10 = 80% (The resulting grammar is 20% smaller)

Figure 6.8: Two minimal Sequitur example showing the performance of the algorithm.

Nevill-Manning [1996] reports on experiments that show that the compressing ability of Sequitur can keep up with established general purpose compression algorithms and tools such as gzip and ppm. Furthermore, Sequitur shows its strength on large input sequences. It operates in space and time that is linear to the input size [Nevill-Manning and Witten, 1997]. For our studies, we build on its Java implementation available at the Sequitur website[6].

6.3.3 Generating Refactoring Proposals Using Sequitur

The goals of Sequitur and of our approach are rather similar: Sequitur removes redundant parts from production rules (sequences of terminal and nonterminal symbols) to reduce the overall size of context-free grammars. Likewise, we want to remove redundant parts of test procedures (sequences of test steps) to avoid unnecessary maintenance tasks. We make use of this similarity and apply Sequitur to analyze recurring parts in test procedures. More specifically, we transform a test suite into an input sequence for Sequitur and apply the algorithm on the generated input sequence. Sequitur will identify recurring subsequences in the generated input sequence and will generate a context-free grammar in that all recurring parts have been extracted to production rules. We then use the optimized grammar to rebuild the test suite and present it to test engineers as a proposal for refactoring. We expect this approach to support test engineers in extracting clones as follows:

Clones in test procedures are identified: Our approach makes use of Sequitur's ability to identify recurring parts in sequences of symbols. By transferring a test suite into a format that can be processed by Sequitur, sequences of test steps in test procedures will be identified that appear multiple times (test clones).

[6] `http://www.sequitur.info`

Test engineers learn how to extract test clones: Once Sequitur identified recurring sequences of symbols in the generated input sequence, it will extract them to production rules – similarly as we want to extract clones to reuse components. By visualizing the contents and structure of production rules of the resulting grammar, test engineers can transfer this information to the original test suite and thereby learn how to extract test clones to reuse component ideally.

Overlapping clones are handled automatically: Sequitur will furthermore identify recurring parts in existing production rules and will restructure rules if necessary – similarly as we want to restructure existing reuse components to extract (overlapping) clones to reuse components. This leads to the fact that clones that overlap und therefore are difficult to extract manually are processed by our approach automatically.

In the following, we describe the generation of the refactoring proposal in detail:

Step 1 – Annotate clones in test suite

At first, we perform automated clone detection to identify clones in test procedures. We annotate all those parts of the test suite that are affected by at least one clone. This step will work as prefilter for Sequitur so it can focus just on those parts that are affected by cloning.

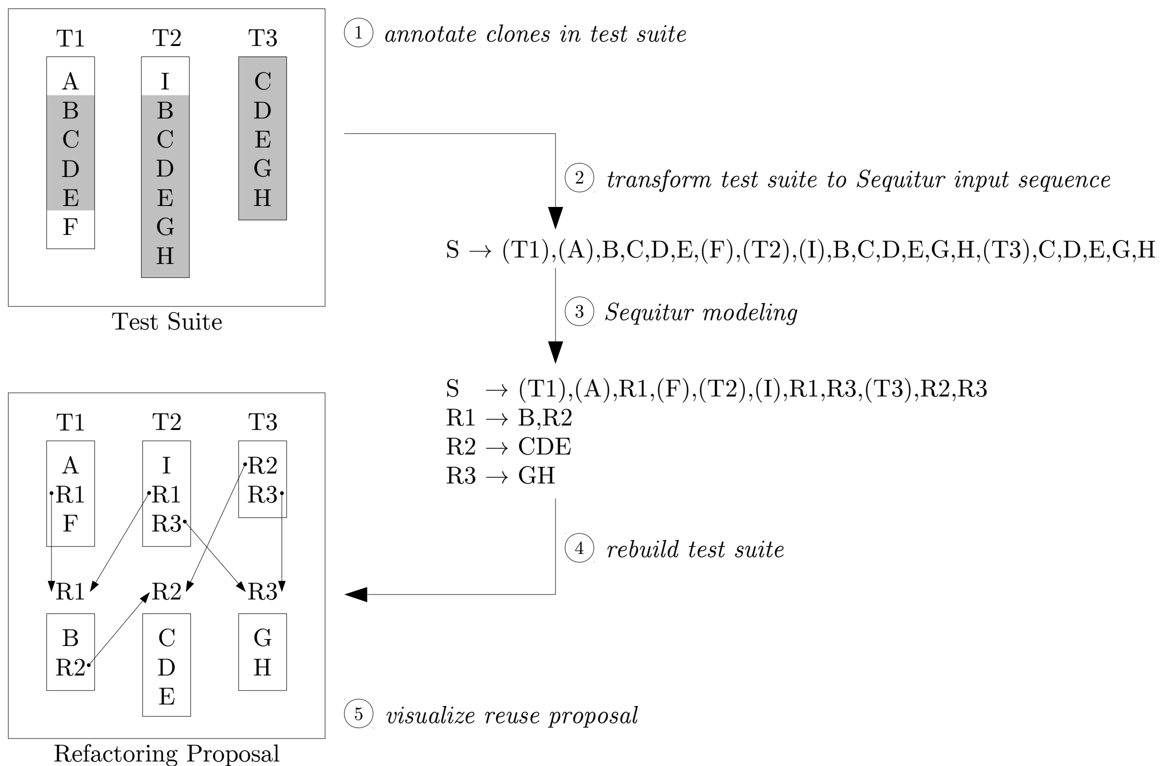


Figure 6.9: Example of the generation of refactoring proposal.

Step 2 – Transform test suite to Sequitur input sequence

We then transform the whole test suite into a single input sequence for Sequitur. We transfer the test procedures of each test case into sequences of text tokens, where each test step is represented by a single token consisting of the name of the referenced step. Similar to our pre-study, we normalized test steps by removing parameters.

We additionally add sentinel tokens at the beginning of each token sequence to mark the start of the test case (see $(T1)$, $(T2)$, $(T3)$ in ② in Figure 6.9). The sentinel tokens are annotated with the name of the starting test cases and are necessary to rebuild the test suite (see step 4). Furthermore, we annotated each step resp. the corresponding token if it is not affected by cloning (see parenthesized symbols in Figure 6.9). The concatenation of all token sequences is used as input sequence for Sequitur (see ② in Figure 6.9).

Step 3 – Sequence modeling

Now, we apply Sequitur to our generated input sequence. Sequitur will identify recurring sequences of tokens in the input sequence and will extract them to production rules. The production rules themselves will be restructured again if they contain recurring sequences of tokens. We adapt Sequitur slightly: Sentinel symbols and steps that are not affected by cloning will never be extracted to production rules and therefore will always stay part of the input sequence S.

The resulting grammar consists of a modified input sequence and a set of production rules. Both do not contain any recurring sequences of tokens (see ③ in Figure 6.9).

Step 4 – Rebuild test suite

We rebuild the test suite based on the grammar which has been generated by Sequitur. We cut the modified input sequence S along the sentinels we added in step 2 and reconstruct the test procedures of each test case.

Each production rule R of the generated grammar will be transformed into a reuse component and added to the test suite. We name the reuse component based on the production rules name ($R1$, $R2$, ...).

Since we abstracted some information from the test suite, the rebuilt test suite will just be an outline of the initial test procedures. Details such as test data or comments will not be part of the test procedures. However, since the goal of our approach is to provide an overview of the new test suite's structure, we consider this information as sufficient.

Step 5 – Visualize reuse proposal

As a last step, we visualize the new test suite to make it easily readable for experts. Goal of the visualization is to understand which parts of a test procedure have been extracted to reuse components, as well as how tests and reuse components are connected by calls. Therefore, we visualize the rebuild test suite in a graph-like structure (see ⑤ in Figure 6.9).

6.4 Evaluation

To analyze the ability of our approach to help test engineers in practice, we apply our approach in an industrial setting.

6.4.1 Study Goal and Research Questions

We define the goal of our study using the goal definition template of Wohlin et al. [2000]:

We analyze	*our approach*
for the purpose of	*evaluation*
with respect to its	*ability to support clone refactoring*
from the viewpoint of	*test engineers*
in the context of	*industrial software projects.*

We define the research questions along the workflow of our approach: RQ 1 targets the general idea of our approach to apply sequence modeling algorithm to test procedures. RQ 2 and RQ 3 evaluate the understandability of the results whereas RQ 4 investigates whether our approach is helpful for test engineers.

RQ 1: Can sequence modeling be applied in our setting?

At first, we have to figure out whether sequence modeling is suitable to be applied in our context. Crucial aspects are if Sequitur's time and memory consumption is suitable to be applied to give rapid feedback to test engineers.

RQ 2: Is the size of the refactoring proposals manageable by experts?

Formal grammars can be complex and difficult to understand. This question investigates whether the size and complexity of the generated refactorings are understandable for test experts.

RQ 3: Do the proposed refactorings result in meaningful reuse components?

Goal of clone refactoring is to make test suites easier to understand and maintain. Therefore, the decomposition of test procedures into reuse components has to be comprehensible. Reuse components have to cover meaningful and cohesive parts of test suites.

RQ 4: Do our refactoring proposals help test engineers?

Finally, we investigate whether our approach provides beneficial information to support test engineers in improving a test suite's quality regarding understandability and maintainability.

6.4.2 Data Collection

For this study, we use the same study objects as in the pre-study (see Section 6.2.2).

RQ 1: Can sequence modeling be applied in our setting?

To investigate whether sequence modeling algorithms can be applied in our context, we measure the *execution time* as well as the *memory usage* of automatically generating refactoring proposals using our approach.

RQ 2: Is the size of the refactoring proposals manageable by experts?

To investigate whether the generated refactoring proposals are understandable by test experts, we measure their size and complexity. We indicate both using the following metrics: *number of reuse components*, *size of reuse components*, and the *depth of the structure of all refactorings*.

RQ 3: Do the proposed refactorings result in meaningful reuse components?

To figure out whether our approach proposes understandable units of reuse, we rely on experts' opinions. We consider a reuse component as understandable, if experts are able to give it a short and concise name describing its scope.

RQ 4: Do our refactoring proposals help test engineers?

To figure out whether our approach provides valuable input for test engineers, we rely on their decisions to implement the proposed refactorings. We perform interviews and ask test engineers whether the proposed refactorings are technically feasible and if they will implement them[7]. Additionally, we ask whether they generally consider our approach as beneficial to support their daily work as a test engineer, and if yes, what concrete benefits they see.

6.4.3 Study Execution

In the following, we describe how we generated refactoring proposals as well as how we assessed the quality of the outcome:

Generation of refactoring proposals

We used the tooling of our pre-study (see Section 6.2) to detect clones in the same study objects as in our pre-study (see Section 6.2.2). We automatically transformed the study objects into input sequences for Sequitur. After the processing, we rebuild the basic structure of the test suite based on the generated grammar and visualized it using the graph visualization program Graphviz[8].

Expert interview

We interviewed the lead test engineer of the study object MR2 for two hours. The interview started with an introduction of our approach and the workflow how it is supposed to be used by test engineers. We then guided the test engineer through all proposed reuse components for study object MR2. The test engineer inspected the refactoring proposal's visualization as well as the corresponding clones in the actual test suite. For each proposed refactoring, the test engineer answered the following questions:

[7] Assuming that they have the necessary time to perform the refactoring.

[8] `http://www.graphviz.org`

(Q1) *Can you give this reuse component a short and concise name?*

(Q2) *Is it technically feasible to extract this reuse component?*

(Q3) *Would you implement this reuse component (slight modifications are allowed)?*

At the end of the interview, the test engineer gave an additional assessment of the helpfulness of the approach and results.

Action research

Since no test engineer of test suite MR1 was available for our study, we simulated an expert evaluation by letting two researchers of our research group assess the understandability of the generated refactoring proposals. Both researchers have been working in the field of system test quality for several years and are experienced in test automation. However, since both researchers are not part of the testing team of MR1, we focus just on Q1. Both researchers went through all refactoring proposals collaboratively and tried to find short but comprehensive names for the proposed reuse components based on the sequence of test steps. After each reuse component was named, they went a second time through all proposed components to double check the created names and ensure consistency.

6.4.4 Results

We present the results along the research questions:

RQ 1: Can sequence modeling be applied in our setting?

The generated Sequitur input sequence had a length of 1518 (MR1) and 432 tokens (MR2). Generating refactoring proposals took ∼9 seconds in total for MR1 respectively ∼5 seconds in total for MR2. However, the execution of Sequitur took less than half a second in both cases. The memory usage of Sequitur was 8 Megabyte for MR1 and 1 Megabyte for MR2 (see Table 6.3).

Table 6.3: Execution times and memory usage.

	MR1	MR2
Length input sequence	1518 tokens	432 tokens
Load test suite	1,040 ms	493 ms
Clone detection	4,742 ms	2,556 ms
Transformation	29 ms	4 ms
Sequitur	317 ms	78 ms
Visualization	2,711 ms	1,927 ms
Total	8,839 ms	5,058 ms
Memory (Sequitur)	83,968 kB	1,024 kB

RQ 2: Is the size of the refactoring proposals manageable by experts?

Our approach proposed between 18 and 50 reuse components covering 2 to 8 steps each. The reuse components were structured in a maximum of 3 nested calls (see Table 6.4).

Table 6.4: Size and complexity of reuse components.

	MR1		MR2	
	Test Definition	Step Automation	Test Definition	Step Automation
No. Elements	23 Comp.	50 Comp.	–	18 Comp.
Min Size	2 Steps	2 Steps	–	2 Steps
Max. Size	7 Steps	8 Steps	–	8 Steps
Max. Nesting	3 Calls	3 Calls	–	2 Calls

RQ 3: Do the proposed refactorings result in meaningful reuse components?

Both, the test engineer and the two researchers were able to find short and concise names for all reuse components of MR2's step automations resp. MR1's test procedures.

RQ 4: Do our refactoring proposals help test engineers?

The test engineer we interviewed rated all presented reuse components as technically feasible to extract. For 10 out of 18 reuse components, the test engineer claimed that he would implement the presented reuse component either as is or with slight modifications. In additional 6 cases, he would perform refactorings with a larger scope, for example, by restructuring parts of the test suite to address several proposed reuse components at once. In the remaining 2 cases, the benefit that one would get would not justify the refactoring effort. In not a single case, the test engineer did find our approach not useful (see Table 6.5).

Table 6.5: Consequences drawn by experts.

	MR2
I will implement this reuse component (maybe modified).	10 (56%)
I will address this reuse component by a bigger refactoring.	6 (33%)
I appreciate this reuse component, but it will not pay off.	2 (11%)
The proposed refactoring is not useful at all.	0 (0%)

During the interview, we also gathered qualitative results: Based on the test engineer's experience, the biggest challenge in test refactoring is understanding relations between tests which was addressed well by our approach. Visualizing clone results helped the test engineer to understand the commonalities of test cases. Many of the generated reuse components uncovered reuse potential that was new for the test engineer. Additionally, the visualization made quick wins of test refactorings easily visible.

Furthermore, the test engineer stated that our approach helps to uncover potential for more far-reaching quality improvements since many of the presented refactorings indicated conceptual problems of design of the test suite.

6.4.5 Threats to Validity

In this section, we discuss threats to the external and internal validity of the study and describe how we mitigated them:

External Validity:

We conducted the pre-study and the evaluation study of this paper on a set of two automated test suites. The limited number of study objects restricts the generalization of the results, and thus threatens the external validity of the studies. However, the study objects are real-world examples, since they are developed and applied in industry. Furthermore, we expect these test suites to be representative for test suites in this domain regarding their size and complexity.

Internal Validity:

In our studies, we presented measures for the complexity of the interdependencies between clones (overlapping clones) and the complexity of the results generated by our approach. These metrics might not cover all aspects of complexity. However, we interviewed the practitioners who developed the test suites informally about the complexity of the interdependencies between overlapping clones and the results generated by our approach and found that their impression of the complexity matched our metrics.

In one part of our study, researchers acted as test engineers. They went through reusable components that have been suggested by our approach and tried to find short and concise names for them. As these researchers are not part of the project team of the study objects, this might bias the results. We mitigated this threat by triangulating the findings among the participating researchers. Furthermore, the researchers who conducted the study had several years of experience in the field of software testing and software systems of Munich Re.

6.4.6 Interpretation and Discussion

We draw the following conclusions from our evaluation:

Sequence modeling is applicable for test refactoring.

The key concept of our approach, to apply sequence modeling algorithms to uncover refactoring potential of test suites seems technically feasible. The execution time as well as the memory need enables our approach to be applied in online tools giving rapid feedback to test engineers.

Results of our approach are understandable by experts.

The refactoring proposals generated by our approach can be large in numbers, however, the complexity seems to be understandable by experts. The proposed reuse components cover cohesive units of test procedures (resp. step automations). Furthermore, the decomposition proposed by our approach seems to make sense for test experts as well.

Results help test experts in clone refactoring.

In most cases, test experts would follow our recommendations and implement the proposed reuse components. In some cases, our approach was even uncovering structural problems of the test suite, which require far-reaching refactorings.

6.5 Benefit Estimation

In the previous section, we analyzed the ability of our approach to support test engineers in making automated test suites easier to maintain by removing test clones. The results of the study show that in 16 out of 18 cases, test engineers appreciate our refactoring proposals and would implement them. However, performing refactorings to remove test clones requires an initial invest that might not pay off. In this chapter, we perform a rough benefit estimation showing how much overall system testing effort can be saved by our approach, and under what circumstances our approach pays off.

6.5.1 Benefit Estimation Model

To estimate the overall benefit of our approach, we use the relative benefit estimation model that we introduced in Chapter 5 *Choosing Execution Modes* (Section 5.5.1 *Benefit Estimation Model*). This model is based on the idea, that the overall costs of system testing are composed of the costs of each testing activity of system testing (see Figure 6.10). Furthermore, the model defines the relative sizes of each activity within the whole calculation. Knowing these proportions between system testing activities allows calculating how the overall system testing costs develop when the costs for single testing activities change.

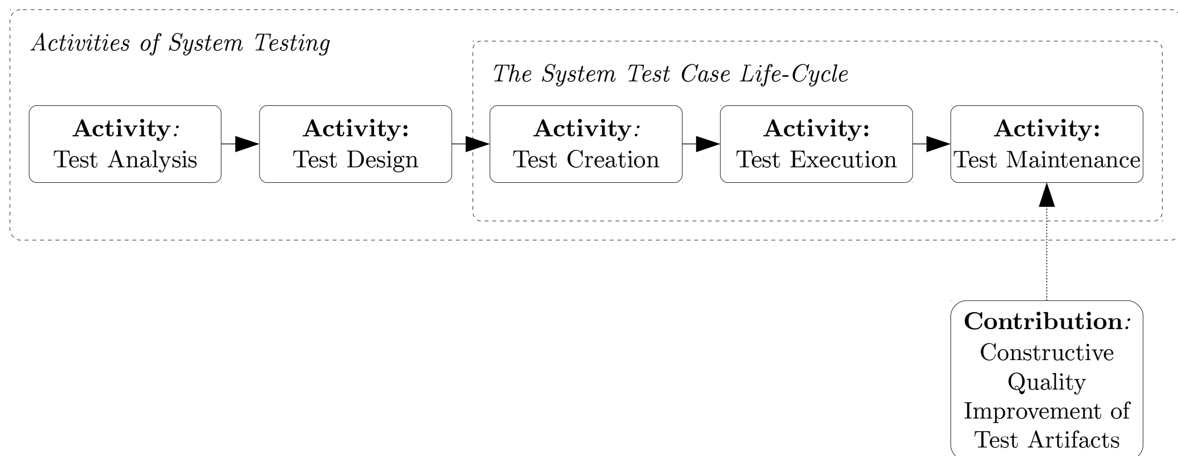


Figure 6.10: System testing activities that are addressed by the contribution of this chapter.

The approach that we presented in this chapter addresses the activity *test maintenance* (see Figure 6.10). The benefit of our approach for this activity can easily be estimated by bringing together the maintenance effort that will be saved with the estimated overall maintenance effort of a software system. To estimate how much costs can be saved in total, we use our relative benefit estimation model to extrapolate the benefit for the overall system testing costs.

6.5.2 Assumptions

In the remainder of this benefit estimation, we make the following assumptions:

Maintenance is equally distributed: To simplify the estimation of saved maintenance effort, we assume that all parts of the test suite are similarly likely to be affected by maintenance tasks. More specifically, we assume that one maintenance task affects all reuse components.

Refactoring proposals are practical: We assume that the languages and tools with that the test suites are created and maintained provide suitable refactoring and reuse mechanisms to extract clones. We expect that there are no cases in that refactorings cannot be performed due to limitations of the test tooling.

All refactorings are similarly expensive to perform: Lastly, to ease the estimation of clone removal effort, we assume that all proposed refactorings are similarly labor expensive to perform.

6.5.3 Calculation of the Benefit for the Activity *Test Maintenance*

We calculate the benefit of our approach for the activity *test maintenance* as follows: The basic idea of our calculation is that we compare the effort of removing clones (performing refactorings that extract cloned parts to reuse components) with the maintenance benefit that is saved over time by having test cases with less redundancy. We estimate the costs to remove test clones by the effort to perform the refactorings that have been proposed by our approach. Assuming that all refactorings are similarly difficult to perform, we multiply the average effort of performing a single refactoring with the number of accepted refactoring proposals to calculate the overall refactoring effort:

	Avg. effort to perform a single refactoring
×	Number of refactoring proposals (that have been accepted by engineers)
	Overall effort of performing refactorings

The basic idea of our benefit estimation is that removing redundant parts of test cases will lead to reduction of unnecessary maintenance effort. Hence, to estimate the maintenance benefit that is gained by removing cloned parts of tests, we roughly calculate in as much the test suite is getting smaller by removing clones. We estimate this by comparing the initial number of test steps with the number of test steps of our overall refactoring proposal (as it results from our approach). Since test engineers might not realize all proposed refactorings

(see study in Section 6.4 *Evaluation*), we multiply the number of reduced test steps with the percentage of refactoring proposals that have been accepted (in one test suite within our case study) by test engineers. Thereby, we get the actual reduction of test steps.

To make this value easier to compare with the refactoring effort, we use the average length of a reuse component to calculate how many test reuse components correspond to the number of reduced test steps:

	No. of test steps (original test suite)
−	No. of test steps (after performing all refactoring proposals)
	Reduction of test suite (in no. test steps)
×	Percentage of refactoring proposals that have been accepted by engineers.
	Actual reduction of test suite (in no. test steps)
÷	Avg. number of steps per reuse component
	Actual reduction of test suite (in no. reuse components)
×	Avg. maintenance effort per reuse component (for one maintenance task)
	Maintenance benefit gained (for one maintenance task)

Finally, to calculate the overall benefit that is saved by removing test clones, we multiply the maintenance benefit of one maintenance task with the number of expected maintenance tasks. By subtracting the effort of performing refactorings from the overall maintenance benefit, we get the overall benefit that is gained over time by applying our approach:

	Maintenance benefit gained (for one maintenance task)
×	Number of expected maintenance tasks (per reuse component)
	Maintenance benefit gained (over time)
−	Overall effort for performing refactorings
	Overall benefit gained by our approach (over time)

To estimate the extent to that the overall maintenance costs decrease by applying our approach, we furthermore need an estimation of the overall maintenance costs of a test suite. We calculate this value as follows:

	Avg. maintenance effort per reuse component (one maintenance task)
×	Number of expected maintenance tasks (per reuse component)
	Maintenance costs without our approach (over time)

6.5.4 Data Elicitation

We apply our effort estimation model to the study objects that we used in the two previous studies in Section 6.2 *Significance of Overlapping Clones* and Section 6.4 *Evaluation*. The data that is missing for our calculation is substituted as follows:

Number of Test Steps

We measure the size of the test suites by counting the number of test steps. For each test suite (resp. part of the test suites), we counted the size of the original test suite and of the modified test suite in that all refactoring proposal have been applied (see Table 6.6).

Table 6.6: Size of test suites.

	MR1		MR2	
	Test Definition	Step Automation	Test Definition	Step Automation
Original Test Suite	391 Steps	959 Steps	71 Steps	314 Steps
All Refactoring Proposals Applied	314 Steps	805 Steps	71 Steps	274 Steps
Reduction (absolute)	77 Steps	154 Steps	0 Steps	40 Steps
Reduction (relative)	20%	16%	0%	13%

Accepted Refactoring Proposals

The second study of this chapter (Section 6.4 *Evaluation*) revealed that 16 out of 18 presented refactoring proposals have been considered as candidates to be implemented by test engineers. We use this estimation for our calculation and assume that, for all study objects, 89% of all proposed refactorings are accepted by the test engineers and therefore will be implemented.

Average Number of Steps per Reuse Component

We calculate the average steps per reuse component using our study objects from Section 6.2 *Significance of Overlapping Clones*.

Average Maintenance Effort per Reuse Component (One Maintenance Task)

To estimate the effort that is necessary to perform a single maintenance task on a reuse component, we adapt existing data that we elicited in the case study from Section 5.4 *Evaluation*. Although those values have been elicited for manual test cases, we expect them to be sufficiently precise to approximate maintenance of test scripts in our calculation.

Average Effort to Perform a Single Refactoring

We expect the implementation of a refactoring proposal as similarly labor intensive as a typical refactoring task. Therefore, we use the average maintenance effort also to approximate the average effort to perform a single refactoring.

Number of Expected Maintenance Tasks per Reuse Component

Unfortunately, we have no detailed information on the number of maintenance tasks that are expected for our study objects. We can only compare our cases with the study described in Section 5.4 *Evaluation*. In that study, test engineers estimated that each test step will be subject to maintenance once within a life span of two years. However, we believe that this estimation cannot be transferred to our case directly. The test suites in our study are automated test scripts, which are far more detailed compared with the manual test descriptions of our previous study. Compared with manual test cases, we expect automated test scripts to be more brittle if it comes to minor changes, for example, modifications of the user interface, leading to higher maintenance effort compared with manual test cases. Since we have no detailed information, we leave the number of expected maintenance tasks as an independent variable in our calculation.

6.5.5 Quantitative Benefit

The contribution presented in this chapter addresses the activity of maintaining test cases (see Figure 6.10). To find out in as far the overall system testing costs are reduced by applying our approach, we first calculate in as far the activities of test maintenance benefit from our approach. Having this information, we can use our relative benefit estimation model to calculate in as much the overall costs of system testing benefit.

Reduction of Test Maintenance Costs

Figure 6.11a gives an overview of the maintenance costs that are saved by applying our approach. In the figure, the number of expected maintenance tasks are used as independent variable (x-axis). The y-axis shows the relative improvement (in percent) for the activity *test maintenance* (as calculated by our benefit estimation model) with respect to maintenance costs. Each data series represents one test suite from our case study.

Results: We can get the following information for the maintenance costs in our case study: Based on our assumptions about refactoring and maintenance costs of reuse components, removing clones will pay off after two to three maintenance tasks only. The exact number of maintenance tasks from that our approach starts to pay off is influenced by the relation between implementation time for one refactoring, the maintenance effort for one maintenance tasks, and the overall reduction of the test suite. However, this data is just an intermediate result on the way to calculate the overall system testing costs that are saved by our approach.

Reduction of Overall System Testing Costs

We used our relative benefit estimation model to extrapolate the overall system testing costs that are saved by applying our approach (see Figure 6.11b). Similar to the previous figure, the number of expected maintenance tasks are used as independent variable (x-axis). The y-axis shows the relative improvement of the overall system testing costs for each test suite.

Results: Similarly to the reduction of the maintenance costs, removing clones will have a positive effect to the overall system testing costs after two to three maintenance tasks only. However, in the context of our case study, the actual number of expected maintenance tasks

is unknown. Therefore, we cannot decide whether our approach will pay off in this case. However, assuming that the test suite is maintained 5 - 10 times [9], the overall system testing costs will be reduced up to ∼10%.

6.5.6 Qualitative Benefit

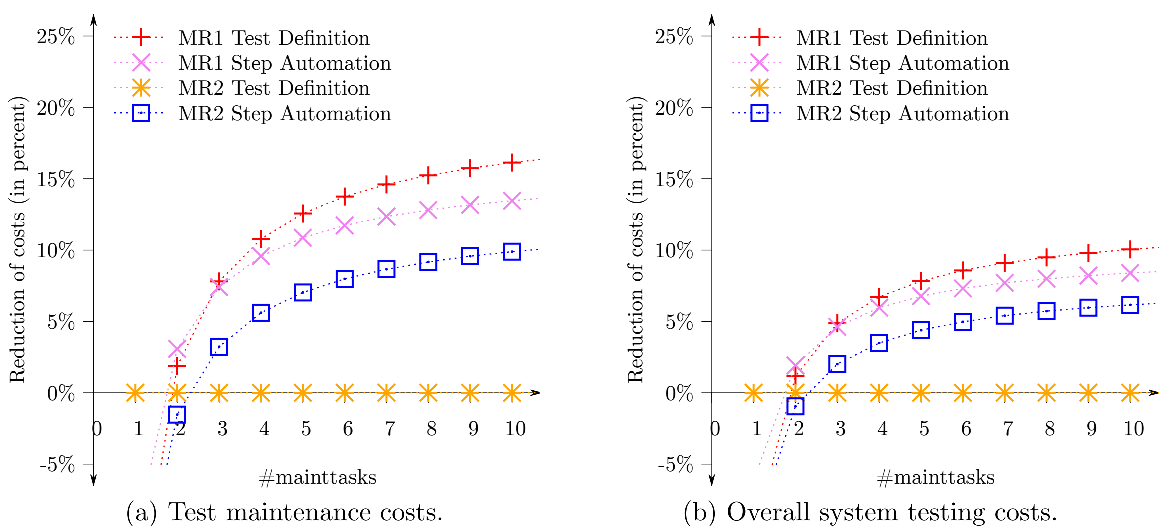
Besides the quantitative results, supporting test engineers in removing (overlapping) clones in test suites does also have qualitative benefits:

Flexibility to adaptions: Removing redundant parts from test suites enables performing maintenance tasks faster since unnecessarily duplicated maintenance activities are avoided. However, this increase in maintenance speed does not only reduce the costs to perform changes, but also minimizes the time until a test suite is ready for operational use again.

More resources for new test cases: The saving in test maintenance can be used to improve existing system testing: More resources are available to perform further test analysis and create additional test cases covering aspects that have not been tested yet.

Lower risk of inconsistent bug fixes: Cloning does not only unnecessarily raise maintenance expenses, it also increases the risk of inconsistently performed changes [Juergens et al., 2009]. In the case of bug fixes, inconsistent changes can leave some parts of the test suite erroneous still having the original bug that should have been fixed. Hence, supporting test engineers in removing clones also help to reduce the risk of uncompleted bug fixes.

[9] In the previous study, just one maintenance tasks was expected. However, that test suite contained (manual) test descriptions, which are rather abstract compared to the test scripts of our current test suite. We expect test scripts to be more brittle to changes of the system under test and therefore leading to more frequent maintenance activities.



(a) Test maintenance costs.

(b) Overall system testing costs.

Figure 6.11: System testing costs saved by our approach.

6.6 Future Work

Some important questions are still unanswered.

Generalizability of the evaluation:

The evaluation study of this paper has been applied on a limited set of study objects which have been taken from the same application domain and company. Replicating the study in different industrial contexts, different companies and using different test automation tools will enable to draw more generalizable conclusions of the abilities of our approach.

Benefits and limitations of clone refactoring in general:

In our evaluation, we were just able to indicate the benefit of test clone removal, but were not able to gain a deeper understanding of its limitations. In further experiments, one has to figure out to what extent tests are better understandable after clone removal as well as when to stop refactoring, because tests may become less understandable by introducing reusable components.

Rapid feedback for test engineers:

To support test engineers in their everyday work, our tool support is not sufficient. Our tooling has to be extended so it can be used instantaneously by test engineers while they are working with the test suites, for example, by extending existing test automation tools to display clone information as well as possible refactorings directly within the test suites.

Continuous quality improvement:

To maintain the quality of automated test suites over time, a continuous quality assurance process is essential. Based on our approach, metrics could be defined to indicate aspects of a test suite's understandability and maintainability. Together with predefined thresholds, such metrics can be used to define quality goals that are used within a continuous quality assurance process.

Refactoring for other types of artifacts:

Cloning is a relevant topic not only in system testing, but also for many other software engineering artifacts. We think it is worthwhile to evaluate the application of our approach to other artifacts such as source code or software design documents.

6.7 Summary

In this chapter, we addressed the challenge of improving the quality of test artifacts. We proposed an approach to support test engineers in removal of overlapping clones in automated test suites with the goal to make them better understandable and maintainable. The proposed approach complements existing clone detection techniques by using sequence modeling to analyze overlapping test clones and to propose suitable units of reuse. The visualized results

can be used by test engineers in gaining understanding of the test suite and to perform refactorings.

In two studies, we analyzed the phenomenon of overlapping clones and evaluated our approach in industry. In a pre-study, we showed that test clones in practice often overlap and can be complex to understand. This makes it challenging finding good strategies to remove clones by refactorings.

Our evaluation study revealed not only that our approach is applicable in real world settings, but also that the results help test engineers in refactoring clones. In particular, test engineers would perform a large part of the refactorings to remove clones suggested by our approach. Furthermore, they stated that the approach helps to understand the internal structure of a test suite, in terms of cloning and the resulting dependencies between test cases. This helps them not only to remove clones, but also gives them the possibility to reveal deeper problems within existing reuse structures. Furthermore, they are able to identify larger refactorings that improve the understandability and maintainability.

Furthermore, we performed a coarse benefit estimation to find out if and when removing clones pays off. For our two study objects, performing our refactorings has a positive benefit if the test suite is subject to maintenance for at least $2 - 3$ times. Assuming that the test suite is maintained for at least $5 - 10$ times, the overall system testing costs will be reduced by up to $\sim$10%.

Chapter 7

Natural Language Test Smells

In Chapter 5 *Choosing Execution Modes*, we found out that in various situations manual testing is the cheapest way of system testing. The alternative, test automation, has high setup costs and therefore does not pay off in all situations.

Unfortunately, manual tests are often of poor quality and written without software engineering best practices in mind. For example, the study presented in Chapter 4 *Clones in Manual System Tests* revealed that natural language system test descriptions contain a significant amount of cloning, which can considerably increase the costs for maintaining and executing them. Other common quality problems result from the use of natural language in manual test description. For example, ambiguity and incomprehensibility both affect the test case's executability and maintainability.

Example:

Table 7.1 shows two manual test descriptions that contain quality problems. Both examples are taken from original test cases from our industry partners, however, we modified them slightly for non-disclosure reasons. The first example (Table 7.1a) shows a snippet of a test description that guides the tester to repeat the two preceding test steps. However, the number of repetitions is not defined and therefore has to be chosen freely by the tester: Different testers may perform a different number of repetitions. This ambiguity makes executing this test case indeterministic since the outcome of the test case is not reliable. Thus, results of test runs cannot be compared with each other. This threatens the expressiveness and usefulness of a test suite in general.

The second snippet (Table 7.1b) guides the tester to verify the system's response depending on a decision that has been made in a previous test step. At this point, the test case contains a branch of the test flow that has been phrased in natural language which may lead to two problems: First, it is not clear if the condition for the branch is satisfied the same way in each test run. Similar to the first example, this may lead to indeterministic test executions and uncomparable test results. Second, since the condition is phrased in natural language, it is not obvious to notice for the tester. This may lead to test cases that are difficult to comprehend or even misapprehended, both negatively affecting the test case's executability and maintainability.

Table 7.1: Example of two test descriptions.

(a) A manual test that is ambiguous which hampers its repeatability.

	Step Description	**Expected Result**
	...	...
Step 6	Repeat Step 4 and Step 5 as long as you want.	...
	...	...

(b) A test that is difficult to comprehend since it contains a branch phrased in natural language.

	Step Description	**Expected Result**
	...	...
Step 12	...	The Entries differ depending on the chosen View Mode (Step 3): ...
	...	...

Problem and Consequences:

Tests written in natural language often do not obey well-established software engineering quality principles. This leads to problems affecting all system test execution life-cycle activities such as, creating new test cases, executing test cases and maintaining them.

To improve the quality of test cases, we need ways to detect quality issues that negatively affect testing activities. Furthermore, detecting quality problems has to be performed continuously to keep good test case quality on the long run. However, since test suites can be large (>1000 test cases – see Chapter 4 *Clones in Manual System Tests*), manual quality assessment techniques are not practicable. To assess the quality of large test suites continually, quality issues have to be detected automatically.

Approach and Contributions of This Chapter:

Other software engineering artifacts, such as source code or unit tests, suffer from similar problems (cf. Section 3 *State of the Art*). Time pressure or inexperience make developers ignore well-known design rules. This results in source and unit test code that is less maintainable and understandable. As a countermeasure, for both source code and unit tests, so called *bad code smells* [Fowler, 1999] and *test smells* [Meszaros, 2007; Meszaros et al., 2003; van Deursen et al., 2001] have been established as indicators for design flaws. In the last years, several studies have shown the negative effect of the existence smells in program code and test code with respect to maintainability and code comprehension [Abbes et al., 2011; Bavota et al., 2012; Khomh et al., 2009; van Deursen and Moonen, 2002].

In this chapter, we transfer the basic concept of bad smells from source code and test code to system test artifacts. Adopting existing code and unit test smells, we define smells for manual tests in natural language - *Natural Language Test Smells (NLTS)*.

We first present a smell definition template as a framework to support test engineers in defining quality problems in form of natural language test smell. This template is connected with a basic quality model which allows selecting smells fitting to individual project needs of test suites. Furthermore, we define six natural language test smell for test cases written in natural language. To perform smell detections continually in large test suites, we present tool support to uncover smells automatically in test artifacts. In an industrial case study, we show the ability of our approach in identifying relevant quality defects in test suites. Finally, we present an estimation of the benefit that is gained by removing natural language test smells from industrial test suites.

Parts of the content of this chapter have been published in Hauptmann et al. [2013].

7.1 Definitions and Terminology

In the following section, we define the concept of *Natural Language Test Smells*, an adaption of bad code smells for test cases written in natural language. Furthermore, we introduce an ontology connecting our smells with a basic quality model and propose a template to define smells.

7.1.1 Definition: Natural Language Test Smells

Borrowing from the original idea of *bad code smells* [Fowler, 1999], we define *Natural Language Test Smells* as indicators for quality violations in test case written in natural language (see Section 2.2.2 *Test Case Representations* for an example of natural language test cases). We define natural language test smells as way to describe quality issues having the following characteristics:

Natural Language Test Smells are Automatically Detectable:

Since test suites can be large (>1000 test cases), manual reviews to find smell findings are not practicable. Therefore, we focus on the subtype of *automatically detectable smells* for that automatic detection techniques exist. Being able to detect smell findings in test case descriptions automatically supports test engineers in improving the quality of test artifacts continuously.

Natural Language Test Smell Findings have Concrete Locations:

Each smell finding points at a concrete location within a test case description where it occurs. This concrete location is necessary to guide test engineers in removing quality problems since it shows at which location to perform improvements.

Natural Language Test Smells Indicate Quality Defects:

Smell findings are often difficult to detect automatically which makes smell detection techniques imprecise and leads to findings that are not always accurate. Therefore, we do not necessarily consider smell findings as quality defects but only as indicators to quality problems that have to be assessed by test engineers.

Natural Language Test Smells are Traced to Activities:

Each natural language test smell is mapped to system test execution live-cycle activities (or sub-activities) that are negatively affected by the smell. This mapping allows selecting smells according to project specific requirements. Furthermore, the mapping helps to identify impacts of smells and allows assessing finding's severity.

7.1.2 Taxonomy and Quality Definition

To put the concept of natural language test smells on a solid base, we present an ontology connecting the used terms. The ontology relates relevant items of the project context with our general quality model. Furthermore, it shows how natural language test smells are defined and how smell instances are linked to test cases (see Figure 7.1).

Our ontology has strongly been influenced by the *Quamoco approach* and its idea of *activity-based quality modeling* [Deissenboeck et al., 2007; Lochmann, 2010; Wagner et al., 2008, 2015, 2012]. The core idea of this approach is to aim at defining quality by how well certain properties of a product support the activities that are applied on the product. We transfer this fundamental ideal to the context of test cases written in natural language.

Project Context:

Central item of the project context is the *Artifact Model* which shows the structure of a *Test Suite* and how it is broken down to (nested) *Test Entities* such as test cases or test steps (see Section 2.2.2 *Test Case Representations*). Test suites are subject to *Activities* (see *System Test Execution Life-Cycle* in Section 1) which are performed by *Stakeholders*, such as test engineer or test executors.

Quality Model:

Goal of the *Quality Model* is to form a clear picture of what is good and bad quality of a test suite. Central point of the quality model are *Quality Problems* that have negative impacts on activities. The quality model furthermore defines *Quality Factors* that declare the visible manifestation of quality problems in test entities that have impacts on the stakeholder's ability to perform a certain activity [Femmer et al., 2015].

Natural Language Test Smell Definition:

In natural language test smells, we explicitly focus on quality problems. To detect natural language test smells, a *Detection Technique* finds instances of quality factors in test entities. To define all relevant aspects of natural language test smell, we propose a smell definition template in the next Subsection (see Section 7.1.3 *Smell Definition Template*).

Natural Language Test Smell Instances:

For each appearance of a quality factor, a *Smell Finding* is created. Each smell finding is equipped with a *Finding Location* pointing at the position within a test entity where a quality factor has been identified by a detection technique. Furthermore, smell findings may cause *Quality Defects* which are concrete instances of the quality problem that is caused by the quality factor.

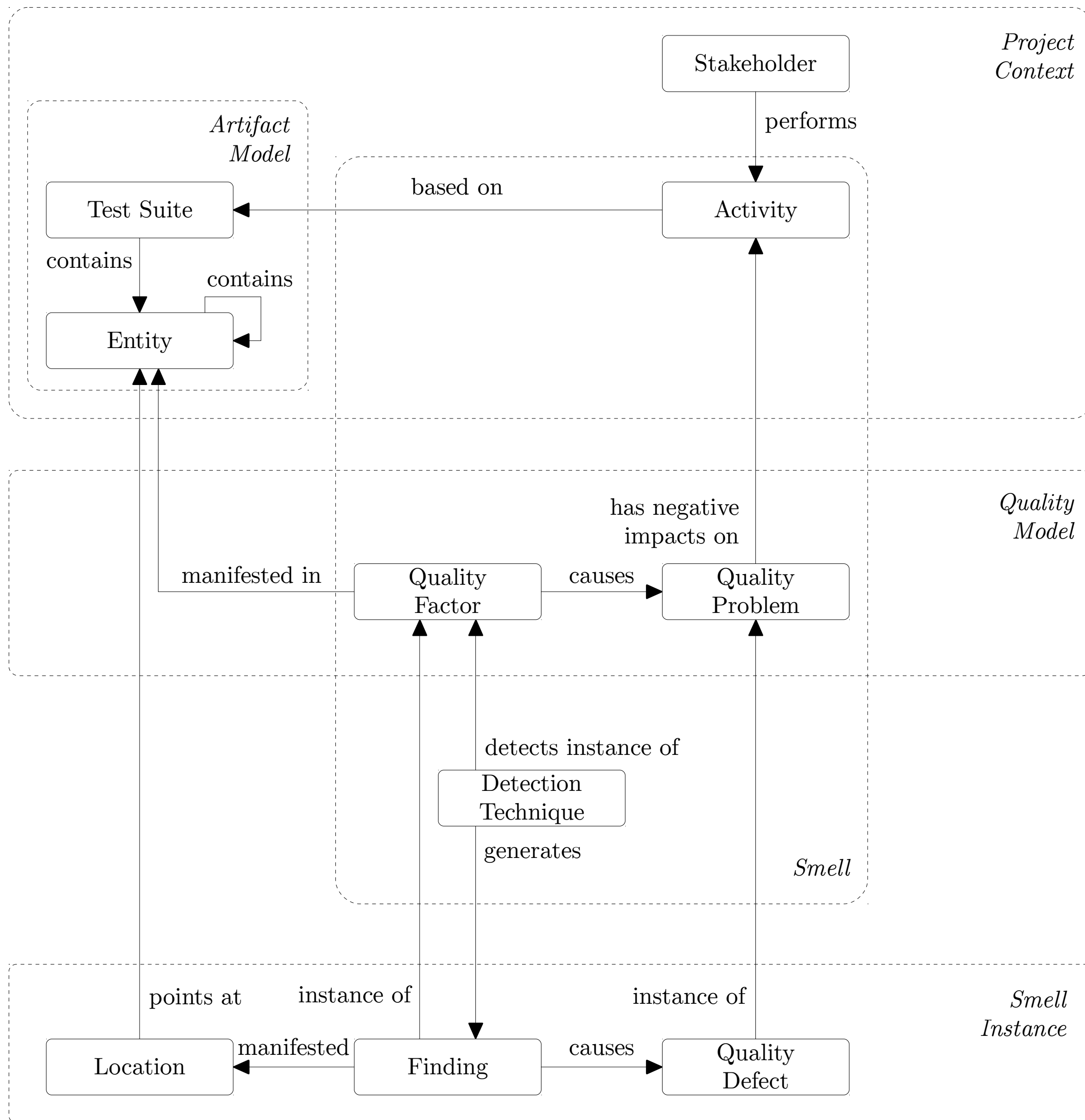


Figure 7.1: An ontology relating terms and concepts of natural language test smells.

Precision of Detection Techniques and Relevance of Smell Findings:

The precision of a smell finding is determined by its detection technique's ability to correctly identify quality factors. A smell finding that has been identified by a smell detection technique, but is not a correct instance of its quality factor is considered as a *false positive*. On the other side, a smell finding that is a correct instance of the smell's quality factor is considered as a *true positive*. However, true positive smell findings do not necessarily lead to quality defects. We consider just those smell findings as *relevant* that actually lead to quality defects (see Figure 7.2)

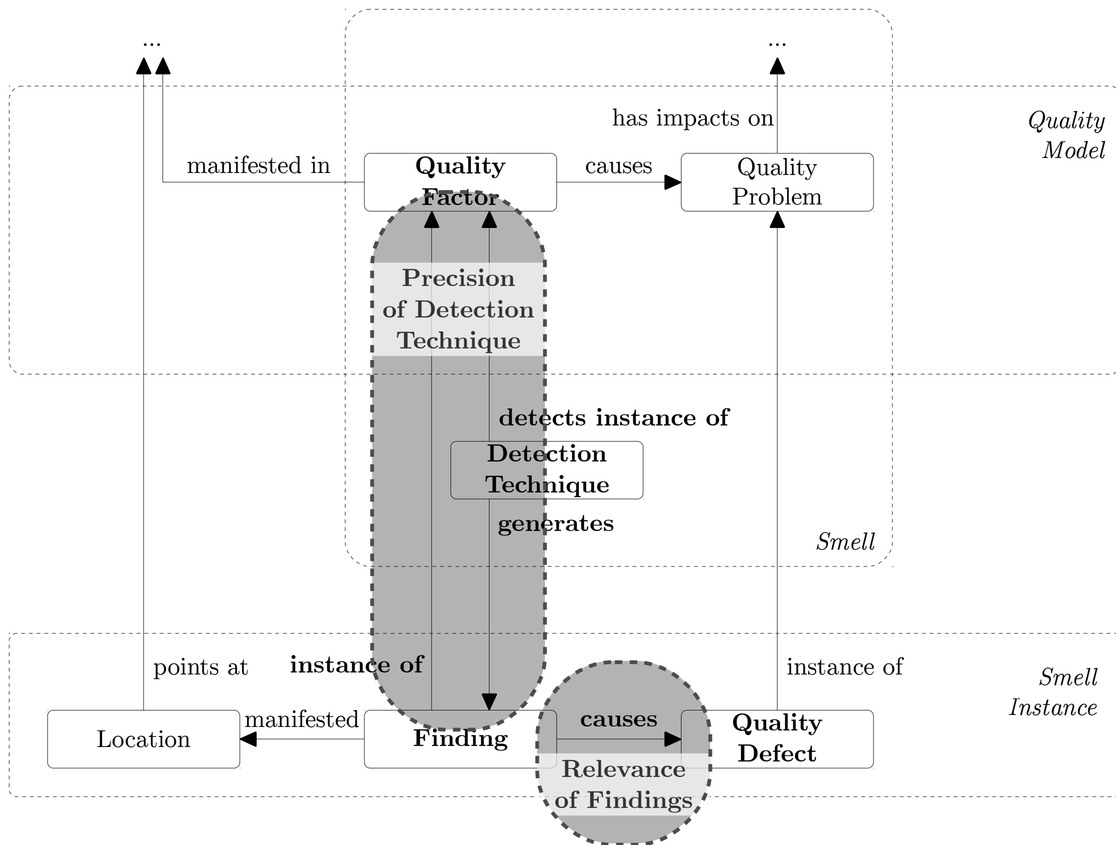


Figure 7.2: Precision and relevance of smell findings.

Figure 7.3 shows an example of all aspects of our ontology for the smell *Ambiguous Tests*.

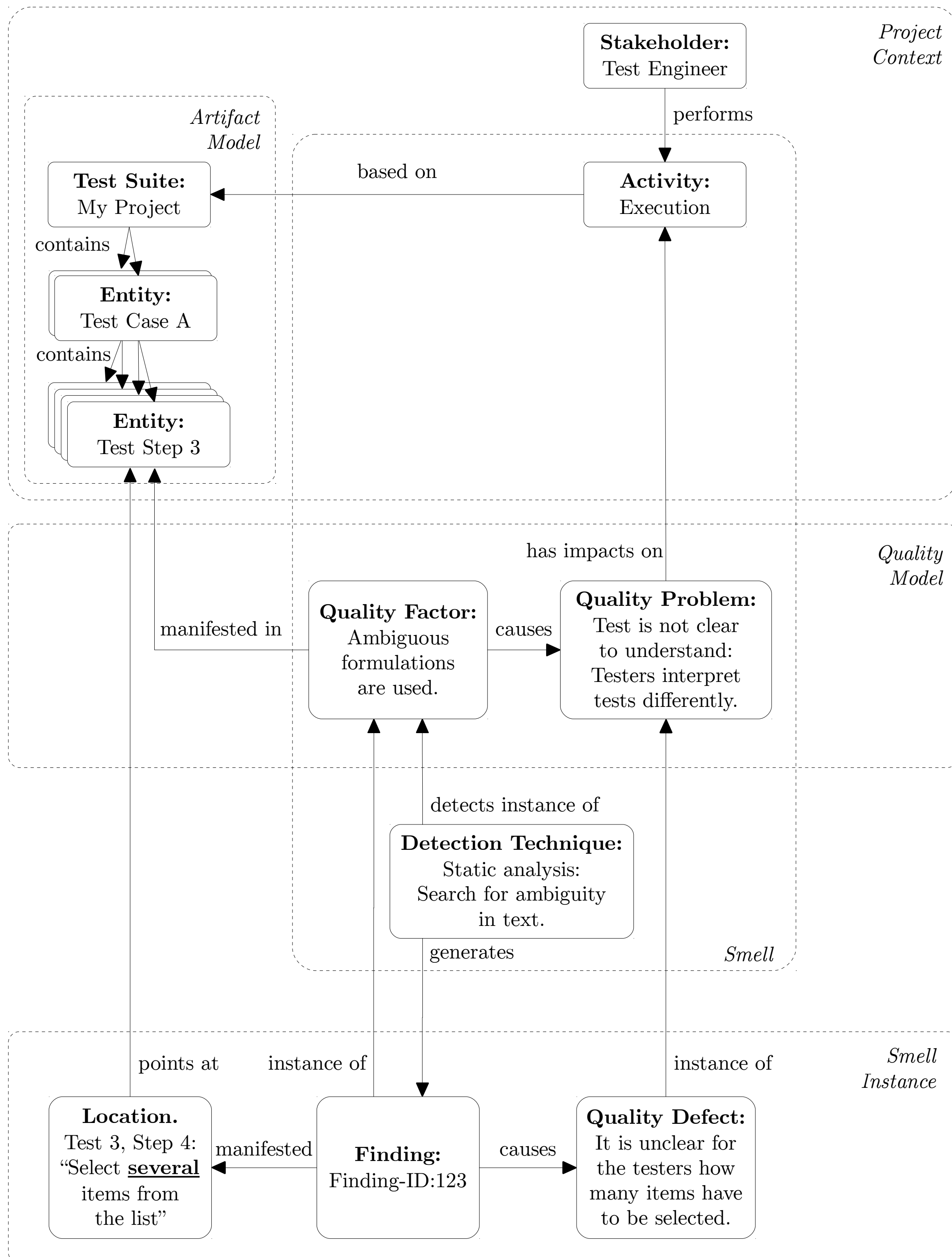


Figure 7.3: An example instance of our ontology showing the smell *Ambiguous Tests*.

7.1.3 Smell Definition Template

We introduce a smell definition template as a framework to support quality engineers in defining smells. Goal of the template is to precisely define all relevant aspects of smells.

Smell Name:	*A unique name to identify the smell.*
Smell Category:	*Classification into a certain category to structure smells.*
Quality Factor:	*Description of the visible characteristic of the smell.*
Affected Activities	*Activities that are negatively affected by the smell.*
Quality Problems:	*Description of the problem that is caused by the quality factor.*
Example:	*Example of a smell finding. Optionally with a description of the quality defect and how to remove it in this example.*

7.2 Approach: Natural Language Test Smells

In the following section, we present a list of natural language test smell as the result of our experiences in quality evaluation of manual system test suites in industry for the last four years. First, we describe the process how we selected the smells. Afterwards, we introduce each smell in detail.

7.2.1 Selection of Natural Language Test Smells

The list of natural language test smell that are presented in this chapter have been developed based on our experiences in quality evaluation of manual system test suites in the last years. It is not complete but reflects a selection of natural language test smell that we developed together with our industry partners. We designed the smells to address real-world challenges regarding test execution and maintenance of our industry partner's test projects. Although this way of selecting smells makes our list not complete, it ensures that each smell addresses relevant quality aspects of test cases. Furthermore, it allows us to evaluate our smells in real-world settings. We developed the list of smells as follows:

Smells for Unit Test Cases

As a first starting point, we adapted existing unit test cases from van Deursen et al. [2001] and Meszaros et al. [2003] (see Section 3.4.1 *Refactoring and Smells of Source Code and Unit Tests*) and transfer them to test cases in natural language when possible. For each of the proposed unit test smells, we checked two criteria:

First, we evaluated whether the existing unit test smells are applicable to tests written in natural language. Since the smells by van Deursen et al. [2001] and Meszaros et al. [2003] are designed for test cases that are programmed in source code (such as JUnit test cases), many smells are not applicable in our context: For example, the smell *Sensitive Equality* [van Deursen et al., 2001] proclaims to avoid using the `toString()`-method to compare Java objects. Since the test cases in our context are written in natural language, this smells is not applicable.

Second, we focused on those smells that point at quality factors that are actually causing problems in our context. For example, the smell *Mystery Guest* [van Deursen et al., 2001] proclaims that test cases should be self-contained and do not reference external resources. However, although this might be the case for automated unit tests, for manual system testing, referencing to fixed sets of test data is not considered to lead to quality problems in our context.

Characteristics of Natural Language Text

As another source for natural language test smell, we took into consideration the special characteristics of manual test cases, namely the fact that they are written in natural language. Based on the experiences of our industry partners, we identified characteristics of natural language that lead to challenges for test execution and maintenance. For example, natural language enables to phrase text in a way that is ambiguous or complicated to understand. This makes test descriptions difficult to understand and can lead to misinterpretations and can cause deviations in test execution.

Focus on Automatically Detectable Smells

Finally, we focused on those smells that are automatically detectable. We skipped smells for that the implementation of automated detection techniques are enormous costly so that their development costs will not pay off by the benefit of the smell.

Based on this selection, we came up with 6 natural language test smells. In the following, we define the smells using our smell definition template (see Subsection 7.1.3) and group them based on the manifestation level of their quality factors as *Test Suite Structure Smells*, *Test Case Structure Smells*, and *Textual Smells*:

7.2.2 Test Suite Structure Smells

The category *Test Suite Structure Smells* summarizes natural language test smells indicating quality problems that are affecting not only the part of a test suite it is manifested in. Instead, the underlying problem affects the whole structure of a test suite.

Smell – *Hard-Coded Test Data*:

For most test cases, concrete input and output values are required for execution. A common way to persist this test data is to embed it in the text of the test descriptions as they are needed. However, although embedding test data directly into the text of test cases makes them self-contained, it also makes them more labor intensive to maintain when test data has to be adapted. To avoid unnecessary duplication of maintenance, concrete values should be avoided when writing test descriptions. Instead, they should be extracted to dedicated test data repositories storing typical sets of test data which can be referenced from within the text of the test cases. Additionally, concrete values can be avoided by defining test data relatively. We define the smell *Hard-Coded Test Data* as follows:

Smell Name:	Hard-Coded Test Data
Smell Category:	Test Suite Structure Smells
Quality Factor:	Test data is embedded in the text of test steps.
Affected Activities	Maintenance
Quality Problems:	If test data has to be changed, it is difficult to find out which test cases and test steps have to be adapted.
Example:	*Text:* *"Load a contract that has been signed before '01/Jan/2014'."* *Quality Defect:* Absolute dates are outdated soon and have to be adapted frequently. *Refactoring:* Replace absolute dates with relative dates: *"Load a contract that has been signed a maximum of five months ago."*

7.2.3 Test Case Structure Smells

The category *Test Case Structure Smells* summarizes natural language test smells indicating quality problems that are affecting not only the part of a test suite it is manifested in. Instead, the underlying problem affects the whole structure of a test case.

Smell – *Branches in Test Flow*:

To make the results of test runs interpretable and comparable, test procedures have to be predefined and deterministically executable. Therefore, test cases should be free of branching logic, such as optional parts or indeterministic case differentiation. However, natural language makes it easy to overlook branching logic phrased in the text of test procedures. We define the smell *Branches in Test Flow* as follows:

Smell Name:	Branches in Test Flow
Smell Category:	Test Case Structure Smells
Quality Factor:	The test flow contains branches or alternative flows that manifest themselves through conditions in the test steps' text.
Affected Activities	Execution, Maintenance
Quality Problems:	*Execution:* In multiple test runs, conditions may be satisfied differently. Therefore, results of test runs are not comparable. *Maintenance:* The test step's intention is difficult to grasp. Therefore, it is difficult to find out which parts to adapt.
Example:	*Text:* *"If the field 'selected customer' is empty, use the search function to select an existing customer."* *Quality Defect:* By looking at a test's result, it is not clear whether the tester had to search and select a new customer or not. *Refactoring:* Make sure that the test procedure is predefined: *"Step 1: Verify that the field 'selected customer' is not set."* *"Step 2: Use the search function to select an existing customer."*

Smell – *Merged Test Steps*:

Writing test cases in natural language allows embedding multiple actions and checks in one piece of text. When several independent tasks are merged to one test step, it is difficult to gasp the actual intention of the test step. Furthermore, it makes test results difficult to interpret since, if a test step failed, it is not clear which part of a test step caused a test case to fail. We define the smell *Merged Test Steps* as follows:

Smell Name:	Merged Test Steps
Smell Category:	Test Case Structure Smells
Quality Factor:	A test step comprises several tasks. Although these tasks are independently from each other, they are combined to one test step.
Affected Activities	Execution, Maintenance
Quality Problems:	*Execution:* Reasons for failed test cases are difficult to localize since it is not clear which part of a test step failed. *Maintenance:* The test step's intention is difficult to grasp. Therefore, it is difficult to find out which parts to adapt.
Example:	*Text:* *"Load the detail view of the current transaction. Verify that all fields are filled. Then open the master data dialog and open the contract validation settings. Deactivate all validation setting options. Close the validation settings dialog. Afterwards, verify that ..."* *Quality Defect:* By looking at a result of a failed test run, it is not clear why it failed. *Refactoring:* Break up the test step into several substeps: *"Step 1: Load the detail view of the current transaction and verify that all fields are filled."* *"Step 2: Open the master data dialog. Then open contract validation settings."* *"Step 3: ..."*

7.2.4 Textual Smells

The category *Textual Smells* summarizes natural language test smells indicating quality problems that are affecting just the parts of a test suite where they are manifested.

Smell – *Complicated or Bloated Phrases*:

Natural language allows describing even simple facts in complicated ways which are difficult to understand. Furthermore, test steps are often overloaded with information that is redundant or not necessarily needed, such as rationales or side information that do not help to understand or execute the test case. This makes it difficult to grasp the initial idea of a test cases and to understand how to execute it. Furthermore, it blows up test artifacts and therefore increases maintenance effort unnecessarily. We define the smell *Complicated or Bloated Phrases* as follows:

Smell Name:	Complicated or Bloated Phrases
Smell Category:	Textual Smells
Quality Factor:	A test step is complicated phrased or bloated with unnecessary information and therefore difficult to understand.
Affected Activities	Execution, Maintenance
Quality Problems:	*Execution:* Testers have problems to understand what to do. *Maintenance:* The test step's intention is difficult to grasp. Therefore, it is difficult to find out which parts to adapt.
Example:	*Text: "To verify the case in which the service desk employee cannot perform the transaction directly while being on the telephone since other jobs have to be done first which may take a longer time, the service desk employee ends the current telephone call and saves the current interaction so he can continue it later."* *Quality Defect:* The sentence is difficult to understand since it has many nested subclauses and contains irrelevant information. *Refactoring:* Remove rationales from test descriptions and make sentences easy to understand: *"End the telephone call. Save the current transaction."*

Smell – *Ambiguous Phrases*:

Using natural language, test descriptions can be written in ambiguous ways, leaving room for multiple ways of interpretation. This leads to different expectations by the reader and therefore may cause different test results, if a test case is executed by different persons. We define the smell *Ambiguous Phrases* as follows:

Smell Name:	Ambiguous Phrases
Smell Category:	Textual Smells
Quality Factor:	The description of a test step is not clear to understand. It can be interpreted in different ways.
Affected Activities	Execution
Quality Problems:	Different testers read different things from the text and form different expectations. Therefore, results of test runs are not comparable.
Example:	*Text: "Check whether the results of the calculation are plausible."* *Quality Defect:* It is not specified how to verify the plausibility of results. Different testers may use different criteria. *Refactoring:* Clarify the criteria for plausibility: *"Check whether 'result-min' is smaller than 'result-max'."*

Smell – *Test Clones*:

Many test cases contain common parts – so called *test clones* (see Chapters 4 *Clones in Manual System Tests* and 6 *Test Refactoring Using Grammar Inference*). Test cases that share large common parts are difficult to distinguish since it is not easy to understand their initial intention. Furthermore, effort for adapting test cases to changes are unnecessarily duplicated by clones. We define the smell *Test Clones* as follows:

Smell Name:	Test Clones
Smell Category:	Textual Smells
Quality Factor:	A text passage of a test step (of considerable length) exists several times within one or several test cases.
Affected Activities	Maintenance
Quality Problems:	*Maintenance 1:* Test sequences that are similar but not identical are not easy to distinguish. It is not easy to grasp a test's intention. *Maintenance 2:* The effort to maintain duplicated parts of test cases increases. Furthermore, it is difficult to find out which parts to adapt.
Example:	*Refactoring:* Extract recurring text passages to reuse components, such as, templates.

7.3 Automated Detection of Natural Language Test Smells

In the following, we present our heuristic approaches to automatically detect natural language test smells in natural language. Similar to our smells presented in the previous Section, the following detection techniques result from our industrial experiences of the last five years. We first introduce basic methods to preprocess natural language text. Afterwards, generic detection techniques are introduced, which are independent form concrete natural language test smells. Finally, for each smell from Section 7.2, a concrete detection approach is presented.

Goal of detection techniques is to identify occurrences of quality factors in test artifacts and reporting them as smell findings. Each smell finding must equipped with the exact location where the quality factor is located in the analyzed text.

However, instances of quality factors are often difficult to find: Natural language enables to write ambiguous text and provides many ways to express the same thing. Therefore, techniques to identify smells in natural language text may not be absolutely precise or costly to perform (such as manual reviews).

7.3.1 Natural Language Text Processing

In the following section, we summarize existing techniques to retrieve information from natural language text. Concretely, we apply the techniques *Text Segmentation* and *Part-Of-Speech Tagging* to preprocess the textual description of natural language test cases. The retrieved information will then form the data basis for our generic detection techniques.

Text Segmentation

Text Segmentation is a standard technique to divide natural language text into sequences of meaningful tokens [Jurafsky and Martin, 2000]. To support our universal detection techniques, we divide the input text into sequences of *Sentences* and *Words*. Figure 7.4 shows an example of a segmented sentence.

```
Input:  'Repeat Step 4 and Step 5 as long as you want.'
Output: 'Repeat', 'Step', '4', 'and', 'Step', '5', 'as', 'long', 'as', 'you', 'want', '.'
```

Figure 7.4: Example of Text Segmentation.

Part-Of-Speech Tagging

Another standard technique of natural language processing is *Part-Of-Speech Tagging*, a technique to markup tokens with their grammatical roles in a sentence [Jurafsky and Martin, 2000; Schmid and Laws, 2008; Toutanova et al., 2003]. Each token is annotated with its part-of-speech, such as *noun*, *adjective*, *verb*, or *punctuation*. A common annotation format for part-of-speech information are *POS-Tags* defined by the *Penn Treebank Project*[1]. Figure 7.5 shows an example of a sentence annotated with POS-Tags.

```
Input:  'Repeat Step 4 and Step 5 as long as you want.'
Output: 'Repeat'  NN  Noun, singular or mass: bicycle, earthquake, zipper
        'Step'    NN  Noun, singular or mass: bicycle, earthquake, zipper
        '4'       CD  Cardinal number: one, two, twenty-four
        'and'     CC  Coordinating conjunction: and, or, either, if, as,
                           since, once, neither, less
        'Step'    NN  Noun, singular or mass: bicycle, earthquake, zipper
        '5'       CD  Cardinal number: one, two, twenty-four
        'as'      RB  Adverb and negation: easily, sunnily, suddenly,
                           specifically, not
        'long'    RB  Adverb and negation: easily, sunnily, suddenly,
                           specifically, not
        'as'      IN  Preposition/subordinate conjunction: except, inside,
                           across, on, through, beyond, with, without
        'you'     PRP Personal pronoun: everyone, I, he, it, myself
        'want'    VBP Verb, non-3rd ps. sing. present: eat, jump, believe,
                           am (as in 'I am'), are
        '.'       .   Sentence-final punctuation
```

Figure 7.5: Example of Part-Of-Speech Tagging.

[1] http://www.cis.upenn.edu/~treebank/

7.3.2 Generic Detection Techniques

In the following section, we present generic detection techniques. Each of them is able to find occurrences of quality factors in natural language text. However, generic detection techniques are independent form any natural language test smell and are used to create detection techniques for concrete natural language test smell.

Word Lists:

The generic detection technique *Word List* identifies occurrences of single words within a given natural language text. This technique is configured using a *word list* defining the words to identify in a given text. It builds up on the natural language text processing technique Text Segmentation. Figure 7.6 shows an example searching for the words *long* and *want*.

Text: *'Repeat Step 4 and Step 5 as long as you want.'*
Word List: *'long'*, *'want'*
Output: *'Repeat Step 4 and Step 5 as long as you want.'*

Figure 7.6: Example of the generic detection technique Word List.

Text Patterns:

An extension of the generic detection technique Word List is *Text Patterns*, which is able to identify more complex text phrases. This generic detection technique is configured using *patterns* defining sequences of words having certain POS-Tags to identify in a given text. This generic detection technique builds up on the natural language text processing technique Part-Of-Speech Tagging. Figure 7.7 shows an example of this generic detection technique identifying appearances of the word *Step* with an arbitrary POS-Tag but followed by a cardinal number (a random word with POS-Tag *CD*).

Text: *'Repeat Step 4 and Step 5 as long as you want.'*
Pattern: *'Step'*⟨∗⟩ *'*'*⟨CD⟩
Output: *'Repeat Step 4 and Step 5 as long as you want.'*

Figure 7.7: Example of the generic detection technique Text Patterns.

Word Counting:

Another generic detection technique is *Word Counting*, which gives access to the number of words of a given piece of text. It counts just tokens which are real words and ignores other tokes, such as numbers, quotation marks, or punctuation characters. This generic detection technique builds up on the natural language text processing technique Part-Of-Speech Tagging and enables creating customized smell detection techniques that are based on the length of the text.

Clone Detection:

Clone Detection summarizes techniques to identify similar sequences of elements in token streams. We use these techniques as generic detection technique building up on Text Segmentation to find clones in the text of test descriptions. Input for this generic detection technique is the parameter *minimal clone length* defining the minimal number of similar tokens to be considered as clone (cf. Chapter 4 *Clones in Manual System Tests* and 6 *Test Refactoring Using Grammar Inference*).

7.3.3 Detecting Natural Language Test Smells

In the following subsection, we describe how the generic detection technique can be used to detect the smells defined in Subsection 7.2.

Smell *Hard-Coded Test Data*:

To identify test data that is hard-coded in the text of test steps, we use the generic detection technique *Text Patterns*. We search for text in quotation marks and tokens consisting of just numbers and special characters.

Smell *Branches in Test Flow*:

To identify branches of the test flow, we perform a two-fold process combining the techniques *Word List* and *Text Patterns*. First, we search for words that potentially phrase case differentiations, such as, *"if"*, *"whether"*, *"when"*, or *"depending"*. In a second step, we eliminate incorrectly found phrases (*false-positives*) that do not indicate branches and them if they match a list of anti-patterns. For example, sentences starting with *"Check whether, ..."* do not indicate conditions.

Smell *Merged Test Steps*:

To find test steps comprising several tasks, we use a heuristic approach based on the technique *Word Counting*. Based on our experiences, we defined a threshold of 130 real words and report test steps if their text contains more words.

Smell *Complicated or Bloated Phrases*:

To identify test steps that are complicated phrased or bloated up with irrelevant information, we use a heuristic approach that builds up on the generic detection technique *Word Counting*. Similar to the smell Branches in Test Flow, we defined a threshold based on our experiences: We report sentences if they contain more than 45 real words.

Smell *Ambiguous Phrases*:

To identify ambiguous phrases, we perform a two-fold process combining the techniques *Word List* and *Text Patterns*. First, we search for words that are likely ambiguous to interpret, such as, *"should"*, *"most"*, *"any"*, *"more"*, or *"appropriate"*. In a second step, we eliminate incorrectly found phrases (*false-positives*) in which those words have a clear interpretation by filtering out our findings if they match a list of anti-patterns. For example, phrases as *"most recent ..."* or *"more than ⟨NOUN⟩"*.

Smell *Test Clones*:

To identify clones in the text of test descriptions, we use *Clone Detection* techniques. We consider a substring of a test description as clone, if it is at least 10 sentences long and appears at least twice in the test suite. Furthermore, we perform clone detection on the whole text of a test case by stringing together the text of each test step. Thereby, we identify clones that span over several test steps within a test case.

Figure 7.8 summarizes the dependencies between the natural language text processing techniques, the generic detection technique, and the concrete techniques used to detect natural language test smells.

7.4 Evaluation

To analyze the ability of our natural language test smells to identify quality problems in test suites, we apply our approach in an industrial setting.

7.4.1 Study Goal and Research Questions

We define the goal of our study using the goal definition template of Wohlin et al. [2000]:

We analyze	*the concept of natural language test smells*
for the purpose of	*evaluating*
with respect to their	*ability to identify relevant quality defects*
from the viewpoint of	*test engineer*
in the context of	*industrial software projects.*

We define the research questions along our ontology (see Section 7.1). In a first part, we investigate the ability of our detection techniques (Section 7.3.3) to identify actual quality factors. In a second part, we use our detection techniques to investigate the relevance of smell findings. Figure 7.9 aligns both parts of this study with our ontology (Figure 7.1).

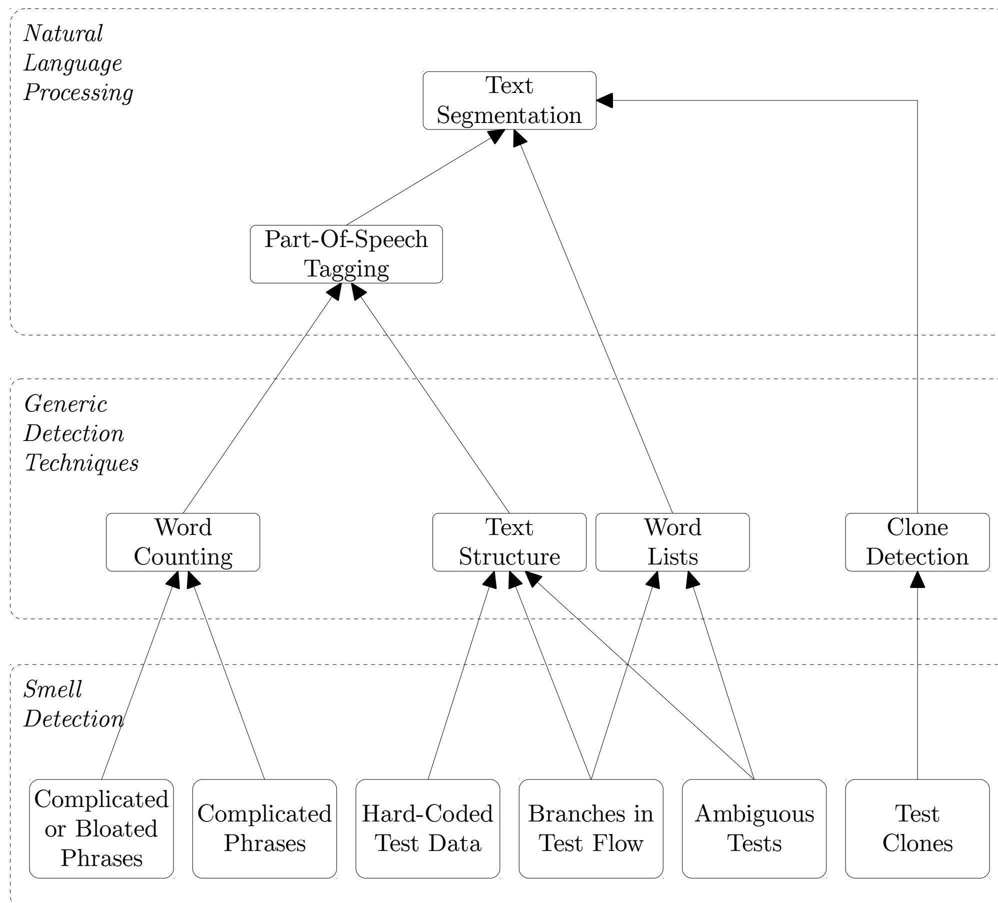


Figure 7.8: Dependencies between the natural language text processing techniques, the generic detection technique, and the concrete techniques used to detect natural language test smells.

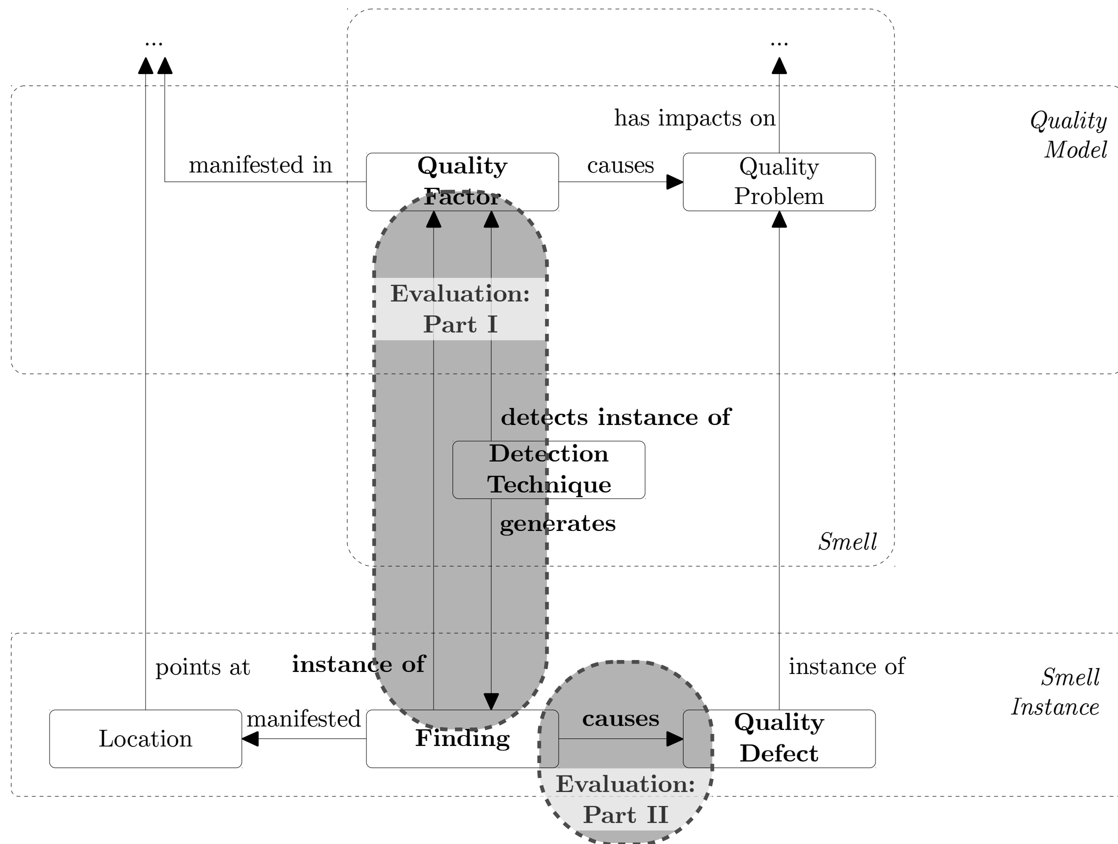


Figure 7.9: Goal of the study aligned with our smell ontology (see Section 7.1).

Part I – Quality of the Detection Techniques

In the first part of this study, we evaluate the quality of the smell detection techniques we presented in Section 7.3.3.

RQ 1: How precise are our detection techniques?

Many natural language test smells are difficult to detect automatically which makes detection techniques imprecise. Since the detection techniques we introduced in Section 7.3.3 are the basis for the rest of this study, their quality is of crucial importance. This research question investigates the ability of our smell detection techniques to identify correct smell findings.

Part II – Relevance of Smell Detection Results

In the second part of this study, we investigate the relevance of smell detection results. We approximate the relevance of the results by the number and severity of smell findings.

RQ 2: How many correct findings are identified by our detection techniques?

In order to indicate whether the effort of performing smell detection pays off, the number of revealed relevant smell findings is significant. This research question investigates how many correct smell findings of each natural language test smell are identified by our smell detection techniques.

RQ 3: How severe are uncovered quality problems?

To detect the relevance of the quality problems that can be uncovered with natural language test smell, this research question investigates the importance of smell findings that are found.

7.4.2 Study Objects and Case Study Context

Our study objects are manual system tests of nine projects from Munich Re and Airbus Defense & Space (see Section 1.5 *Case Study Partners*).

The systems of the analyzed test suites are business information systems having web front-ends, Windows fat client interfaces, or are SAP systems. All test cases are written all in English, by different teams (internal employees and external suppliers), and are testing different functionalities. This increases the generalizability of the study results. The analyzed test cases include regression tests, tests for planned features, and tests of change requests. All systems as well as the corresponding test suites are in productive use. For non-disclosure reasons we named the test suites A to I. Table 7.9 summarizes the sizes of the test suites. The number of test cases differs from 72 (test suite A) to more than 1,000 (test suites B, H, and I) per test suite. In total, we analyzed 19,000 sentences of natural language test description containing more than 1,8 million words.

Table 7.9: Study objects.

		All Tests		
	#Tests	#Test Steps	#Sentences	#Words
Test Suite A	72	628	2,611	30,512
Test Suite B	1,786	17,983	68,808	586,454
Test Suite C	298	2,409	7,353	65,160
Test Suite D	627	8,179	28,257	378,405
Test Suite E	152	4,896	17,732	162,127
Test Suite F	115	3,788	9,452	82,902
Test Suite G	114	4,771	5,263	39,661
Test Suite H	1,104	13,151	31,598	264,483
Test Suite I	1,165	7,132	23,138	223,073
Total	5,433	62,937	194,212	1,832,777

7.4.3 Data Collection

To answer the research question, we perform automated smell detection using our smell detection techniques on all nine study objects.

RQ 1: How precise are our detection techniques?

To determine whether the smell detection techniques presented in Section 7.3.3 are able to identify quality problems, we are manually assessing smell findings from our study objects. To quantify the quality of our detection techniques, we use the metric *Precision* defined as follows:

$$Precision = \frac{\text{no. of True Positives}}{\text{no. of All Smell Findings}}$$

RQ 2: How many correct findings are identified by our detection techniques?

To determine how many correct Smell Findings are uncovered, we manually assess the smell findings that are produced by our smell detection techniques. We use the metric *Number of True Positives* to measure the number of correct smell findings.

RQ 3: How severe are uncovered quality problems?

To investigate the severity of the quality problems that are revealed by smell findings, we perform expert interviews. In this study, we approximate the severity of quality problems by the consequences test engineers draw from smell findings. We use a multi-level classification schema to let experts rate the severity of smell findings.

7.4.4 Study Execution

In the following, we describe how we performed the automated smell detection, the manual assessment of smell findings, and the expert interviews. Figure 7.10 summarizes which study objects have used for which data collection technique.

Table 7.10: Study objects used for data collection techniques.

	Test Suite								
Data Collection Technique	A	B	C	D	E	F	G	H	I
Automated Smell Detection (RQ 1,2,3)	×	×	×	×	×	×	×	×	×
Manual Assessment of Findings (RQ 2)	×	×	×	×	×	×	×	×	×
Expert Interviews (RQ 3)							×	×	×

Automated smell detection

We created prototypical tooling in that we implemented our smell detection techniques (see Section 7.3.3). Our tooling consists of three consecutive phases:

Phase 1 – Reading test suite: As a first step, the tool reads the test suite from its original format, such as word files or database exports. The reader transfers the test suite into our basic artifact model consisting of *Test Cases* having *Test Steps*. Each test step has two fields: A *Step Description* and an *Expected Result*, both described in form of natural language text (see Figure 7.10).

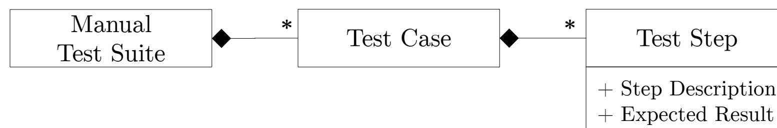


Figure 7.10: The artifact model of our smell detection tooling.

Phase 2 – Smell detector pipeline: As a second step, the tool processes an extensible pipeline of smell detectors. Each smell detector gets access to artifact model and is able to annotate smell findings to any part of the test suite.

Phase 3 – Visualization: As a third step, the tool generates an HTML dashboard that allows browsing through content of artifact model and the annotated smell findings.

The case study was performed on a PC with Linux operating system equipped with an Intel Dual Core 2.9 GHz CPU and 8 GB of RAM. The time for the smell detection was between 30 seconds up to a 5 minutes per test suite.

Manual assessment of findings

To assess the quality of the smell detection techniques, two researchers manually assessed smell findings of our test smells. Both researchers have been working in the field of system test quality for several years and are experienced with natural language text processing. We selected up to 10 random smell findings per smell from each test suite and classified them either as *true positive* or *false positive* depending on whether the smell finding is an actual instance of the quality factor of a smell. The classification took part in three phases for each test smell:

Phase 1 – Alignment: As a first step, the selected smell findings of one test suite were assessed collaboratively by both researchers to get a common understanding of the smell and its quality factor. Each smell finding was discussed between both researchers and criteria to determine the quality factors were defined.

Phase 2 – Individual Assessment: As a second step, smell findings of another seven test suites have been split up into two parts and have been assessed independently by one researcher each.

Phase 3 – Inter Rater Agreement: As a third and final step, the smell findings of the remaining test suite were assessed by both researchers independently to get data points to calculate the inter rater agreement using the metric *Cohen's kappa* [Cohen, 1960].

In total, we manually assessed 420 smell findings. 50 of which have been used to determine the inter rater agreement (Phase 3). The remaining 370 smell findings have been used to calculate the precision of the smell detection techniques (Phase 1 and 2). Table 7.11 shows the concrete number of smell findings that have been manually assessed.

Table 7.11: Number of smell findings that have been manually assessed.

	Sample Size (no. smell findings)		
Smell	Precision	Inter Rater Agreement	Total
Hard-Coded Test Data	79	10	89
Branches in Test Flow	80	10	90
Merged Test Steps	61	10	71
Complicated or Bloated Phrases	71	10	81
Ambiguous Phrases	79	10	89
Test Clones	-	-	-
Sum	370	50	420

Expert interviews

Goal of the expert interview was to identify the severity of correct smell findings. Therefore, we discussed randomly selected smell findings of test smells with the test engineers of the corresponding test suites. To be independent of the precision of the used smell detection techniques (see RQ 1), we filtered the random selection and removed false-positive smell findings that do not satisfy the smell's quality factor. We asked for the consequences test engineers draw from the smell findings. Concretely, the test engineer was asked the following questions for each presented finding:

(Q1) *What consequences do you draw from this smell finding?*

The test engineer was asked to answer the question by rating the smell finding using our four-staged classification schema (see Figure 7.11) and to give a short rational for the decision. We performed one interview for each of the test suites G, H, and I. In each of the interviews, we talked to 1 – 2 test engineers of the corresponding test suite for about 60 – 90 minutes. Because of time constraints of the interviews and since not every smell occurred in each test suite, it was not possible to discuss smell findings of each smell in each interview. In total, 109 smell findings have been assessed by test engineers of all three test suites. Table 7.12 shows the concrete number of smell findings that have been discussed in the interviews.

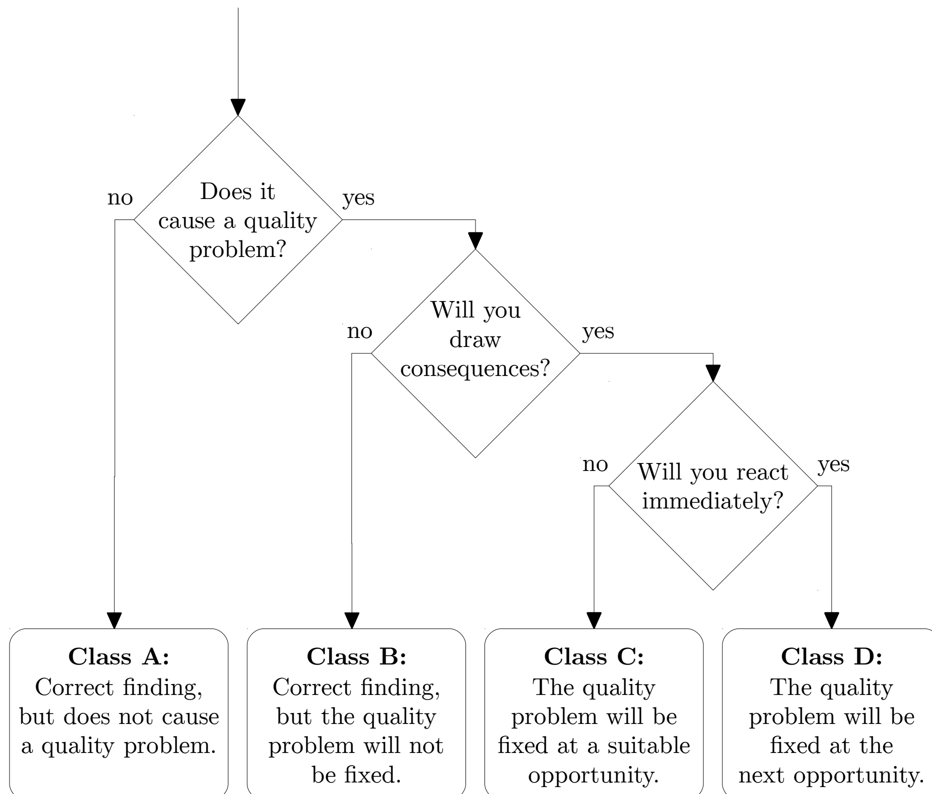


Figure 7.11: Classification schema for manual assessment of the severity of findings.

Table 7.12: Number of smell findings that have been assessed in the expert interviews.

	Test Suite			
Smell	G	H	I	Total
Hard-Coded Test Data	-	-	12	12
Branches in Test Flow	3	8	12	23
Merged Test Steps	3	5	13	21
Complicated or Bloated Phrases	-	-	10	10
Ambiguous Phrases	7	14	7	28
Test Clones	-	-	15	15
Sum	13	27	69	109

7.4.5 Results

We present the results along the research questions:

RQ 1: How precise are our detection techniques?

Our detection technique to uncover Test Clones builds up on existing clone detection algorithms from the area of source code clone detection. Since those algorithms are very accurate, we assume that the Smell Test Clones has a precision of 100%. Therefore, we did not manually assess the precision of any Test Clones smell findings.

From the other smells, we manually assessed 370 smell findings (see Table 7.11). The detected precision of our smell detection techniques was between 56.3% (Smell Complicated or Bloated Phrases) and 97.5% (Smell Hard-Coded Test Data) with an average of 73.8% (calculated over all assessed smell findings). The inter rater agreement over all smell findings that have been assessed by both researchers was $k = 0.76$ (values between 0.61 and 0.80 are considered as *substantial* [Altman, 1990; Byrt, 1996; Fleiss et al., 2013; Landis and Koch, 1977]). Table 7.13 summarizes the results of the manual assessment.

Table 7.13: Precision of smell detection techniques (summary of all test suites).

Smell	Precision
Hard-Coded Test Data	97.5%
Branches in Test Flow	82.5%
Merged Test Steps	65.6%
Complicated or Bloated Phrases	56.3%
Ambiguous Phrases	63.3%
Test Clones	(100%)
Overall (w/o Test Clones)	73.8%

RQ 2: How many correct findings are identified by our detection techniques?

We extrapolate the number of correctly identified smell findings by correcting the number of all found smell findings with the precision of our detection techniques (see RQ 1). Hence, we assume that 79,424 smell findings have been correctly detected by our smell detection tooling. The number of correct smell findings per Smell ranges from 302 of Smells Merged Test Steps to 62,848 of Smell Hard-Coded Test Data. Table 7.14 shows the number of correct smell findings for each Smell and each test suite.

Table 7.14: Number of correct findings of all test suites (extrapolation).

	Test Suite				
Smell	A	B	C	D	E
Hard-Coded Test Data	1,674	17,843	1,749	19,732	1,788
Branches in Test Flow	28	1,286	162	794	100
Merged Test Steps	5	58	1	171	0
Complicated or Bloated Phrases	7	63	3	340	23
Ambiguous Phrases	74	1,947	225	1,593	235
Test Clones	59	2,844	61	1,250	677
Sum	1,847	24,041	2,201	23,879	2,823

	Test Suite				
Smell	F	G	H	I	Total
Hard-Coded Test Data	2,209	1,912	6,569	9,371	62,848
Branches in Test Flow	78	87	360	489	3,383
Merged Test Steps	1	10	40	18	302
Complicated or Bloated Phrases	38	3	62	17	557
Ambiguous Phrases	133	74	731	641	5,653
Test Clones	107	65	995	623	6,681
Sum	1,565	2,268	8,757	11,159	79,424

RQ 3: How severe are uncovered quality problems?

Except for the smell Hard-Coded Test Data, the majority of the Smell Findings have been classified as leading to quality problems (Classes B, C, and D). In average, ~75% of the smell findings have been classified as to be fixed at suitable or even next opportunity (Classes C and D). In the case of Merged Test Steps and Test Clones, even more than 50% smell findings are to be fixed at next opportunity (Class D). Table 7.15 shows the classification results for each smell.

Table 7.15: Severity of smell findings.

Smell	Class A	Class B	Class C	Class D
Hard-Coded Test Data	50%	25%	8.3%	16.7%
Branches in Test Flow	26.1%	4.3%	47.8%	21.7%
Merged Test Steps	-	-	47.6%	52.4%
Complicated or Bloated Phrases	-	20%	30%	50%
Ambiguous Phrases	25%	7.1%	60.7%	7.1%
Test Clones	-	-	40%	60%
Overall	17.4%	7.3%	44%	31.2%

7.4.6 Threats to Validity

In this section, we discuss threats to the external and internal validity of the study and describe how we mitigated them:

External Validity:

In our study, we analyzed test suites 9 two business information systems of 2 companies from two different application domains. It is possible that test cases are designed differently in other application domains or other companies because they use different tools and processes to create and maintain test cases. To make our results more generalizable, the study has to be repeated using test cases of different application domains created by different organizations.

Internal Validity:

Since the categorization in RQ 1 and RQ 3 has been performed manually, the results might be subjective to personal bias due a metric's subjectivity. For RQ 1, we addressed this risk by starting the assessment for each smell with an alignment phase in that both researchers did together to form a common understanding of the smells and their quality factors. We measured the inter rater agreement to quantify in as much the assessment of both researchers differs. The resulting Cohen's kappa value of $k = 0.76$ considers the agreement of both researchers as *substantial* $(0.61 - 0.80)$. For RQ 3, we were interested in the actual consequences that test engineers draw from smell findings. Since the test engineers that took part in our interviews were responsible for executing and maintaining the corresponding test suite, we consider the results as realistic and significant.

To answer RQ 1, two researchers manually assessed the quality of smell findings. As these researchers are not part of the project team of the study objects, this might bias the results. However, the researchers who conducted the study have worked with Munich Re for several years and are experienced in the field of software testing.

The categorization in RQ 1 and RQ 3 have not been performed on all identified smell findings but just on small samples. Selecting samples can potentially introduce inaccuracy. However, we tried to reduce this issue by selecting the samples randomly.

7.4.7 Interpretation and Discussion

Goal of this study was to evaluate the concept of natural language test smell regarding its ability to identify quality problems in industrial test suites. We split up this goal into two parts:

Part I – Quality of detection techniques

In the first part of the study, we evaluated the quality of the smell detection techniques that have been presented in Section 7.3.3. The precision of the smell detection techniques is a crucial criterion to user acceptance. If smell detection tooling leads to many smell findings without relevance for test engineers, its chances to get accepted decreases easily.

In RQ 1, we analyzed the precision of our detection techniques. Some detection techniques have shown to be quite accurate with a precision >95%, such as Hard-Coded Test Data and Test Clones. However, other smell detection techniques, such as Complicated or Bloated Phrases, have a moderate precision leading to many false positives. However, the implementation or our detection techniques that have been used in this study was a prototypical implementations and still provide many ways for improvement. Therefore, we consider the measured precision as a lower bound.

Part II – Relevance of findings

In the second part of the study, we evaluated the relevance of smell findings. We approximated the relevance in two ways: The number of (correctly) identified smell findings (RQ 2) and their relevance for test engineers (RQ 3).

Although the number of findings per smell differs strongly, our manually assessed samples indicate that our smell detection techniques correctly identified ∼1,500 up to ∼24,000 smell findings per test suite (∼80,000 in total). Based on our interviews, in ∼75% of the cases (severity class C+D), test engineers consider them as relevant enough to fix the identified quality problems.

Although the data of this study is not detailed enough to allow precise predictions, combining the assessment of the test engineers with the number of correct smell findings indicates that our approach revealed quality problems to be fixed in the order of ∼1,000 to ∼19,000 in each test suite.

7.5 Benefit Estimation

In the previous study, we have shown that natural language test smells are able to identify relevant quality defects in industrial software projects. However, so far it is unclear how much benefit will be gained by removing smell findings as well as if and when the effort of removing them will pay off. To address these questions, we present a rough estimation of the benefit of removing smell findings in natural language test cases. Goal of this calculation is not to present a precise cost model but to learn in what settings the usage of natural language test smells pays off and how much testing costs can roughly be saved.

7.5.1 Benefit Estimation Model

To estimate the overall benefit of our approach, we rely on our relative benefit estimation model that we introduced in Chapter 5 *Choosing Execution Modes* (see Section 5.5.1 *Benefit Estimation Model*). This model is based on the idea, that the overall costs of system testing are composed of the costs of each testing activity of system testing (see Figure 7.12). Furthermore, the model defines the relative sizes of each activity within the whole calculation. Knowing these proportions between the system testing activities allows calculating how the overall system testing costs develop when the costs for single testing activities change.

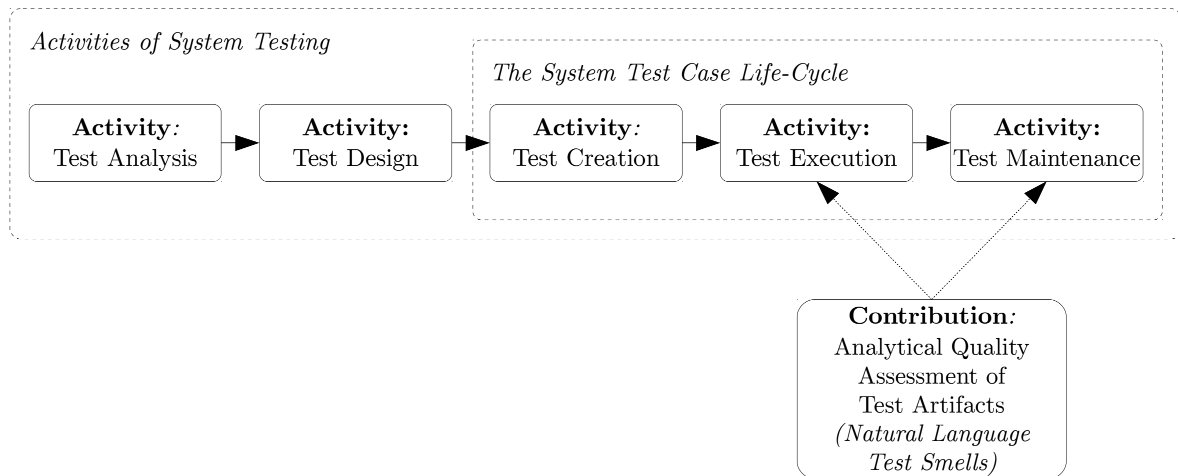


Figure 7.12: System testing activities that are addressed by the contribution of this chapter.

The approach that we presented in this chapter addresses the activities of test execution and test maintenance (see Figure 7.12). In the following sections, we estimate the benefit of our approach for these two activities. Furthermore, to estimate how much costs can be saved in total, we use our relative benefit estimation model to extrapolate the benefit for the overall system testing costs.

7.5.2 Assumptions

To ease the estimation, we make the following assumptions:

Just one smell per test step: To simplify the estimation of execution and maintenance benefits of smell removal, we assume that there is a maximum of one smell finding per test step. Thereby, we can estimate the effort and benefit of removing findings on a per test step base.

Independent benefit: We assume that removing findings from one test step does not influence the execution and maintenance of other test steps. Furthermore, removing a finding does not influence the benefit that is gained from removing other findings.

Findings of same smell have similar removal costs and benefit: To avoid assessing costs and benefits of each finding individually, we assume that all findings of the same smell require the same effort to be removed and provide the same execution and maintenance benefit once they are removed. We just estimate the costs and benefit for each smell once.

Each test step is executed exactly once: We assume that each test step is executed exactly once per test run. Thereby, we do not have to consider test cases that are executed several times or parts of test cases (for example, individual test steps) that are called by other test cases and are therefore executed several times.

No setup costs: Lastly, we ignore the costs for performing automated smell detection, such as installation, configuration and execution costs.

7.5.3 Calculation of Benefit for the Activities *Test Execution* and *Test Maintenance*

We calculate the benefit of our approach for the activities test execution and test maintenance as follows: The benefit is approximated by comparing the benefit that is gained from removing findings with the costs to remove them. Concretely, we subtract the removal costs of findings from their execution and maintenance benefits:

	Execution benefit for one execution × No. of planned test runs
+	Maintenance benefit for one maintenance task × No. of expected maintenance tasks
−	Costs of findings removal
=	Benefit of findings removal

To estimate the overall benefit that is gained from removing all smell findings, we sum up the benefit from each individual smell finding:

$$\sum_{smell \in Smells} \text{No. of findings (per smell)} \times \text{Benefit of removing one finding (per smell)}$$

= Benefit of removing all findings

To estimate the extent to that the activity costs decrease by applying our approach, we furthermore need an estimation of the overall test execution and maintenance costs of a test suite. We calculate these values as follows:

	Avg. execution effort per test step
×	Number of test steps
×	Number of planned test runs
	Execution costs without our approach (over time)

	Avg. maintenance effort per test step
×	Number of test steps
×	Number of expected maintenance tasks
	Maintenance costs without our approach (over time)

7.5.4 Data Elicitation

We apply our cost estimation model to the study objects of the previous study (Section 7.4 *Evaluation*). However, since for this study not all necessary data is available, we substitute the missing data with data from our previous study (Chapter 5 *Choosing Execution Modes*) and with own estimations.

Number of Findings

To estimate the number of relevant findings, we multiply the number of correctly identified findings from Section 7.4 *Evaluation* with the relevance that has been determined in the same study. Table 7.16 summarizes the calculation.

Table 7.16: Number of correct and relevant findings (summary of all study objects of the previous case study).

Smell	Correct Findings	Relevance (Classes C+D)	Relevant Findings
Hard-Coded Test Data	62,848	25.0%	17,712
Branches in Test Flow	3,383	69.5%	2,351
Merged Test Steps	302	100.0%	302
Complicated or Bloated Phrases	557	80.0%	446
Ambiguous Phrases	5,653	67.8%	3,833
Test Clones	6,681	100.0%	6,681
Sum	79,424		29,325

Execution Benefit of Finding Removal

For simplification, we assume that all findings of the same smell provide the same benefit for test execution when they are removed (see Section 7.5.2 *Assumptions*). To estimate the execution benefit for findings of a certain smell, we assign each smell to an *execution impact class*. Each class represents a certain type of execution benefit that is gained by removing smell findings. Furthermore, each class is assigned to a speedup factor by which the execution of a test step is improved once a smell has been removed from a test step. Tables 7.17 and 7.18 summarize the execution impact classes and our assignments to natural language test smells.

Table 7.17: Classes of smell impacts on test execution.

Execution Impact Class	Execution Speedup	Description
None	0%	No impact on test execution.
Comprehension by Tester (low)	15%	Tester understands test step easier and needs less time to clarify questions.
Comprehension by Tester (high)	30%	Tester understands test step easier without the need to clarify questions.

Table 7.18: Classification of smell impact on test execution.

Smell	Execution Impact Class
Hard-Coded Test Data	None
Branches in Test Flow	Comprehension by Tester (low)
Merged Test Steps	Comprehension by Tester (high)
Complicated or Bloated Phrases	Comprehension by Tester (high)
Ambiguous Phrases	Comprehension by Tester (high)
Test Clones	None

To estimate the actual execution time that is saved by removing a smell finding, we build up on concrete values from our case study presented in Chapter 5 *Choosing Execution Modes*. We take the average execution time of a test step of the case study and calculate the actual time that is saved based on the speedup factor of the assigned execution impact class.

Maintenance Benefit of Finding Removal

Similar to the execution benefit of removed findings, we assume that findings of the same smell provide the same benefit for maintenance tasks when they are removed (see Section 7.5.2 *Assumptions*). We assign each smell to a *maintenance impact class*, which represent classes of benefits that are gained by removing findings. Similar to execution impact classes, we assign to each maintenance impact class a maintenance speedup factor by which we assume that maintenance tasks are are performed faster once smell findings are removed. Tables 7.19 and 7.20 summarize the maintenance impact classes and the assignments to natural language test smells.

Table 7.19: Classes of smell impacts on test maintenance.

Execution Impact Class	Maintenance Speedup	Description
None	0%	No impact on test maintenance
Comprehension by Maintainer (low)	20%	Maintainer understands test case easier and needs less time to clarify questions.
Comprehension by Maintainer (high)	40%	Maintainer understands test case easier without the need to clarify questions.
Global Maintenance Improvement	50%	Maintenance is performed easier, since changes are locally and limited to a single test case.

Table 7.20: Classification of smell impact on test maintenance.

Smell	Maintenance Impact Class
Hard-Coded Test Data	Global Maintenance Improvement
Branches in Test Flow	Comprehension by Maintainer (low)
Merged Test Steps	Comprehension by Maintainer (high)
Complicated or Bloated Phrases	Comprehension by Maintainer (high)
Ambiguous Phrases	Comprehension by Maintainer (high)
Test Clones	Global Maintenance Improvement

To estimate the actual maintenance time that is saved by removing a smell finding, we build up on concrete values that we elicited in the case study from Chapter 5 *Choosing Execution Modes*. We take the case study's average maintenance time for test steps for one maintenance task and use the maintenance speedup factor to calculate the actual maintenance time that is saved.

Costs of Finding Removal

To ease the approximation of finding removal effort, we assume that findings of the same smell cause similar effort to be removed (see Section 7.5.2 *Assumptions*). We introduce *removal effort classes* that define classes of efforts to remove findings. Since we lack concrete values of efforts to remove smells from test cases, we define them relatively to the effort of maintenance tasks. Each class has a factor that defines the effort to remove smell findings of that class relatively to the effort of general maintenance tasks. Tables 7.21 and 7.22 summarize the remval cost classes and the assignments to natural language test smells.

Table 7.21: Classes of smell removal effort.

Removal Effort Class	Removal Effort (relative to maintenance)	Description
None	0%	No removal costs.
Quick Fix	25%	Finding can be removed by making a small, local change in a test step.
Scope: Test Step	50%	Finding can be removed by changing a test step
Scope: Test Case	75%	Finding can be removed by changing several test steps within the same test case.
Scope: Test Suite	125%	Finding can be removed by changing several test cases within the test suite.

Table 7.22: Classification of smell removal effort.

Smell	Removal Effort Class
Hard-Coded Test Data	Quick Fix
Branches in Test Flow	Quick Fix
Merged Test Steps	Scope: Test Case
Complicated or Bloated Phrases	Scope: Test Step
Ambiguous Phrases	Quick Fix
Test Clones	Scope: Test Suite

Similar to the execution and maintenance benefit, we build up on concrete values of the case study of Chapter 5 *Choosing Execution Modes*. We take a test step's average maintenance effort for one maintenance task and use the factor of the removal effort class to calculate actual values for removing single smell findings.

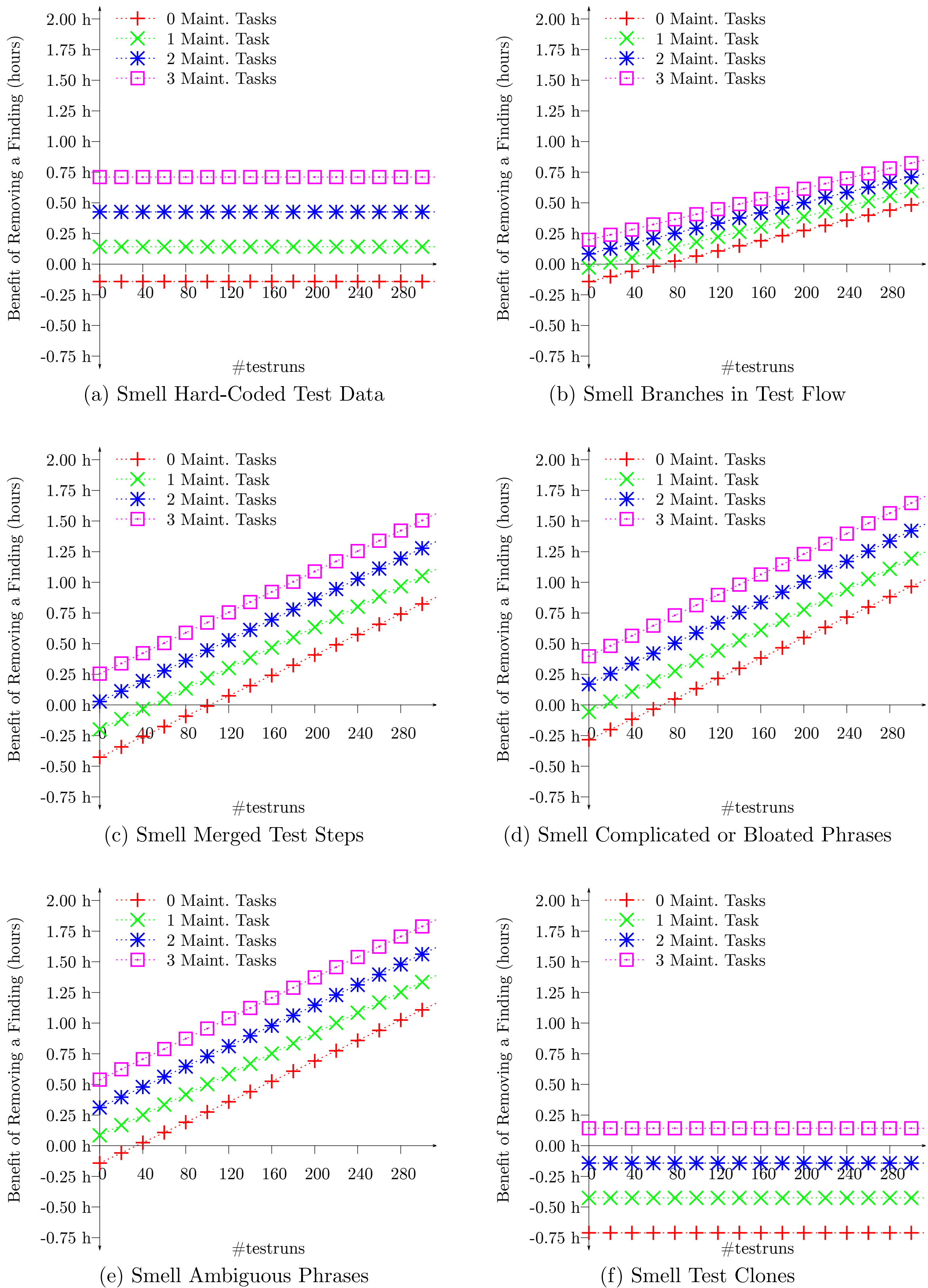
7.5.5 Quantitative Benefit

Our approach addresses the activities of test execution and test maintenance (see Figure 7.12). To assess to what extent the overall system testing costs are reduced by our approach, we first calculate the benefit of removing individual smell findings. From that, we calculate how much the activities of test execution and test maintenance benefit from removing those findings from our study objects. Afterwards, we use our relative benefit estimation model to calculate the overall cost benefit for system testing that is gained by our approach.

Benefit of Single Smell Findings

As a first step towards a benefit analysis of natural language test smells, we quantify the benefit of removing single smell findings. Figure 7.13 gives an overview of the time savings that are gained from removing a single smell finding. In each subfigure, the number of planned test runs and the number of expected maintenance tasks have been used as independent variables: The x-axis shows different numbers of planned test runs. Each subfigure contains several data series for different numbers of expected maintenance tasks. The y-axis shows the time that is saved by removing a single smell finding.

Results: From this data, we can get the following information about the benefit of smell removal: First, for most smells, the benefit increases the more often a test suite is executed. This is because most smells negatively affect the activity of test execution (except for smell Hard-Coded Test Data and smell Test Clones). Second, the more often a test suite is maintained, the larger is the benefit of removing smell findings. Similar to the previous observation, all of our smells negatively affect the activity of test maintenance. Third, removing smell findings does not pay off in each situation. Removing findings of the smell Hard-Coded Test Data and Test Clones, for example, just pays of after one resp. three maintenance tasks. For test suites that will not be maintained often, these smells will not save testing effort. Furthermore, removing most smell findings will not pay off before 40 to 100 test runs (if they are not maintained at all).



(a) Smell Hard-Coded Test Data

(b) Smell Branches in Test Flow

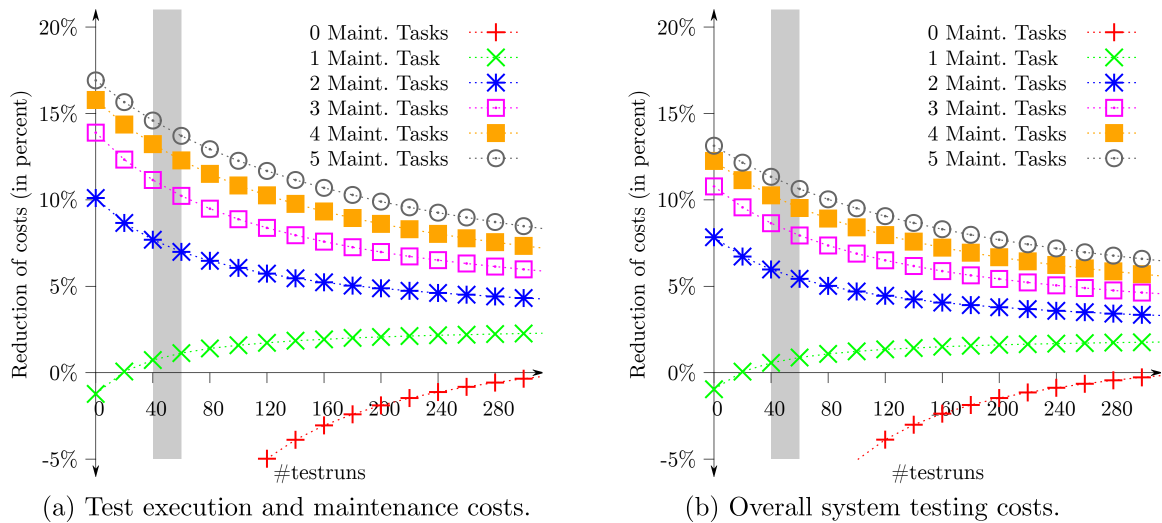
(c) Smell Merged Test Steps

(d) Smell Complicated or Bloated Phrases

(e) Smell Ambiguous Phrases

(f) Smell Test Clones

Figure 7.13: Benefit of removing a single smell finding.



(a) Test execution and maintenance costs.

(b) Overall system testing costs.

Figure 7.14: System testing costs saved by our approach.

Reduction of Test Execution and Maintenance Costs

Figure 7.14a shows the relative reduction of the costs for the activities test execution and maintenance. In our calculation, the number of planned test runs and the number of expected maintenance tasks have been used as independent variables (similar as in the figures before). The x-axis of the figure represents different values of planned numbers of test runs, whereas the y-axis shows the percentage of execution and maintenance costs that will be saved by removing all smell findings (the costs for removing smell findings has been considered in the calculation). Furthermore, each data series represents a different value for the number of expected maintenance tasks.

Results: In our case study, the natural language test smells have their major impact regarding the test maintenance costs. Most smell findings are instances of smells that primarily (or even only) affect the activity maintenance, for example, the smells *Hard-Coded Test Data* or *Test Clones* (see Table 7.16 on page 139). Furthermore, the cost saving effect of removing smell findings is generally higher for the activity maintenance than for the activity test execution (see our classification in Tables 7.18 and 7.20 on pages 140 and 141).

This effect is also visible in our calculation. The more often the test suite will be maintained, the higher will be the costs saving (visible in the difference between the data series). However, the more often the test suite is executed, the lower is the impact of the high maintenance savings (development on the x-axis). Since the execution benefit of our smells is generally lower compared with the maintenance benefit, the relative reduction of costs (in percent) will shrink the more often the test suite is executed despite the absolute cost saving is increasing.

Reduction of Overall System Testing Costs

We used our relative benefit estimation mode to extrapolate the overall system testing costs that are saved by removing smell findings (see Figure 7.14b). Similar to the previous figure, the numbers of planned test runs and expected maintenance tasks are used as independent

variables (x-axis and individual data series). The y-axis shows the relative improvement of the overall system testing costs.

Results: Based on our calculation, we can get the following information for the test suites of our case study. Identifying and removing smell findings will not pay off until the test suite is executed or is subject to maintenance often. The break-even point is at ∼20 test runs if the test suite is maintained once or at ∼300 planned test executions if the test suite is never subject to maintenance at all. Applying the context of our case study from Chapter 5 *Choosing Execution Modes* (40 – 60 test runs, 1 maintenance tasks), our approach is just above zero. However, if the test suite is subject to maintenance more often (e.g., 2 to 5 times), ∼10% of the overall system testing costs could be saved by removing smell findings.

7.5.6 Qualitative Benefit

In addition to the quantitative results, we identified qualitative benefits of removing natural language test smells in test suites:

Same testing in less time: Many of our natural language test smell findings identify quality defects that are slowing down the execution of test cases, for example, because testers have to clarify ambiguities in test descriptions. However, a decrease of execution time does not only lead to reduced testing costs, but also enables quicker testing of all test cases leading to faster development cycles.

More testing in the same time: Faster test execution can also be used to (create and) execute more test cases in the same time. This leads to more effective testing, for example, by increasing test coverage.

Ease of maintenance: Furthermore, all of our natural language test smells findings points at challenges in maintaining test cases. Removing these findings leads to test cases that are easier to maintain. This does not only reduce maintenance costs but also leads to test suites that are faster to adapt to changes and therefore are earlier ready-to-use.

7.6 Summary

In this chapter, we addressed the challenge of improving the quality of test artifacts. We proposed *Natural Language Test Smells*, an approach to support test engineers in identifying quality problems in test suites that are written in natural language. The proposed approach is purely analytically and complements constructive quality improvement approaches by revealing quality problems to be fixed.

We introduced a smell definition template as a framework to support quality engineers in defining quality problems and smells. Furthermore, we bring together our concept of natural language test smells with a basic quality model which allows to individually select smells that are of importance in certain project contexts.

Based on existing code and unit test smells and on our own industrial experience, we define an initial set of six smells for test cases written in natural language. To detect natural language test smells automatically, we introduced four universal smell detection techniques using static analysis techniques, such as, Natural Language Processing (NLP). Using these universal techniques, we presented concrete ways of detecting our smells.

In a case study, we analyzed the ability of our approach to identify relevant quality problems in industrial test suites. We prototypically implemented our smell detection techniques and applied them to 5,433 test cases from 9 manual test suites taken from two different companies. We performed manual assessments and performed interviews with test engineers to evaluate the quality of our detection techniques and the relevance of the smell findings that have been reported by our tooling. The study revealed that our implementation of smell detection techniques has an average precision of ∼74% leading to ∼80,000 correctly identified quality problems in our study objects. Furthermore, the interviews revealed that in ∼75% of the cases, the corresponding test engineers would fix the identified quality problems at next or at suitable opportunities.

We furthermore performed a coarse benefit estimation to find out if and when the costs for removing smell findings are paid off by the benefit of removed smell findings. For our study objects, it turned out that removing smell findings is starting to have a positive benefit, once the test suite is executed ∼20 times and is subject to maintenance at least once (alternatively, ∼300 planned test executions if the test suite is never subject to maintenance at all). However, if our test suites are maintained more often, ∼10% of the overall system testing costs could be saved by removing smell findings.

Chapter 8

Summary

In this final chapter, we summarize the contributions of this thesis and describe directions for further research.

8.1 Summary of the Problem and the Contributions

In the following, we outline the problem statement and the contents of this thesis. We start by describing the importance of software testing and explain the problem of high testing efforts. Afterwards, we present the phenomenon of commonality of test procedures that influences the activities of creating, executing and maintaining test cases. Having this phenomenon in mind, we provide specific contributions to support test engineers in performing the activities of test creation, execution and maintenance.

8.1.1 Importance of Software Testing

Testing is a central activity in software development, however, it is also among the most expensive activities in software development: For the last decades, efforts for testing have been in the dimension of half of the overall development expenses (see Chapter 1 *Introduction*).

Testing comprises different levels, such as unit testing and integration testing. In this thesis, we limited ourselves to *system testing* as a way to verify whether complete, integrated systems comply with their (mostly functional) requirements. We furthermore focused on the application domain of interactive systems, such as business information systems and on the efforts for system testing that arise during the following three activities of the system test case life-cycle: test creation, test execution, and test maintenance (see Section 1.1.1 *The System Test Case Life-Cycle*).

8.1.2 Commonalities in Test Procedures

Based on our experiences from five years of close collaboration with industry, we identified the phenomenon of *commonality in test cases* as a cause for high system testing efforts: Key elements of all system test life-cycle activities are test cases. Looking at test cases of the same test suite, their flow of test steps (the test procedure) is often very similar. Reasons for that are manifold and lie in the basic idea of testing: To reach high test coverage, as many different paths as possible through the software system are executed. Since these paths often differ just slightly, this inevitably leads to test cases with similar test procedures. Furthermore, if

the functionalities that are tested are alike, then test cases testing this functionality will be alike, too. Lastly, dependencies between functionalities (such as preconditions or inheritance of functions) also lead to commonality of test procedures (see Section 1.1.2 *Commonalities in Test Procedures*).

Commonality in test procedures leads to the following negative consequences for the system test life-cycle: First, commonality in test procedures complicates the decision when to automate tests and when to rely on manual execution. Both execution techniques have advantages and disadvantages. For each test suite, one has to decide when to use which execution technique. However, this decision is not a trivial task, and if made inappropriately, testing budgets can be wasted easily. Second, commonality often leads to bad artifact quality, which hampers executing and maintaining test cases. If test artifacts are inappropriate to support execution and maintenance, testing efforts increase (see Section 1.1.3 *Consequences of Commonality to the System Test Case Life-Cycle*).

8.1.3 Our Approach to Address the Problem

Goal of this thesis is to investigate approaches to reduce overall system test case life-cycle effort focusing on testing of business information systems. We pursued this goal by following two key principles: First, we designed our contributions to be as less invasive as possible to ease their adoption in practice. Second, we addressed the problem of high system testing effort by taking test cases as starting points for our contributions. More specifically, we addressed our goal by supporting all three system test case life-cycle activities: Test case creation, execution, and maintenance. Furthermore, we followed the research approach *industry-as-laboratory* [Potts, 1993] by clarifying the research problem as well as developing contributions with close involvement of industry partners. This allowed us to evaluate our approaches in industrial real-life settings.

In the beginning, we investigated the phenomenon of commonality in real-world test suites and its impact to system testing effort. This study did not only lead to a deeper understanding of the problem domain, but also helped us to identify two relevant challenges we want to address: Choosing adequate execution modes as well as improving and keeping the quality of test artifacts. Afterwards, we followed a two-fold approach addressing the negative consequences of commonality of test procedures:

First, we support test engineers in choosing best fitting execution modes. We presented an effort estimation model that enables test engineers in balancing the advantages and disadvantages of manual and automated test execution and allows finding best fitting execution modes. Thereby, we support the activity of executing test cases and avoid unnecessary expenses for creating and maintaining manual test descriptions or automated test scripts.

Second, we support test engineers in creating and improving high quality test artifacts in two different ways. Building upon existing grammar inference techniques, we developed an approach guiding test engineers in removing test clones, a common negative consequence of commonality of test procedures. Furthermore, we proposed an analytical technique to point to quality problems in test artifacts. Both approaches support test engineers in creating and maintaining high quality test artifacts, which are easier to execute and maintain.

8.1.4 Contributions

Concretely, we made the following four contributions:

A Study on Clones in Manual System Test Cases: We performed clone detection on seven industrial test suites (72 – 1800 manual test cases each). The study identified that the test suites contained between 43% and 86% identical text passages (some appearing up to 30 times). The results of the study supports our assumption that commonality in test artifacts has negative affects to the system test case life-cycle.

An Effort Estimation Model to Identify Adequate Execution Modes: We proposed an effort estimation model to support test engineers in choosing execution modes. Applying our model in industry (41 test cases) revealed that execution modes have a strong influence on testing efforts: Testing efforts differed by the factor of four depending on how manual and automated testing are applied. A coarse benefit estimation indicated that, in our case study, an adequate execution will reduce the overall system testing costs up to ∼20% – ∼30% (within two years).

Constructive Quality Improvement of Test Artifacts: We presented an approach to identify cloned parts in automated test cases and describes specific ways of removing them. Compared with the sate of the art, our approach shows its strength by finding ways of extracting clones that overlap and are therefore difficult to handle manually. A study revealed that our approach is considered beneficial by test engineers: In 16 out of 18 cases, test engineers would implement our refactorings. A coarse benefit estimation showed that our approach pays off after the test suites are maintained 2 – 3 times. The overall system testing costs will be reduced up to ∼10% if the test suites are maintained 5 – 10 times.

Natural Language Test Smells: We introduced natural language test smells, a technique to identify potential quality defects in manual test cases. In an industrial study (9 test suites from two companies, 5,466 test cases in total), we identified that our technique has an average precision of ∼74%. Interviews revealed that 75% of the (correct) findings have been considered by test engineers as worth to be fixed. A coarse benefit estimation showed, that removing smell findings can reduce the overall system testing costs up to ∼10%.

8.2 Outlook and Future Work

In the following sections, we discuss directions for future research. First, we present next steps that are directly related to our work. Second, we discuss alternatives to our approach. Third, we point out future research topics having a larger scope and are addressing not only the system test case life-cycle, but software testing in general.

8.2.1 Future Work Related to Our Approach

This section discusses possible extensions and improvements related to our approach and illustrates directions for further research.

Prioritization of Smell Findings using Cost-Benefit Analysis

The industrial evaluation presented in Chapter 7 *Natural Language Test Smells* has shown that our concept of smells for natural language test cases is able to identify a high number of quality problems: We identified ∼1,500 up to ∼24,000 smell findings per test suite and 75% of the assessed findings have been rated as worth to be fixed. However, time and budgets of test engineers are limited. Therefore, it is unrealistic to assume that each of these findings is getting fixed in near future. This poses the challenge of determining on which smell findings test engineers should focus first.

We believe that a process to prioritize smell findings will help test engineers in dealing with the high number of smell findings. Our study revealed that not all smell findings are similarly severe: For example, 17.4% of the detected smell findings have even been rated as not relevant at all. Additionally, not all findings are equally easy to correct. This implies that trading off benefits of smell finding removal with the expected removal costs is a good starting point towards a prioritization of smell findings. For source code, similar approaches already exist. Steidl and Eder, for example, propose to focus on findings that can be fixed with quick refactorings. Other approaches propose to follow the *Boy Scout Rule* by coupling development tasks with fixing quality findings that affect the same code regions [Martin, 2009; Steidl et al., 2014].

Constructive Solutions for Further Smells

In Chapter 6 *Test Refactoring Using Grammar Inference*, we presented a constructive approach to help test engineers removing quality problems in test artifacts. However, the presented approach focuses only on clones and is (without further adaption) just applicable to automated test scripts in keyword-driven style. For other quality problems, we just provide hints to possible solutions so far.

For some types of quality problems, fixes are rather obvious and trivial to realize. However, other types of quality problems are more difficult to be fixed. We believe that test engineers would benefit from tool support to work quality problems that are difficult to fix manually.

Integrating Analytical and Constructive Quality Improvement

In Chapter 6 *Test Refactoring Using Grammar Inference* and Chapter 7 *Natural Language Test Smells*, we presented two different ways to obtain high artifact quality. Whereas the latter is a purely analytical approach pointing out likely quality problems, the first is a constructive approach helping test engineers in improving test artifacts. Although both approaches have the same goal, they are not integrated yet.

Both approaches are complementary and test engineers would benefit from integrating both. Following the original idea of coupling smells and refactorings [Fowler, 1999], we propose to integrate natural language test smells with constructive approaches such as the one presented in Chapter 6 *Test Refactoring Using Grammar Inference*. Equipping each natural language test smell with constructive ways of fixing the caused quality problems would make it easier for test engineers to find out how to improve the quality of test artifacts.

Using Test Smells as Metrics for Continuous Quality Monitoring

We presented natural language test smell as a tool to support test engineers in identifying quality problems with the goal to fix them and thereby improve the overall quality of test artifacts.

However, we believe that smell findings can also be used differently: By combining smell detection results, new metrics indicating the overall quality of test suites can be defined. For example, metrics, such as *total number of smell findings* or *amount of smell findings per 100 words* can be used to compare the overall quality of similar test suites or different parts of the same test suite. Furthermore, having such metrics allows monitoring the quality of test suites continuously and enables detecting quality trends early.

Integration into Software Development Processes

The approaches of this thesis aim at reducing efforts for creating new test cases and for executing and maintaining existing test cases. To be as generally applicable as possible, we made as little assumptions about the surrounding (software development) activities as possible.

However, in some styles of software development processes, testing plays a particularly important role. Following the development approach of *continuous delivery*, for example, software is produced in short development cycles and deployed (and tested) as early as possible. Furthermore, a common approach in software development projects is to structure testing and deployment processes in several *stages* starting in development environments, going over to dedicated testing environments until the application is finally deployed and tested in a production environment. In each of the stages, individual tests are performed.

Compared to waterfall-like development processes, test cases are executed more frequently and time periods between test runs are shorter. To leverage the approaches presented in this thesis, they have to be integrated into software development processes and to be aligned with existing quality assurance processes in order to be implemented effectively. For example, it has to be defined at what stages or how often within a development process the approaches are performed (in every development cycle or only once per release?) and how the results of the approaches are used (immediately or for the next test run or release?).

8.2.2 Alternative Approaches to Reduce System Testing Effort

In this thesis, we aimed at reducing system testing effort by sticking to our key principles: (1) Contributions should be as little invasive as possible to ease their adoption in practice. (2) Taking test cases as starting point for improvements. (see Section 1.3.1 *Key Principles*). In this section, we discuss alternative approaches that do not follow both of our key principles. More specifically, we present approaches that do not follow principle 1 by proposing contributions that (in some cases) strongly change existing testing processes and tool chains.

Test Suite Optimization

Our approach to a solution does not question the semantics of existing test suites: We take a given set of test cases as granted and do not question their existence. However, a fundamentally different approach to reduce system testing effort is to optimize a test suite in a

way that it is smaller leading to less test execution and maintenance effort (see Section 3.3 *Optimizing Goals of Test Suites*).

Reducing and optimizing system test suites would cover questions such as the relevance of each test case. For example, by prioritizing the aspects of the software system that have to be tested (the *test conditions* – see Section 2.3 *A Generic System Test Process*), less important test cases could be identified. This information could also be considered to find best fitting execution modes (Chapter 5 *Choosing Execution Modes*).

Most existing approaches for test suite reduction and optimization require formal system specifications and formal test requirements which are both uncommon in the area of business information systems. However, clones in test cases could be used as a heuristics to identify redundantly tested functionality. Thereby, redundant realizations of test conditions can be uncovered. Additionally, clones in test cases can be used to find candidates for merging test cases.

Tools and Languages for Semi-Automatic Test Execution

The study in Chapter 5 *Choosing Execution Modes* revealed that mixing manual and automated test execution is a promising way to save test creation, execution, and maintenance effort. The most promising way of mixing both techniques was a combination of both execution techniques even within single test cases – so called *Semi-Automatic Test Execution*.

However, implementing semi-automatic test execution requires testing tools that allow mixing both techniques freely. Based on our observations while performing case studies, most available testing tools are designed to dedicatedly support creation and execution of either manual or automated test cases. To leverage the benefits of semi-automatic test execution, we need tools and languages that enable easy and stepwise implementation of semi-automatic test execution.

Tools and Languages for High Quality Test Artifacts

In interviews with test engineers (see Chapter 4, Chapter 6, and Chapter 7), it turned out that many quality issues concerning the executability and maintainability of test cases are due to inadequacy of testing tools and languages. For example, complex phrases in manual test descriptions are encouraged by writing unrestricted natural language text. One approach is to write manual test cases exclusively in form of semi-structured text, using text patterns of phrases to be used. For example, behavior-driven development (BDD) (see Chapter 3 *State of the Art*), a technique to write test cases in form of user stories, proclaims to write test cases in a *Given-When-Then* style. However, user stories are not suitable to write system test cases in every setting. Furthermore, test engineers claimed that in many cases, clones in test procedures were inevitable since the used test definition tools did not provide sufficient reuse abilities. Tools did not provide fitting abstraction mechanisms to extract cloned parts to reuse components.

Tools and languages for test cases have a strong influence on the quality of the resulting artifacts. To have high quality artifacts, tools and test definition languages have to support test engineers in writing test cases as well as to guide them in avoiding typical quality problems.

8.2.3 Beyond the System Test Case Life-Cycle

Similar to the previous section, we now discuss alternative approaches to reduce system testing effort that do not follow our key principles. We present approaches that do not follow both of our principles since they are not limited to the system test case life-cycle, but consider testing in general.

Optimized Test Strategies

Software testing as a quality assurance method consists of different test levels, such as unit testing, integration testing, or system testing (see Chapter 2 *Fundamentals, Terms, and Definitions*). Mostly, several test levels are mixed to reach a project's testing goal. Since many test conditions can be tested using different test levels, many strategies of mixing test levels are possible. However, test levels have strengths and weaknesses. Therefore, not each test level is suitable for each type of test condition. This leads to test strategies that are not efficient since they are using test levels in unsuitable ways. For example, exhaustive testing of software libraries using manual testing on the level of the graphical user interface is inefficient. In this case, (automated) unit testing combined with data-driven techniques is in most cases by far more efficient.

A completely different way of reducing testing effort compared with ours is to consider all levels of testing as a whole and optimize complete testing strategies of software projects. Thereby, more efficient ways of verifying the quality of software systems could be found. This would lead to less creation, execution, and maintenance effort for system testing.

Structured Derivation of Test Conditions and Test Cases

Based on our experience, test conditions and test procedures for system testing of business information systems are mostly created manually. Test engineers analyze system specifications and define testable aspects of the system – so called test conditions. Based on these test conditions, test cases including test procedures are created (see *Test Analysis* and *Test Design* in Section 2.3 *A Generic System Test Process*). Both steps are labor intensive and error prone.

A test analysis and design process that is more structured and tool supported has the following advantages: Having a structured way of deriving test conditions from system specifications would ensure that all important aspects of a software system have been considered in testing. Furthermore, a structured way of deriving test conditions and test cases would allow linking test cases to those parts of the system specification that are tested. This enables detailed reasoning about the coverage of executed test cases and of identifying erroneous functionality from failed test cases. Furthermore, functional changes of the software system can easily be tracked from the system specification to identify test cases that have to be adapted.

Although model-based testing aims at similar goals, most existing approaches are not applicable for business information systems: Model-based testing approaches target at deriving low-level test cases completely automatically and therefore require system requirements in form of formal specifications, such as state machines. However, business information systems are often specified in form of informal or semi-formal requirements documents using natural language text (for example, in form of use-case documents). By creating languages and tools to model requirements of business information systems in a more structured form (for example, having a clear meta-model with well-defined semantics of elements), test cases could be semi-automated derived from requirements.

Quality of Semantics of Test Cases

In this thesis, we focused on test executability, understandability and maintainability but never challenged the execution behavior of test cases. We assumed that all test cases have correctly been transferred to high-level test cases and test descriptions or test scripts. However, if this assumption is wrong, test results lead to wrong conclusions about the system's correctness.

By developing quality measures regarding the semantics of test cases, it would be possible to identify unintended behavior of test cases. Having such measures, it could be verified whether test cases are actually verifying the functionality they are executing. For example, analyzing the assertions within test procedures may show whether test cases do not only execute the system under test, but also verify its response (*assertion coverage* [Piziali, 2004]). Furthermore, aligning test conditions with the source code of the system under test enables verifying whether the correct functionality of the system under test is executed [Eder et al., 2013]. Thereby, the overall process of testing would be more effective.

Bibliography

Abbes, M., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of CSMR*, pages 181–190. IEEE.

Altman, D. G. (1990). *Practical statistics for medical research.* CRC press.

Aranha, E. and Borba, P. (2007). An estimation model for test execution effort. In *Proceedings of ESEM*, pages 107–116. IEEE.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Proceedings of ICSM*, pages 56–65. IEEE.

Beck, K. (1999). Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77.

Beck, K. (2000). *Extreme programming explained: Embrace change.* Addison-Wesley.

Beck, K. (2003). *Test-driven development: By example.* Addison-Wesley.

Beizer, B. (1990). *Software Testing Techniques.* Van Nostrand Reinhold, 2nd edition.

Blackburn, M., Busser, R., and Nauman, A. (2004). Why model-based test automation is different and what you should know to get started. In *Proceedings of PSQT*, pages 212–232. ACM.

Boehm, B. W. (1979). Guidelines for verifying and validating software requirements and design specifications. In *Proceedings of EURO IFIP*, pages 75–88. IEEE.

Boehm, B. W., Clark, Horowitz, Brown, Reifer, Chulani, Madachy, R., and Steece, B. (2000). *Software cost estimation with COCOMO II.* Prentice Hall.

Bourque, P. and Fairley, R. E. (2014). *Guide to the software engineering body of knowledge (SWEBOK).* IEEE.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A. (2005). *Model-based testing of reactive systems: Advanced lectures.* Springer.

Byrt, T. (1996). How good is that agreement? *LWW Epidemiology*, 7(5):561.

Chen, T. and Lau, M. (1998). A new heuristic for test suite reduction. *Elsevier Information and Software Technology*, 40(5–6):347–354.

Chen, T. Y. and Lau, M. F. (1996). Dividing strategies for the optimization of a test suite. *Elsevier Information Processing Letters*, 60(3):135–141.

Chen, W.-K., Shen, Z.-W., and Chang, C.-M. (2008). GUI test script organization with component abstraction. In *Proceedings of SSIRI*, pages 128–134. IEEE.

Chen, W.-K. and Wang, J.-C. (2012). Bad smells and refactoring methods for GUI test scripts. In *Proceedings of SSNPD*, pages 289–294. IEEE.

Chikofsky, E. and Cross, J.H., I. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Durham Educational and psychological measurement*, 20(1):37–46.

Cohn, M. (2004). *User stories applied: For agile software development*. Addison-Wesley.

Cunha, J., Fernandes, J. P., Ribeiro, H., and Saraiva, J. (2012). Towards a catalog of spreadsheet smells. In *Proceedings of ICCSA*, pages 202–216. Springer.

de Almeida, É. R. C., de Abreu, B. T., and Moraes, R. (2009). An alternative approach to test effort estimation based on use cases. In *Proceedings of ICST*, pages 279–288. IEEE.

Deissenboeck, F., Heinemann, L., Hummel, B., and Wagner, S. (2012). Challenges of the dynamic detection of functionally similar code fragments. In *Proceedings of CSMR*, pages 299–308. IEEE.

Deissenboeck, F., Hummel, B., Jürgens, E., Schätz, B., Wagner, S., Girard, J.-F., and Teuchert, S. (2008). Clone detection in automotive model-based development. In *Proceedings of ICSE*, pages 603–612. ACM.

Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., and Girard, J.-F. (2007). An activity-based quality model for maintainability. In *Proceedings of ICSM*, pages 184–193. IEEE.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). *Object-oriented reengineering patterns*. Elsevier.

Domann, C., Juergens, E., and Streit, J. (2009). The curse of copy&paste – cloning in requirements specifications. In *Proceedings of ESEM*, pages 443–446. IEEE.

Dustin, E., Rashka, J., and Paul, J. (1999). *Automated software testing: Introduction, management, and performance*. Addison–Wesley.

Eder, S., Hauptmann, B., Junker, M., Juergens, E., Vaas, R., and Prommer, K.-H. (2013). Did we test our changes?: Assessing alignment between tests and development in practice. In *Proceedings of AST*, pages 107–110. IEEE.

Elbaum, S., Malishevsky, A., and Rothermel, G. (2001). Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of ICSE*, pages 329–338. IEEE.

Elbaum, S., Malishevsky, A. G., and Rothermel, G. (2000). Prioritizing test cases for regression testing. In *Proceedings of ISSTA*, pages 929–948. IEEE.

ETSI (2005-06). Standard 201 873-1: The testing and test control notation version 3 – Part 1: TTCN-3 core language (v3.1.1).

Fabbrini, F., Fusani, M., Gnesi, S., and Lami, G. (2001). An automatic quality evaluation for natural language requirements. In *Proceedings of REFSQ*, pages 4–5.

Fanta, R. and Rajlich, V. (1999). Restructuring legacy C code into C++. In *Proceedings of ICSM*, pages 77–85. IEEE.

Femmer, H. (2013). Reviewing natural language requirements with requirements smells – A research proposal. In *Proceedings of IDoESE*.

Femmer, H., Méendez Fernáandez, D., Juergens, E., Klose, M., Zimmer, I., and Zimmer, J. (2014). Rapid requirements checks with requirements smells: Two case studies. In *Proceedings of RCoSE*, pages 10–19. ACM.

Femmer, H., Mund, J., and Méendez Fernáandez, D. (2015). It's the activities, stupid! A new perspective on re quality. In *Proceedings of RET*, pages 13–19. IEEE.

Fewster, M. and Graham, D. (1999). *Software test automation: Effective use of test execution tools*. Addison-Wesley.

Fleiss, J. L., Levin, B., and Paik, M. C. (2013). *Statistical methods for rates and proportions*. Wiley.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley.

Garmus, D. and Herron, D. (2001). *Function point analysis: Measurement practices for successful software projects*. Addison-Wesley.

Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., and Willcock, C. (2003). An introduction to the testing and test control notation (TTCN-3). *Elsevier Computer Networks*, 42(3):375–403.

Greiler, M., Van Deursen, A., and Zaidman, A. (2012). Measuring test case similarity to support test suite understanding. In *Proceedings of TOOLS*, pages 91–107. Springer.

Harrold, M. J. (2000). Testing: A roadmap. In *Proceedings of ICSE*, pages 61–72. ACM.

Harrold, M. J., Gupta, R., and Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2:270–285.

Hauptmann, B., Bauer, V., and Junker, M. (2012a). Using edge bundle views for clone visualization. In *Proceedings of IWSC*, pages 86–87. IEEE.

Hauptmann, B., Eder, S., Junker, M., Juergens, E., and Woinke, V. (2015). Generating refactoring proposals to remove clones from automated system tests. In *Proceedings of ICPC*, pages 115–124. IEEE.

Hauptmann, B. and Junker, M. (2011). Utilizing user interface models for automated instantiation and execution of system tests. In *Proceedings of ETSE*, pages 8–15. ACM.

Hauptmann, B., Junker, M., Eder, S., Amann, C., and Vaas, R. (2014). An expert-based cost estimation model for system test execution. In *Proceedings of ICSSP*, pages 159–163. ACM.

Hauptmann, B., Junker, M., Eder, S., Heinemann, L., Vaas, R., and Braun, P. (2013). Hunting for smells in natural language tests. In *Proceedings of ICSE*, pages 1217–1220. IEEE.

Hauptmann, B., Junker, M., Eder, S., Juergens, E., and Vaas, R. (2012b). Can clone detection support test comprehension? In *Proceedings of ICPC*, pages 209–218. IEEE.

Hermans, F., Pinzger, M., and van Deursen, A. (2012a). Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of ICSE*, pages 441–451. IEEE.

Hermans, F., Pinzger, M., and van Deursen, A. (2012b). Detecting code smells in spreadsheet formulas. In *Proceedings of ICSM*, pages 409–418. IEEE.

Hermans, F., Pinzger, M., and van Deursen, A. (2014). Detecting and refactoring code smells in spreadsheet formulas. *Springer Empirical Software Engineering*, 20(2):549–575.

Hermans, F., Sedee, B., Pinzger, M., and van Deursen, A. (2013). Data clone detection and visualization in spreadsheets. In *Proceedings of ICSE*, pages 292–301. IEEE.

Hicinbothom, J. H. and Zachary, W. W. (1993). A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, pages 1042–1042. SAGE.

Hoffman, D. (1999). Cost benefits analysis of test automation. In *Proceedings of STAR East*.

Holten, D. (2006). Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748.

Hussain, I., Ormandjieva, O., and Kosseim, L. (2007). Automatic quality assessment of srs text by means of a decision-tree-based text classifier. In *Proceedings of QSIC*, pages 209–218. IEEE.

IEEE (1990). Standard 610.12-1990: Glossary of software engineering terminology.

IEEE (1993). Standard 830-1993: Recommended practice for software requirements specifications.

IEEE (2008). Standard 829-2008: Standard for software and system test documentation.

International software testing qualifications board (2011). Foundation level syllabus.

International software testing qualifications board (2014a). Foundation level extension syllabus agile tester.

International software testing qualifications board (2014b). Istqb/gtb standard glossary of terms used in software testing (v2.4).

ISO/IEC/IEEE (2013a). Standard 29119-1: Software and systems engineering – Software testing – Part 1: Concepts and definitions.

ISO/IEC/IEEE (2013b). Standard 29119-2: Software and systems engineering – Software testing – Part 2: Test processes.

Itkonen, J. and Rautiainen, K. (2005). Exploratory testing: A multiple case study. In *Proceedings ESE*. IEEE.

Jones, J. A. and Harrold, M. J. (2003). Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209.

Juergens, E., Deissenboeck, F., Feilkas, M., Hummel, B., Schaetz, B., Wagner, S., Domann, C., and Streit, J. (2010a). Can clone detection support quality assessments of requirements specifications? In *Proceedings of ICSE*, pages 79–88. ACM.

Juergens, E., Deissenboeck, F., and Hummel, B. (2010b). Code similarities beyond copy & paste. In *Proceedings of CSMR*, pages 78–87. IEEE.

Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. (2009). Do code clones matter? In *Proceedings of ICSE*, pages 485–495. IEEE.

Jurafsky, D. and Martin, J. H. (2000). *Speech & language processing*. Pearson.

Kaner, C., Bach, J., and Pettichord, B. (2002). *Lessons learned in software testing*. Wiley.

Khomh, F., Di Penta, M., and Gueheneuc, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. In *Proceeding of WCRE*, pages 75–84. IEEE.

Koschke, R. (2006). Survey of research on software clones. In *Duplication, redundancy, and similarity in software*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

Kruchten, P. (2003). *The rational unified process: An introduction*. Addison-Wesley, 3rd edition.

Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *JSTOR Biometrics*, 3(11):159–174.

Leotta, M., Clerissi, D., Ricca, F., and Spadaro, C. (2013a). Improving test suites maintainability with the page object pattern: an industrial case study. In *Proceedings of ICSTW*, pages 108–113. IEEE.

Leotta, M., Clerissi, D., Ricca, F., and Tonella, P. (2013b). Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of WCRE*, pages 272–281. IEEE.

Linz, T. and Daigl, M. (1998). Automated software testing. In Dustin, editor, *Implementation and results of the ESSI Project Number 24306, 1998. In: Dustin, et. al., Automated software testing*, page 52. Addison-Wesley.

Lochmann, K. (2010). Engineering quality requirements using quality models. In *Proceedings of ICECCS*, pages 245–246. IEEE.

Lötzbeyer, H. and Pretschner, A. (2000). Testing concurrent reactive systems with constraint logic programming. In *Proceedings of RCoRP*.

Mäntylä, M. V. and Lassenius, C. (2006). Subjective evaluation of software evolvability using code smells: An empirical study. *Springer Empirical Software Engineering*, 11(3):395–431.

Marick, B. (1999). When should a test be automated? In *Proceedings of STAR East*.

Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Pearson.

Mcgrath, J. E. (1995). Methodology matters: Doing research in the behavioral and social sciences. In *Readings in Human-Computer Interaction: Toward the Year 2000 (2nd ed.)*, pages 152–169. Morgan Kaufmann.

Memon, A. M. and Soffa, M. L. (2003). Regression testing of GUIs. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127.

Mens, T. and Tourwe, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Addison-Wesley.

Meszaros, G., Smith, S., and Andrea, J. (2003). The test automation manifesto. In *XP'03*, pages 73–81. Springer.

Mohagheghi, P., Anda, B., and Conradi, R. (2005). Effort estimation of use cases for incremental large-scale software development. In *Proceedings of ICSE*, pages 303–311. IEEE.

Myers, G. J. and Sandler, C. (2004). *The art of software testing*. John Wiley & Sons.

Neukirchen, H. and Bisanz, M. (2007). Utilizing code smells to detect quality problems in TTCN-3 test suites. In *Testing of Software and Communicating Systems*, volume 4581 of *LNCS*, pages 228–243. Springer.

Neukirchen, H., Zeiss, B., Grabowski, J., Baker, P., and Evans, D. (2008). Quality assurance for TTCN-3 test specifications. *Wiley Software Testing, Verification and Reliability*, 18(2):71–97.

Nevill-Manning, C. G. (1996). *Inferring sequential structure*. PhD thesis, University of Waikato.

Nevill-Manning, C. G. and Witten, I. H. (1997). Identifying hierarchical structure in sequences: A linear-time algorithm. *AI Access Foundation Journal of Artificial Intelligence Research*, 7:67–82.

North, D. (2006). Behavior modification: The evolution of behavior-driven development. *Better Software Magazine*, 8(3).

Offutt, A. J., Pan, J., and Voas, J. M. (1995). Procedures for reducing the size of coverage-based test sets. In *Proceedings of ICST*, pages 111–123.

Opdyke, W. F. (1992). *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

Parnas, D. L. (1994). Software aging. In *Proceedings of ICSE*, pages 279–287. IEEE.

Persson, C. and Yilmazturk, N. (2004). Establishment of automated regression testing at ABB: Industrial experience report on 'avoiding the pitfalls'. In *Proceedings of ASE*, pages 112–121. IEEE.

Pham, H. and Zhang, X. (1999). A software cost model with warranty and risk costs. *IEEE Transaction on Computers*, 48(1):71–75.

Piziali, A. (2004). *Functional verification coverage measurement and analysis*. Springer.

Pol, M., Teunissen, R., and Van Veenendaal, E. (2001). *Software Testing: A guide to the TMap Approach*. Addison-Wesley.

Potts, C. (1993). Software-engineering research revisited. *IEEE Software*, 10(5):19–28.

Ramler, R. and Wolfmaier, K. (2006). Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. In *Proceedings of AST*, pages 85–91. ACM.

Rothermel, G., Harrold, M., Ostrin, J., and Hong, C. (1998). An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of ICSM*, pages 34–43. IEEE.

Rothermel, G., Untch, R., Chu, C., and Harrold, M. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948.

Roy, C. K. and Cordy, J. R. (2007). A survey on software clone detection research. Technical Report 007-541, Queen's University at Kingston Ontario, Canada.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *The unified modeling language reference manual*. Pearson Higher Education, 2nd edition.

Schmid, H. and Laws, F. (2008). Estimation of conditional probabilities with decision trees and an application to fine-grained pos tagging. In *Proceedings of ICCL*, pages 777–784. Association for Computational Linguistics.

Schwaber, C. and Gilpin, M. (2005). Evaluating automated functional testing tools. Technical report, Forrester Research.

Sommerville, I. (2010). *Software Engineering*. Addison-Wesley, 9th edition.

Spillner, A., Linz, T., and Schaefer, H. (2007). *Software testing foundations: A study guide for the certified tester exam*. Rocky Nook.

Steidl, D., Deissenboeck, F., Poehlmann, M., Heinke, R., and Uhink-Mergenthaler, B. (2014). Continuous software quality control in practice. In *Proceedings of ICSME*, pages 561–564. IEEE.

Steidl, D. and Eder, S. (2014). Prioritizing maintainability defects by refactoring recommendations. In *Proceedings of ICPC*, pages 168–176. ACM.

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. Technical report, NIST.

Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of NAACL HLT*, pages 173–180. Association for Computational Linguistics.

Utting, M. and Legeard, B. (2010). *Practical model-based testing: A tools approach.* Morgan Kaufmann.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Wiley Software Testing, Verification and Reliability*, 22(5):297–312.

van Deursen, A. and Moonen, L. (2002). The video store revisited – Thoughts on refactoring and testing. In *Proceedings of XP*, pages 71–76. Citeseer.

van Deursen, A., Moonen, L., van den Bergh, A., and Kok, G. (2001). Refactoring test code. In *Proceedings of XP*, pages 92–95. University of Cagliary.

Wagner, S., Deissenboeck, F., and Winter, S. (2008). Managing quality requirements using activity-based quality models. In *Proceedings of WoSQ*, pages 29–34. ACM.

Wagner, S., Goeb, A., Heinemann, L., Kläs, M., Lampasona, C., Lochmann, K., Mayr, A., Plösch, R., Seidl, A., Streit, J., et al. (2015). Operationalised product quality models and assessment: The Quamoco approach. *Elsevier Information and Software Technology*, 62:101–123.

Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R., Seidl, A., Goeb, A., and Streit, J. (2012). The Quamoco product quality modelling and assessment approach. In *Proceedings of ICSE*, pages 1133–1142. IEEE.

Watson, A. H., McCabe, T. J., and Wallace, D. R. (1996). Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500(235):1–114.

Wilson, W. M., Rosenberg, L. H., and Hyatt, L. E. (1997). Automated analysis of requirement specifications. In *Proceedings of ICSE*, pages 161–171. ACM.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: An introduction.* Kluwer Academic Publishers.

Xiaochun, Z., Bo, Z., Fan, W., Yi, Q., and Lu, C. (2008). Estimate test execution effort at an early stage: An empirical study. In *Proceedings of CW*, pages 195–200. IEEE.

Yu, Y., Jones, J. A., and Harrold, M. J. (2008). An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of ICSE*, pages 201–210. ACM.

Zeiss, B., Neukirchen, H., Grabowski, J., Evans, D., and Baker, P. (2006). Refactoring and metrics for TTCN-3 test suites. In *System Analysis and Modeling: Language Profiles*, volume 4320 of *LNCS*, pages 148–165. Springer.

Zhu, X., Zhou, B., Hou, L., Chen, J., and Chen, L. (2008). An experience-based approach for test execution effort estimation. In *Proceedings of CYCS*, pages 1193–1198. IEEE.