Exploiting Execution Profiles in Software Maintenance and Test

Sebastian Eder

Institut für Informatik
der Technischen Universität München

Exploiting Execution Profiles in Software Maintenance and Test

Sebastian Eder

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:		Univ.-Prof. Bernd Brügge, Ph.D.
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
	2.	Univ.-Prof. Dr. Alexander Pretschner

Die Dissertation wurde am 03.08.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.10.2016 angenommen.

Abstract

Many business information systems are in use for decades and due to changing users, environments, or business models, the systems are frequently updated. Such changes are implemented during software maintenance, which accounts for 60% to 90% of the overall life-cycle costs of software systems.

In a traditional maintenance process, users are only sparsely involved. In consequence, product owners identify, prioritize and analyze changes with little to no information about how users typically use which parts of the software system. This leads to developers implementing changes in unused functionality, which does not provide any value for the users. Maintenance in this potentially useless functionality might not provide value to the users and therefore imposes the risk of being a waste of resources.

Insights into usage are therefore one prerequisite to avoid ineffective maintenance. We measure usage by execution profiles that express, which parts of a software system were executed during a given period of time.

In this thesis, we explore the benefit of analyzing execution profiles collected for business information systems to reduce maintenance efforts. We contribute a series of studies to gain insights into the actual execution of source code and its implications, approaches that transfer execution profiles to other artifacts, and we embed the approaches into the maintenance process.

In our first case study, we quantify unused source code in a business information system and confirm that maintenance in unused source code can be a waste of resources. Building on these insights, we present a constructive approach to transfer the information contained in execution profiles to requirements artifacts by detecting links between source code and requirements. The evaluation of this approach shows that it is capable of detecting requirements artifacts that express unused functionality. We embed this approach in the maintenance process for product owners and developers and explain how product owners can prioritize change requests and file new change requests based on knowledge about usage.

After changing software systems in maintenance, regression testing is performed to validate that newly created or changed code did not impair existing functionality. We explore applications of execution profiles to also support testing activities.

In our second study, we quantify untested code in a business information system and empirically underpin the assumption that the fault rate in changed but untested source code is higher than in other code regions. Building on that, we present a constructive approach to indicate gaps in code coverage, which are indicated by execution profiles, and to relate them to existing regression test cases. In an evaluation, we show that, in 90% of all cases, our approach identifies a test case that actually covers the gap in question. We furthermore embed these techniques in the maintenance process and explain how test engineers can assess whether the changes, which were made during maintenance, are covered by tests. We further suggest a method for selecting regression test cases based on this knowledge.

The aforementioned approaches rely on detecting semantic links between artifacts. To acquire these links, we provide a constructive approach for detecting semantic links between artifacts fully automatically. We evaluate the approach in a case study and confirm that the proposed approach is capable of detecting semantic links between artifacts accurately. With this, we foster, technically, the applicability of the aforementioned contributions.

Acknowledgments

Writing this thesis reminded more than once on climbing a huge mountain. In the beginning, the mountain seems too high to be climbed. During climbing, there are exhausting ups and downs. But close to the summit, all the effort is forgotten and the fun outweighs. Climbing serious mountains always is teamwork, and impossible without the support of fellow climbers. Now, that I am close to my personal summit, the completion of this thesis, I sincerely want to thank all the people, who helped me in reaching this summit I climbed over the last years.

First of all, I want to express my gratitude to Prof. Manfred Broy. He gave me the opportunity to work at his chair in this great atmosphere of scientific thinking. I am thankful for his continuous support and motivation. He guided me up this mountain, while giving me the possibility to realize my own ideas. I also want to thank Prof. Alexander Pretschner for co-supervising this thesis, for giving very helpful comments, and for motivating me.

My thanks go to all the great people of Prof. Broy's research group, who helped me during this endeavor. In particular, I want to thank Benedikt Hauptmann, Henning Femmer, and Maximilian Junker for the numerous discussions and reviews of this thesis. During these discussion, I learnt a lot and they often changed my mind and way of thinking. These people allowed me to be part of a great (rope) team, and I am very glad they trust me even beyond research. Elmar Jürgens gave me a great start into the scientific world and his support, especially in the beginning of this endeavor, heavily influenced, certainly very positively, my way of problem solving and scientific writing. Moreover, he initiated many ideas that are written down in this thesis. Even though Daniel Méndez Fernández does look good, he might not be in the physical condition to climb real mountains. But, he was always in the mental condition to support me in all questions, especially regarding theory of science. His exceptional knowledge and our discussions brought me further in this thesis and in topics way beyond it. I thank my part-time office mate Daniela Steidl for great and funny times, but also for all the help in scientific questions. And, for letting me co-author an award-winning paper! I furthermore thank Jonas Eckhard, Andreas Vogelsang, and Veronika Bauer for the feedback, discussions and advice, but also for the motivation and distraction that helped me over exhausting passages more than once. I particularly thank Silke Müller for guiding me through the bureaucracy at the university, and also for always raising my blood sugar level with sweets.

I'd like to thank my friends Christoph Frenzel and Josef Graubmann for climbing real mountains with me. These trips were vital distractions and the experiences I made there – there is always a better option than giving up, and usually one can do more than he would expect – gave me a lot of trust in the success of this project. I furthermore thank Christoph Frenzel, Severin Strobl and Martin Schreiber for numerous discussions about computer science and other topics during late night sessions that kept my mind open for new ideas, different points of view, and inspired me to begin this whole project.

All of this could not have happened without the unconditional trust of my parents through all my life. Everything I do, and everything I have done, would not have been possible without their tremendous mental and material support. I thank my brother for always being a role model, for being the older one, and for the chance to grow at his side. By being with me for my whole life, my parents and my brother gave me the prerequisites and strength to write this dissertation.

Above all, I thank Julia. She paid the largest bill during these intense years. During all times, she covered my back and supported me regardless of what happened or how stressed I was. This gave me the strength to climb this mountain. With her, I feel the strength for climbing even higher ones.

Publication Preface

[1] **Publication A**:
© 2012 IEEE. Reprinted, with permission, from
Sebastian Eder, Maximilian Junker, Elmar Juergens, Benedikt Hauptmann, Rudolf Vaas, Karl-Heinz Prommer,
How much does unused code matter for maintenance?,
34th International Conference on Software Engineering (ICSE), 2012

[2] **Publication B**:
© 2013 IEEE. Reprinted, with permission, from
Sebastian Eder, Benedikt Hauptmann, Maximilian Junker, Elmar Juergens, Rudolf Vaas, Karl-Heinz Prommer,
Did we test our changes? Assessing alignment between tests and development in practice,
8th International Workshop on Automation of Software Test (AST), 2013

[3] **Publication C**:
© 2014 Association for Computing Machinery, Inc. Reprinted by permission.
Sebastian Eder, Benedikt Hauptmann, Maximilian Junker, Rudolf Vaas, Karl-Heinz Prommer,
Selecting manual regression test cases automatically using trace link recovery and change coverage,
9th International Workshop on Automation of Software Test (AST), 2014
http://dx.doi.org/10.1145/2593501.2593506

[4] **Publication D**:
© 2014 IEEE. Reprinted, with permission, from
Sebastian Eder, Henning Femmer, Benedikt Hauptmann, Maximilian Junker,
Which Features Do My Users (Not) Use?,
IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014

[5] **Publication E**:
© 2015 IEEE. Reprinted, with permission, from
Sebastian Eder, Henning Femmer, Benedikt Hauptmann, Maximilian Junker,
Configuring Latent Semantic Indexing for Requirements Tracing,
2nd International Workshop on Requirements Engineering and Testing (RET), 2015

Contents

1. Introduction 1
1.1. Context 1
1.2. Motivation 2
1.2.1. Anecdotal Evidence 2
1.2.2. From Literature 3
1.2.3. Observations in Practice 4
1.3. Problem Statement 5
1.3.1. Execution Profiles 6
1.3.2. Research Objective 6
1.4. Contributions 6
1.4.1. Provide Insights into the Actual Usage of a Software System to Product Owners 7
1.4.2. Provide Insights into Regression Test Coverage to Test Engineers 8
1.4.3. Automatically Linking Source Code and Other Artifacts 9
1.4.4. Summary 10
1.5. Overview 10

2. Background 11
2.1. Software Maintenance Process 12
2.2. Software Maintenance 13
2.2.1. Definition 13
2.2.2. Types of Maintenance 13
2.2.3. Involved Roles 14
2.2.4. Change Requests 15
2.3. Software Test in Maintenance 16
2.3.1. Definition 16
2.3.2. Software Testing Process 16
2.3.3. Roles 17
2.3.4. Levels of Granularity 18
2.3.5. Goals of System Testing 19
2.3.6. Regression Test Case Selection 20
2.3.7. Test Coverage 20
2.4. The Gap between Developers, Product Owners, and Users 21
2.5. Software Usefulness and Usage 23
2.5.1. Usefulness 23
2.5.2. Connection of Usefulness and Actual Usage 24

2.6. Summary 25

3. Related Work **27**
3.1. Challenges in Software Maintenance 28
3.1.1. Understanding the User 28
3.1.2. Questions about Users 28
3.2. Involving Users in the Software Maintenance Process 29
3.2.1. User Involvement 29
3.2.2. Collection of Data about User Behavior 31
3.2.3. Summary 41

4. Execution Profiles in Software Maintenance and Test **43**
4.1. Characterization of Execution Profiles 44
4.1.1. Description 44
4.1.2. Profiling Technique 44
4.1.3. Collected Data 45
4.1.4. Data Collectors 46
4.2. Relation to Functionality 46
4.2.1. Relation of Functionality to Source Code 46
4.2.2. Relation of Execution Profiles to Usage of Functionality 48
4.3. Relation to Test Cases 50
4.3.1. Relation of Test Cases to Source Code 50
4.3.2. Relation of Execution Profiles to Code Coverage 50
4.4. Conclusions based on Execution Profiles 50
4.4.1. Conclusions on Usage Data 51
4.4.2. Conclusions on Coverage Data 53
4.5. Comparison 54
4.5.1. Advantages of Execution Profiles 54
4.5.2. Disadvantages of Execution Profiles 56
4.5.3. Summary 56
4.6. Summary 57

5. Contributions **59**
5.1. Unused Source Code in Maintenance 62
5.2. Transfer of Unused Source Code to Requirements Artifacts 63
5.3. Considering Usage Data in Maintenance 64
5.3.1. Product Owner 65
5.3.2. Developer 66
5.3.3. Process 67
5.3.4. Implications on Resources 68
5.4. Uncovered Source Code in Regression Testing 70
5.5. Transfer of Uncovered Source Code to Regression Test Cases 71
5.6. Considering Coverage Data in Testing 72
5.6.1. Test Engineer 72
5.6.2. Process 73
5.6.3. Implications on Resources 74
5.7. Automatically Linking Source Code with Other Artifacts 80

6. Conclusions **81**
6.1. Summary 82
6.1.1. Contributions 82
6.1.2. Key Takeaways 84
6.2. Limitations 85
6.2.1. Limitation to Existing Artifacts 85

6.2.2. Limited accuracy 86
6.2.3. Assumptions 87
6.2.4. Generalizability 89
6.3. Future Work 89

Bibliography **91**

Appendices **101**

A. Publication A [1] **101**

B. Publication B [2] **113**

C. Publication C [3] **119**

D. Publication D [4] **127**

E. Publication E [5] **133**

F. Reprint Permission for Publication A [1] **141**

G. Reprint Permission for Publication B [2] **143**

H. Reprint Permission for Publication C [3] **145**

I. Reprint Permission for Publication D [4] **149**

J. Reprint Permission for Publication E [5] **151**

CHAPTER 1

Introduction

1.1. Context

In this thesis, we consider business information systems, which are "software systems to support forecasting, planning, control, coordination, decision making, and operational activities in organizations" [6]. These systems provide value to their users by supporting their workflows in their daily business. We consider business information systems that are used interactively by users via a user interface, and not systems, that are merely triggered by other systems. Often, these systems are developed and maintained for specialized users in a specific environment. These specialized users are experts in their working domain and therefore know, to a large extent, which functionality supports them in their working environment, and which does not, once they see it. In this context, users perceive functionality as useful, if using it increases their productivity, job performance, or effectiveness on the job [7].

Business information systems are often in use for decades in industry [8]. During these long periods of time, the users, environments, or business models change. Therefore, the software systems themselves have to be adapted to these changes.

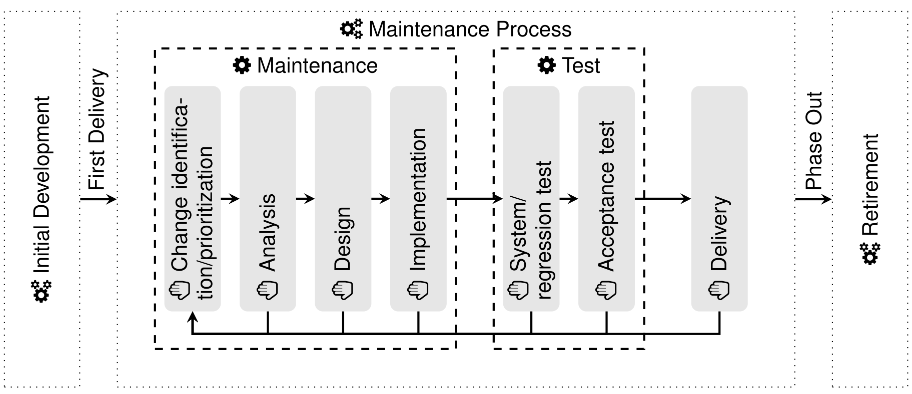


Figure 1.1.: Simplified view of the software life-cycle and the maintenance process.

We consider software maintenance as the activity of identifying and implementing changes to a software system. This activity is embedded in the software maintenance process [9], which is triggered by the aforementioned changes, and is illustrated in Figure 1.1. In this the first phase of this process, the change identification, the product owner[1] collects changes from users, identifies changes on his own, and prioritizes them. In the second and third phases, analysis and design, the change is evaluated for feasibility and coarse and fine grained plans for its implementation are developed. In the fourth phase, implementation, the developers implement the change in source code (and perform therefore the activity maintenance). After this, the test engineer leads the fifth phase, system testing, where testers execute test cases to ensure the new functionality is correct and existing was not impaired (regression testing). Then, the customer validates whether to accept the changed system in the phase acceptance testing. If so, the system is delivered to the customer in the phase delivery, and otherwise, the maintenance process starts from the beginning. This process is repeated during the whole life-cycle of a software system.

Maintenance can be categorized into four categories [10]. Corrective maintenance are modifications to the software system to correct faults. Perfective maintenance are additions of new functionality or modifications to existing functionality. Adaptive maintenance are changes that adapt a software system to new environments, e.g., adapt a system to changed communication protocols with other software systems. Preventive maintenance prepare a software system for future modifications, e.g., improving the comprehensibility of the source code of a software system.

1.2. Motivation

In traditional maintenance processes, like the waterfall process, users and customers are involved only sparsely. In these traditional maintenance processes, anecdotal evidence, literature, as well as our own observations in practice point to several challenges that arise due to a lack of the understanding of users and customers. These challenges indicate that information about usage and coverage is interesting for product owners, developers, test engineers, and testers.

1.2.1. Anecdotal Evidence

Agile development focuses on direct collaboration of the stakeholders developing the software and the customers [11]. The goal is to get rid of unnecessary documentation, while developers understand the requirements of the customer.

In lean software development, the reduction of *waste* is a central goal [12, 13]. Waste is, e.g., work that is only done partially, useless functionality, or defects. Especially the saying *"You ain't gonna need it!"* [14] is well known nowadays and it means that developers should be careful which functionality they implement, since lots of functionality does not provide value to the users, even though the developers think it is useful.

IBM reports that organizations waste about 40% or more of their resources [15]. Jim Johnson of the Standish Group reported, describing a study about four software systems, that 45% of all features of software systems are never used, and additional 19% only rarely used [16, 17]. This is backed up by Bergman [18], who reports about

1 We use the term *product owner* to summarize the tasks change identification/prioritization and analysis. If the role itself does not exist in a project, its tasks still exist, which we refer to as the tasks of the product owner for the sake of simplicity.

large amounts of dead source code in a software system, after it was developed for several years.

Ron Lichty [19] asks developers, how many features of an imaginary 400 pages requirements document they typically implement. The answers are that usually only 15% to 25% of the contained requirements are actually delivered to the customer. However, the delivered systems were accepted by the customers. The unimplemented requirements either were not necessary for the customers, or their absence remained unnoticed. However, we expect the customers in our domain to know, which functionality is necessary and not to accept a system with requirements that are necessary, but not implemented. Therefore, we conclude, 75% to 85% of the requirements were not necessary for the customers.

These anecdotes indicate a risk of wasting resources during development, and furthermore, that product owners and developers have problems understanding what the actual requirements of the customers are. This anecdotal evidence focuses primarily on the development of software. In this thesis, we focus on maintenance, which is performed after the initial development. But, we expect similar challenges in software maintenance, due to the anecdotal evidence given above.

1.2.2. From Literature

Heiskari and Lethola report general challenges in software maintenance and emphasize the need for information about users [20]. They collected, among others, the following concerns from practitioners in software maintenance:

- "There is too little user information."
- "There is no feedback from outside the house during development."
- "User information is scattered, unorganized, and difficult to access."

One consequence of these challenges is that product owners and developers do not accurately understand the requirements of the users, and therefore risk developing software systems that do not fit the users' requirements.

Begel and Zimmermann report some of the most essential questions of program managers, developers and testers [21]. They emphasize the aforementioned concerns. Among these questions, some emphasize the need of product owners, developers, and testers for knowledge about the usage of software systems.

- "How do users typically use my application?"
- "What parts of the software are most used and/or loved by my customers?"
- "What are the common patterns of execution in my application?"
- "How well does test coverage correspond to actual usage by our customers?"

All questions reported by Begel and Zimmermann [21] are concerned with the actual usage of software systems. The last question also addresses software systems in their testing environment by mentioning test coverage. With the answers to these questions, product owners and developers can gain more insights into what the users actually do with their software systems and adapt maintenance accordingly. Testers can direct their testing efforts more towards a realistic user behavior.

In agile development, the users are closely involved in the development or maintenance process. Thus, at least partial answers to these questions can be obtained directly from them. Therefore, agile methods are one possibility to prevent these questions. However, the studies mentioned above show that these question do arise in practice. One reason is that not every software development or maintenance project is conducted agile [22], since agile methods do not fit on every project, especially on large projects [23, 24].

Since we do not expect these questions to often arise in projects using agile methods, we concentrate on maintenance projects, which do not use agile methods or involve the users closely. However, there is evidence that obsolete requirements are still implemented also in agile development projects [25, 22]. These obsolete requirements can lead to the questions presented above.

1.2.3. Observations in Practice

Motivated by the challenges and questions described above, we conduct studies in practice about usage and coverage of business information systems, which underpin the aforementioned issues.

Maintenance Our own observations underpin the concerns and questions described by Heiskari, Lethola [20], Begel, and Zimmermann [21]. In our studies, we found that almost 25% of the source code of an industrial business information system were not executed in a period over two years [1]. Unused functionality does often not provide value to the users, since if it would provide value to the users, they would use it [26, 27]. We consider functionality, which does not provide value to the users, in the context of custom developed business information systems, as useless. Useless functionality is caused by, e.g., developers, who are sometimes drawn towards implementing functionality they perceive as nice and beautiful, while forgetting their focus should be on providing value to the users [28]. Another example is obsolete functionality that remains in software systems, because developers do not remove it.

Maintenance in unused, and therefore potentially useless functionality might not provide value to the users. In a study, we show that almost half of the maintenance effort spent on unused functionality of a business information system was a waste of resources [1]. This problem is intensified by the fact that software maintenance accounts for 60% to 90% of the overall life-cycle costs of software [29, 30, 31]. One reason for modifications in unused functionality are adaptive and preventive maintenance, which account for 25% to 52% of all changes [32, 33, 34]. Adaptive and preventive maintenance is performed in source code, often regardless of the functionality it implements, as described in the examples given above for adaptive and preventive maintenance (see Section 1.1). Thus, these types of maintenance do not necessarily target functionality that is useful to the users. Therefore, we expect maintenance to occur in source code implementing unused and useless functionality, too.

Building on estimations of usage and usefulness, product owners decide which changes are implemented in which order. However, their estimations might be wrong. Juergens et al. report that, in their study of a business information system, deviations of the expected usage and actual usage occur in 40% to 53% of all features. In 70% of unused features, the stakeholders did not know that the features were not used at all [35]. This shows that product owners have a lack of knowledge about the usage and usefulness of the functionality of the software systems they are responsible for.

But what is the origin of these deviations? One reason are the different areas of expertise of product owners, developers and users. This bears the risk that the product owner does not fully understand what the users require [36, 37, 38, 39, 40]. Therefore, product owners potentially waste resources due to missing knowledge about actual usage. Our own findings confirm this, as they show that almost half of the maintenance in unused code was a waste of resources [1].

Regression Test The test engineer is responsible for coordinating the testing activities. Among his tasks is selecting regression test cases, with the goal of revealing the faults in the system under test. A common strategy is to select test cases that cover modifications. The rationale behind this is that recently added or changed source code is more likely to contain faults than other source code [41, 42, 43, 44]. Also, in our own studies, we found that fault rates in changed, but untested source code are higher than in other source code [2]. Our findings confirm the hypothesis of the aforementioned approaches, that concentrating on modified parts of the source code during testing is sensible.

However, these approaches assume that the changes made to the software system under test are known to the test engineer, and the test engineer has knowledge about which source code is covered by tests. Regression testing is often performed by dedicated testing teams that are often separated from development teams. Therefore, the test engineers may not be aware of all changes in the software. Additionally, there is often no tool support for the test engineer to collect coverage data for system tests. Consequently, maintenance and testing can be aligned badly, and changes done in maintenance can be missed in regression testing. We confirm this fact in our own study, where we found that significant parts of changed source code were not covered during regression testing [2]. As a consequence, faults introduced by maintenance can remain in the system and occur in the productive environment of the software system.

1.3. Problem Statement

The challenges described above indicate that the usage and coverage of software systems are of interest to product owners, developers, test engineers, and testers. Therefore, we formulate the hypothesis, that it is helpful for these roles, to gain insights into usage and coverage.

Unused functionality often does not provide value to the users. However, in maintenance projects, which do not involve users closely, e.g., by applying agile methods, product owners lack knowledge about the actual usage of the software systems they are responsible for [35]. Maintenance in useless functionality can be a waste of resources. Therefore, the product owner needs insights into the actual usage of the software system he is responsible for to prevent a waste of resources. The test engineer needs insights into regression test coverage, since otherwise, expensive faults introduced by adaptive preventive maintenance are more likely to be missed by regression testing.

However, resources and budgets of product owners and testers usually are constrained. Therefore, providing plain insights into usage and coverage is not enough for helping practitioners. For the information to be helpful, its collection and analysis must not impose additional efforts, must be accurate and complete, and must not interfere with existing software systems and processes.

In agile development, the insights the product owners, developers, test engineers, and testers are asking for, are given by frequent meetings of users or customers with the developing stakeholders. However, there is still the risk of obsolete functionality remaining in a software system, or extra functionality not providing value to the customer [25, 22]. In this case, insights into actual usage of a software system are necessary, since some information becomes only apparent, when the system is in productive use.

But how can product owners gain insights into the usage and test engineers into the coverage of software systems in a lightweight and minimal invasive way?

1.3.1. Execution Profiles

Our approach to provide insights into the usage and coverage of business information systems are execution profiles. They express which methods, in an object oriented sense, of a software system were executed in a given period of time[2]. To keep the negative impact on runtime performance low, execution profiles are collected automatically by ephemeral profiling [45]. This technique yields accurate data, and does not require direct interaction with the users of the software system.

Execution profiles can be collected in all environments the considered system is executed in. In the productive environment, real users use the system. The execution of methods shows, that they implement functionality that was used. Therefore, execution profiles, which are collected in the productive environment, allow for gaining insights into the usage of a software system. Therefore, we call execution profiles collected in the productive environment *usage data*. With usage data, product owners and developers can gain insights into what users actually do not use, and coordinate their maintenance efforts accordingly, to prevent a waste of resources.

If execution profiles are collected in the testing environment of a software system, they show, which methods were executed during testing. Therefore, execution profiles collected in the testing environment show test coverage. Thus, we call execution profiles collected in the testing environment of a software system *coverage data*. With coverage data, test engineers can assess which parts of the source code were not covered, and consequently, not tested. With this knowledge, they can close gaps in their test coverage.

1.3.2. Research Objective

Motivated by the concerns of Heiskari and Lethola [20], the questions reported by Begel and Zimmermann [21], and our own observations [1, 2], and by a gap in literature of how to answer the questions and deal with the challenges, we formulate the research objective for contexts, in which users are not involved closely in the maintenance process:

We explore the benefit of analyses of execution profiles collected for business information systems in software maintenance and test.

1.4. Contributions

We use execution profiles to provide information about the actual usage of a software system to the product owner, and information about regression test coverage to the test engineer.

Figure 1.2 illustrates the overview of contributions. We structure the contributions in the activities maintenance and test. The contributions present studies to gain insights into the actual execution of source code and its implications, approaches that transfer execution profiles to other artifacts, and processes that allow product owners and test engineers to consider execution profiles.

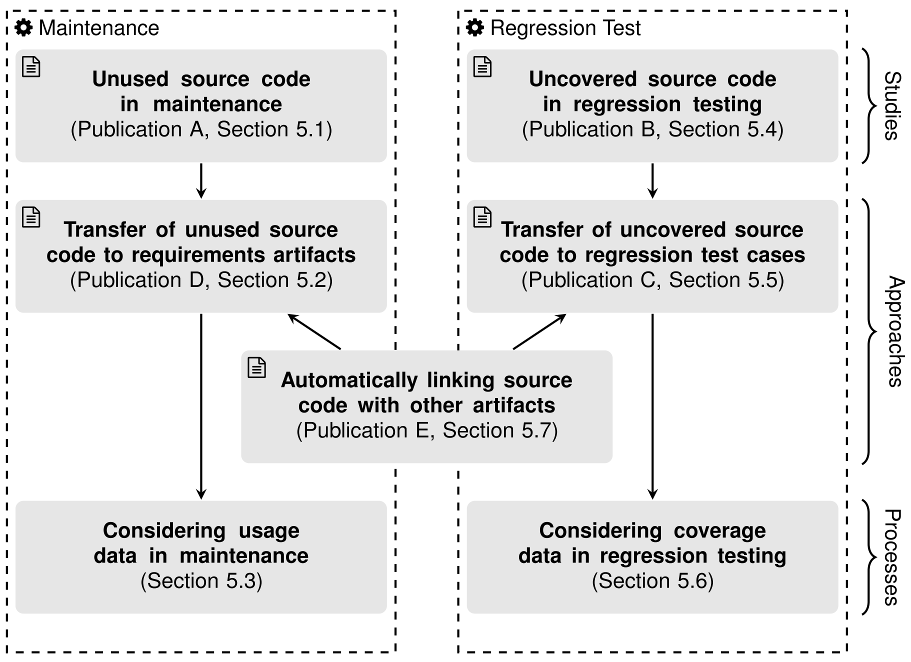


Figure 1.2.: Overview of the contributions.

1.4.1. Provide Insights into the Actual Usage of a Software System to Product Owners

Published in [1] at ICSE 2012
acc. rate 16%
10 pages
Full paper

Unexecuted Source Code in Maintenance This contribution confirms in a case study, that maintenance in unused source code can be a waste of resources, and therefore lays a foundation for this thesis.

The contribution quantifies the effects of maintenance in source code that was never executed by users. In a case study, we examine the extent of unused source code in a business information system and quantify the modifications in unused code performed due to maintenance. Building on this information, the contribution quantifies the amount of maintenance in unused code, which provided no value for the users. Consequently, the study shows, which share of costs could have been saved by considering usage indicated by execution profiles. In addition, we qualitatively examine the helpfulness for developers for preventing maintenance, which does not provide value to the users.

The results of the case study conducted in this contribution show that, in the examined software system, 25% of the source code was not executed over a period of four years. There, 48% of all maintenance actions in unused code were a waste of resources. Examples for maintenance in unused code that were not considered as a waste of resources, was maintenance in error handling code or code that was prepared for future use. The developers wanted to know which source code was unused, which corresponds to the questions reported by Begel and Zimmermann [21] described above. Due to the high probability of maintenance being a waste of resources, the developers found the insights gained into the usage of the software system under development helpful in practice.

As 7.6% of all maintenance actions fell in unused code, 3.6% of all maintenance actions provided no value to the users.

2 Execution profiles do not express *how often* methods were executed during the time interval.

The system under consideration was maintained by nine to sixteen developers[3]. Assuming 220 days of work per person and year, this means that eight days per person and year could have been saved. Scaled to the persons working on the maintenance of the system under consideration, the savings lie between approximately four (in case of nine developers working in the system) and seven (in case of sixteen developers) person months per year. The developers of the system perceived this amount of waste considerable and reported, that they found it beneficial if these resources could be saved or allocated in different projects. But, practitioners state that these savings should be considered only in projects where they enable a different planning of resources.

Published in [4] at ICSME 2014
acc. rate 36%
5 pages
Full paper

Transfer of Unused Source Code to Requirements Artifacts This contribution presents a constructive approach to transfer execution profiles to requirements artifacts by detecting links between source code and requirements artefacts, and an evaluation of this approach.

The latter contribution focuses on source code, since execution profiles also reside on this level. In contrast to developers, product owners often do not know the source code of the software system's they are responsible for. Therefore, product owners often do not understand information, especially execution profiles, that reside on the level of source code. But, product owners usually know the functionality the system provides.

This contribution presents a technique to map unused source code identified by execution profiles to use case documents, which express functionality. For this mapping, we use Latent Semantic Indexing [46] (LSI) from the field of Information Retrieval to detect semantic links between source code and requirements documents. With our technique, the product owners are able to identify use cases, which were not performed by any user during a given period of time. We furthermore show, that the technique's exactness is good enough for being used in practice. Its average precision lies at 0.89.

Considering Usage Data in Maintenance In this contribution, we describe in which phases of the maintenance process the product owners and developers use the aforementioned techniques. Additionally, we give a detailed explanation of how product owners can prioritize changes and file new change requests based on execution profiles. With the resulting change requests, unused source code and functionality can be reduced.

1.4.2. Provide Insights into Regression Test Coverage to Test Engineers

Published in [2] at AST@ICSE 2013
acc. rate 45%
4 pages
Short paper

Unexecuted source code in regression testing This contribution confirms in a case study, that the fault rate in untested, and maintained source code is high, and thus, lays a foundation for this thesis. It confirms the assumption that it is sensible to test modified source code. Additionally, this contribution provides a metric to assess the alignment of maintenance and testing.

This contribution quantifies the effects of a bad alignment of maintenance in source code and regression testing. In a case study, this contribution examines the extent of changes in source code that were not covered during regression testing. It furthermore measures the frequency of faults occurring in the productive environment of a software system in the modified, but uncovered source code. Motivated by

3 The number varied over time.

the fact that there are more field bugs in modified, but uncovered source code, this contribution also provides a metric that is suitable for assessing the alignment of maintenance and regression testing by quantifying the amount of uncovered changes in source code.

In the system under examination, one third of all methods remained untested, about 8% of all methods were changed, and about 44% of the changed methods were executed by at least one regression test. The untested and changed methods contained about 40% of all field bugs. The probability of was therefore considerably higher faults in untested and changed methods than in other methods.

Published in [3] at AST@ICSE 2014
acc. rate 43%
7 pages
Full paper

Transfer of Uncovered Source Code to Regression Test Cases This contribution presents a constructive approach to transfer gaps in coverage, measured by execution profiles, to regression test cases, and an evaluation of this approach.

Often, test engineers and testers are employed in different departments than the developers. Therefore, they are often even physically separated from each other. This impacts the communication between test engineers, testers, and developers negatively. Therefore, test engineers usually do not know the source code of the system under test and which regression test cases test which source code. Thus, test engineers often do not know which test cases to execute to cover formerly untested modifications in source code.

This contribution provides a technique for test engineers to identify existing regression test cases that cover untested source code without executing the test cases first. For this approach, we make use of LSI to detect semantic links between source code and test cases statically without executing test cases. We evaluate the accuracy of the approach in an industrial case study and show its applicability in practice.

In 90% of all cases, our approach suggested a test case for a method, which really executes the method, while suggesting 1.75 in average of four possible test cases per method.

Considering Coverage Data in Regression Testing In this contribution, we describe in which phases of the maintenance process the test engineer uses the aforementioned techniques. Additionally, we give a detailed explanation of how test engineers can assess the alignment of maintenance and regression testing and how they select regression test cases using execution profiles.

1.4.3. Automatically Linking Source Code and Other Artifacts

Published in [5] at RET@ICSE 2015
acc. rate NA
7 pages
Full paper

This contribution provides a constructive approach for detecting semantic links between artifacts fully automatically and thus fosters, technically, the applicability of the aforementioned contributions.

Two of the aforementioned contributions connect execution profiles residing on source code with use case documents and test cases. In both cases, we use LSI to identify semantic links between source code and the other artifacts. However, the accuracy of this technique heavily varies depending on its configuration [47, 48, 49, 50, 51, 52, 53].

Therefore, we propose a technique to configure LSI automatically, to enable a fully automatic identification of semantic links between software development artifacts. In a case study, we show the accuracy of the technique. With this technique, product owners and test engineers can identify semantic links between execution profiles and artifacts they understand better.

The evaluation shows that, given LSI is able to provide accurate results, our approach selects configurations for LSI that produce accurate results.

1.4.4. Summary

We provide approaches based on execution profiles to reduce *waste* [12] in the maintenance process. Therefore, we pursue goals that are similar to the goals of agile development. However, agile development focuses on frequent face-to-face discussions to gain insights into the actual usefulness of functionality to product owners, developers, test engineers, and testers [11]. Thereby, agile methods focus primarily on preventing the implementation of useless functionality. Our techniques allow for a retrospective identification of useless features, in contrast to agile methods. Therefore, our contributions target mainly non-agile maintenance projects.

One reason for employing agile methods are frequently changing requirements. Requirements can also become obsolete, after their realization. Then, the implemented, but obsolete functionality remains in the software system. With our approaches, we detect also this functionality.

In the case, agile development processes are employed, we accompany discussions with the proposed approaches. Thereby, we support the discussions with data about the actual execution, usage, and coverage. If traditional development processes like the waterfall model are in use, we give first insights into execution, usage, and coverage to product owners, developers, test engineers, and testers.

1.5. Overview

We start with an explanation of the key concepts in this thesis in Chapter 2. We focus on the activities software maintenance and testing in the context of the software maintenance process, and on user involvement in this process. We furthermore present our notion of usefulness of functionality and why this functionality is likely to be used. In Chapter 3, we describe related work. We focus on general challenges in the software maintenance process, and on how users are involved into the software maintenance process. After that, we characterize execution profiles in detail. Therefore, we describe in Chapter 4, how execution profiles are collected, their advantages, and which conclusions we can draw from them. With the foundations of the work explained, we present the contributions of this thesis in Chapter 5. The thesis closes with Chapter 6, where we summarize the work, present limitations of the contributions and future work. We furthermore formulate the key takeaways for this thesis.

CHAPTER 2

Background

Contents

2.1. Software Maintenance Process 12
2.2. Software Maintenance 13
2.3. Software Test in Maintenance 16
2.4. The Gap between Developers, Product Owners, and Users . . . 21
2.5. Software Usefulness and Usage 23
2.6. Summary 25

This section discusses the fundamental terms and concepts of this thesis. We first describe the software maintenance process, focusing on the contained activities software maintenance and test. We separate between the software maintenance process and the activities maintenance and testing. We thereby focus on the stakeholders who are responsible for the tasks that have to be done in these activities: product owners, developers, test engineers, testers, and users.

In practice, we often notice a gap between product owners, developers, test engineers, testers, and users. This results in product owners, developers, test engineers, and testers having difficulties understanding the users of the software systems they are responsible for. We explain reasons for this gap.

Furthermore, we describe our notion of usefulness of functionality for users, and the consequences of usefulness for the actual usage of functionality.

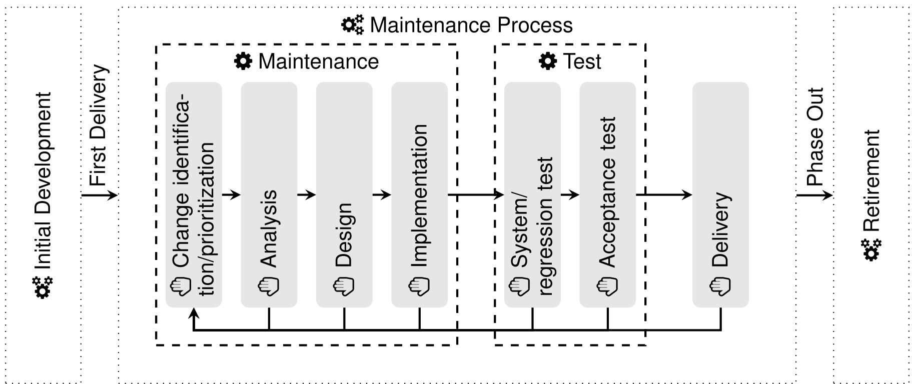


Figure 2.1.: Simplified view of the software life-cycle and the maintenance process.

2.1. Software Maintenance Process

The users, environments, and business models of business information systems change over time. Long living custom business information systems are in productive use often over many years, or even decades. This means for a software system, in order to stay useful for its users, it has to be adapted to the aforementioned changes. The changes made to the software system are performed in software maintenance, which is organized in the software maintenance process. In this section, we describe the software maintenance process in detail.

The software maintenance process comprises several tasks. We follow the definition of the software maintenance process in the IEEE standard 1219 [9]:

As illustrated in 2.1, the maintenance process comprises the following tasks:

Change Identification/Prioritization In this task, changes to the software system are identified, classified and assigned. Changes can be accepted, rejected or further evaluated, and is estimated for the size of the modification and prioritized. The latter means that the changes are assigned their importance relative to each other and by this, their order of implementation is determined.

Analysis The change is analyzed for feasibility and its impact on the system. Furthermore, its scope is identified and a preliminary plan for implementation is created. Thereby, also the prioritization of changes may change.

Design The change to the software system is designed. All artifacts connected to the software system, and all other data collected about the system are to be considered.

Implementation The change to the software system is implemented.

Regression/System Testing The modified system is tested to validate that changed or newly created code does not introduce faults that did not exist prior to the maintenance activity.

Acceptance Testing The fully integrated system is tested by the customer or a third party designated by the customer to confirm the functionality of the software system meets the customer's expectations, the software system does not contain newly introduced faults, and the software system is interoperable within the customer's environment. Additionally, test cases are created or modified.

Delivery The software system (the release) is delivered to the customer. This may include a physical installation, a notification of the user community, archiving the prior version of the software system, and training for the users.

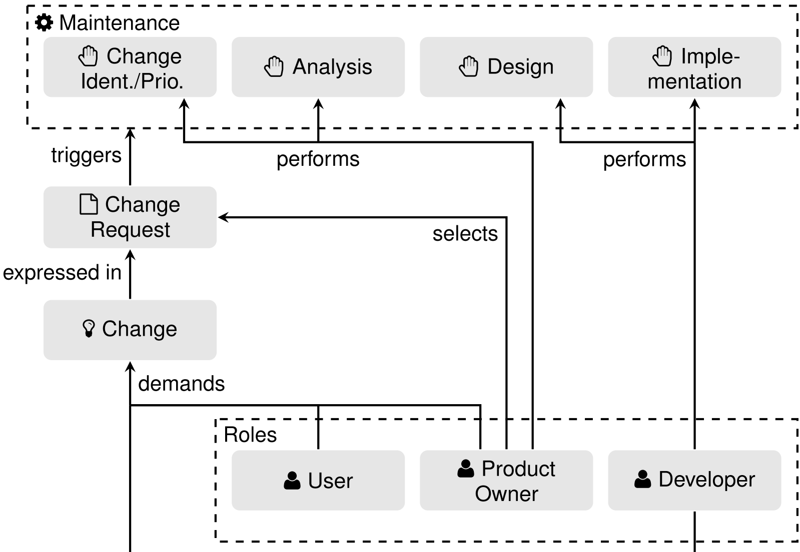


Figure 2.2.: Overview of terms, roles, and tasks in the activity maintenance.

From each of these activities, the need for more changes can arise. One example are faults discovered during regression testing. The need for these faults to be fixed triggers the activity maintenance with its tasks, and test afterwards.

2.2. Software Maintenance

Figure 2.2 gives an overview of the terms, roles, and tasks in the activity software maintenance. In the following, we explain the relations between the roles and tasks, and changes and change requests in the activity maintenance in more detail.

2.2.1. Definition

Software maintenance is the activity of modifying software systems after their first delivery [54, 55, 56, 57] to correct faults in existing functionality, implement new functionality, adapt the system to changed environments, or to prepare them for future changes. Software maintenance is performed after the first delivery of the software system and comprises the tasks *change identification/prioritization*, *analysis*, *design*, and *implementation*, as depicted in Figure 2.1.

We consider artifacts to be maintained, if they were changed or created during the activity software maintenance.

2.2.2. Types of Maintenance

Maintenance is performed due to the users' demand for new or changed functionality, due to changing environments, or new technical advances (e.g., transfer to mobile devices, connect to faster database servers). According to Bennett and Rajlich [10], maintenance can be divided into four categories, along these reasons:

Corrective: Faults in the system are fixed.

Perfective: New functionality is implemented or existing functionality is modified.

Adaptive: The system is adapted to work in a changed environment.

Preventive: The system is prepared for future modifications.

Adaptive maintenance has the goal to preserve existing functionality in a changed environment. Perfective maintenance prepares a system for modifications of its functionality in the future, without changing the functionality itself. Adaptive and preventive maintenance often do not change or add functionality. Instead, if functionality is changed in these types of maintenance, it is often not deliberately.

2.2.3. Involved Roles

In the activity maintenance, we focus on the roles *developer*, *product owner*, and *user*. There can be settings, where the tasks we assign to these roles, are dispersed over different roles. However, we rely on our rather simple assignment of tasks to reduce complexity.

Developer The *developer* is the person who modifies a software system. The modifications include source code and documentation. Therefore, the developer is assigned to the task *implementation*. As the developer also needs to plan his concrete actions to realize his modifications, he is also in charge of the task *design*.

Product Owner The product owner is responsible to maximize the value of the software system.

The tasks of the *product owner* are identify, define, and prioritize changes, design, implement and disseminate reference architectures to developers, choose technologies and tools for development, communicate with customers and developers, give insights into the software systems business domain to developers, assess technical risks, and plan releases [58].

In the software maintenance process, the product owner is, therefore, in charge of the maintenance tasks *change identification/prioritization* and *analysis*. The product owner indirectly coordinates the efforts of developers, because his tasks are concerned with planning the implementation task. In agile development, the product owner should be provided by the customer. In other development and maintenance projects, however, the tasks of the product owner are usually adopted by the developing company.

In other settings, there can also be a *change control board* consisting of the product owner, developers, and architects[1]. Especially in contexts where agile methods are applied, also the customers or users can be part of this board. This board then decides about the prioritization of changes. Additionally, the name of the role can be different or its tasks can be split up and distributed over several persons, especially in non-agile maintenance projects. However, for the sake of simplicity, we summarize these responsibilities under the role of the product owner.

User Besides the developers and product owners, who are directly involved into the software maintenance process, there are the users.

The *user* uses the software system. There, he performs one or more tasks with a software system by executing it in work activities, which we call *usage*. In the context of custom software systems, the user is often the customer. Users are responsible for communicating domain knowledge and changes to the product

1 The personnel may vary between companies and projects.

owner. Therefore, the users do not perform tasks in the maintenance process, but are involved indirectly.

We concentrate on the user in his working environment, and do not consider hedonistic use of software [59][2]. The reason is that the focus of this thesis lies on custom business information systems. These are usually built to help users in their working environment.

2.2.4. Change Requests

Software maintenance is triggered by changes to the software systems demanded by users, developers, or product owners. These changes are expressed in *change requests*. *Change requests* are proposed changes to a product that is being maintained [55].

Not only product owners, but all stakeholders of a business information system can file change requests to express their demand for changes to the system. Common stakeholders filing change requests are the users itself, or the developers [60, 61], and the product owners. However, it is the product owner's responsibility to prioritize changes, and therefore, change requests.

Change requests can, like maintenance, be categorized as corrective, perfective, adaptive, and preventive change requests. The categorization depends on the type of maintenance the change request causes.

2.2.4.1. Reasons for Change Requests

There are several reasons for change requests. The following list is not complete, but gives some examples for the reasons of why a change request is issued. Therefore, these are also reasons for change requests.

New Ideas Users, customers, product owners, or developers have new ideas over what the current system should do, or how it should do it. This leads to perfective and corrective change requests.

Missed Functionality Functionality requested by the users or customers was not implemented before. This leads to perfective change requests.

Misunderstandings Product owners and developers misunderstood the functionality the users or customers demanded. This leads to corrective change requests.

Defects Wrongly implemented functionality needs to be corrected, so that the software system complies with the functionality requested by the users or customers. Defects lead to corrective change requests.

New Software Systems The users or customers get a new software system. They want interfaces to this new system, or features that enable them to combine the software systems, e.g., opening files from one system in the other. This leads to perfective and adaptive maintenance.

2 Playing video games is one example.

Changing Legislations Changing laws can affect which and how functionality is implemented, and lead to new functionality. Therefore, they lead to corrective and perfective maintenance. Also, adaptive maintenance can be caused by changing legislations, since new technical constraints might have to be fulfilled.

Changing Technology As technologies change, e.g., database or presentation technologies, also the software systems needs to be changed accordingly. These changes lead to adaptive change requests.

Subsequent Changes One change in a software system can cause subsequent changes. This can lead to all kinds of change requests. One example are attempted bug fixes that cause new faults in different places.

The reasons above are just examples. In general, change requests emerge in the minds of all stakeholders. However, it depends on the project, which stakeholders are filing them. The various stakeholders are likely to file change requests of different categories:

Users and Customers The users or customers are confronted with the functionality of a software system in their daily business. Therefore, we expect users and customers to file primarily corrective and perfective change requests. However, also users and customers may issue adaptive and preventive change requests. But we expect them to file these less than corrective and perfective change requests.

Product Owners and Developers These stakeholders often have a deeper knowledge about the technical aspects of a software system. Therefore, we expect them to also file adaptive and preventive change requests. Also developers and product owners can file perfective, and especially corrective change requests. However, we expect the product owners and developers to be the main source for adaptive and preventive change requests.

2.3. Software Test in Maintenance

Modifications to a software system possibly introduce faults. To validate that a software system exposes the functionality desired by the users or customers, testing is performed.

2.3.1. Definition

Software testing is the activity of executing a software system under conditions and inputs specified in test cases, and comparing the system's output with the outputs specified in test cases to validate the functionality against the users or customers' expectations [55, 56, 62].

Test cases are sets of inputs, execution conditions and expected results [55, 56, 62]. Test cases are executed by testers or developers. They can be performed manually by testers, or automated, where no manual interaction is required [63].

Test cases are contained in *test suites*, which are sets of test cases [62].

2.3.2. Software Testing Process

The test process consists of the activities test planning, test design, test implementation, evaluating exit criteria and reporting, and test closure activities [62]. The

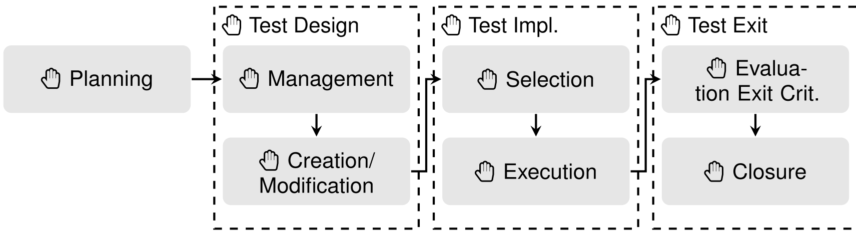


Figure 2.3.: Overview of the testing process.

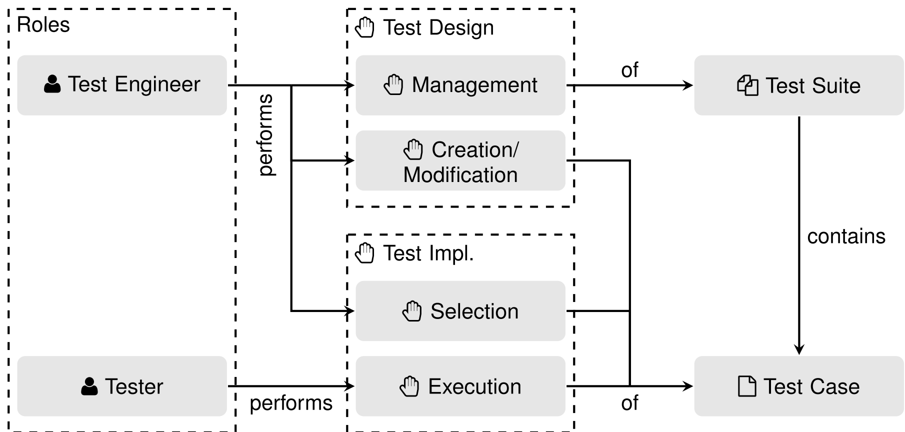


Figure 2.4.: Overview of terms, roles, and tasks in the task system/regression test.

activities are described in detail below and illustrated in Figure 2.3.

Test Planning During this activity, the scope, approach, resources, and schedules for testing are defined. The functionality to be tested is also fixed. The exit criteria for the test process are also defined.

Test Design In this activity, the specification of the functionality to test is reviewed, and the objectives of the test are defined (analysis). Additionally, test cases are created or modified (design). Furthermore, test cases are added to or removed from test suites.

Test Implementation In test implementation, test cases are selected, and test data is created. In execution, the test cases are run under on the system under test.

Evaluating Exit Criteria and Reporting The exit criteria for completing the test process are evaluated, and the results of the tests are reported.

Test Closure The data collected during testing is archived.

Figure 2.4 gives an overview of the terms, roles, and tasks in the task system/ regression test. We concentrate on the roles test engineer and tester, and on their tasks.

2.3.3. Roles

In the testing activity, we concentrate on two roles. The test engineer, who designs and selects test cases, and the tester, who performs the testing.

Test Engineer The tasks of the test engineer are, among others, the *management* test suites by adding or removing test cases, the *creation and modification* of test cases, and the *selection* of test cases for execution.

Additionally, the test engineer is in charge of performing *coverage analysis*, which is the "measurement of achieved coverage to a specified coverage item during test execution referring to predetermined criteria to determine whether additional testing is required and if so, which test cases are needed." [62].

Examples for *coverage analysis* are the investigation of source code that was not tested, or the examination of which functionality was validated against the customer's expectations.

Tester The tester is responsible for the, possibly manual, *execution* of test cases. Additionally, the tester reports the verdict (fail, success, abort) of the test to the test engineer, developers, and product owner, if this is not achieved automatically.

A fault in the system, detected by the tester, can be handled in different ways. First, the stakeholders can ignore it. This can make sense for unfinished functionality, or for functionality that is not about to be delivered to the customer. Second, if the test was wrong, but the system worked as expected by the users or customers, it is the responsibility of the test engineer to modify test cases. Third, in case the test was correct, but the system did not work as expected, the product owner is in charge to file a corrective change request with the goal to fix the fault.

2.3.4. Levels of Granularity

Tests can be performed on different levels of granularity, depending on the scope of the executed test cases. The scope of test cases ranges from very small portions of the source code, like single methods or classes, to the whole system. All definitions given in this section comply with the IEEE standards 24765 (Systems and software engineering – Vocabulary) [55] and 610 (IEEE Standard Glossary of Software Engineering Terminology) [56].

2.3.4.1. Unit Testing

The scope of unit testing are single algorithms, functions, methods, or classes [55, 56]. Developers trigger execution of unit test cases often directly from their development environment. The results are usually displayed within the development environment. The developer is presented the verdict of the test, and also the line or statement coverage. In the case of unit testing, the developer himself fulfills the roles tester and test engineer, since the developer performs the testing by himself in the development environment.

Unit test cases are usually automated and, as described above, this activity is well supported by tools. Reasons for this are the small complexity of the test objects examined during unit testing, and the resulting small complexity of test cases.

2.3.4.2. Component Test

The scope of component testing are functionally distinct software items, or groups of them [55, 56]. For this kind of test, testers execute (groups of) components with inputs on their interfaces, and observe their outputs. However, the level of granularity varies with the notion of the term *component*. In the programming language Java, component testing considers, e.g., all classes in a package. To the extreme, one can see single methods, or whole systems, as components. This kind of testing is closely related to integration testing.

2.3.4.3. System Testing

The scope of system testing is the complete, integrated system [55, 56]. Test cases for system tests are often written in natural language and testers perform them manually. The manual execution renders this kind of test rather expensive. However, there are also automated system tests.

Modern business information systems often comprise a plethora of functionality. Consequently, system test cases are rather complex. This complexity leads to expensive initial development maintenance for automated system test cases. Therefore, in practice, system tests are often not automated in practice.

The tool support for system testing is restricted to reporting verdicts and managing or creating test cases, and does not allow for collection of coverage information. On reasons for this is the large negative impact on run-time performance caused by common coverage collection tools. Therefore, test engineers usually have no insights into the coverage of their system tests.

2.3.5. Goals of System Testing

Besides the granularity of tests, also the goals vary. We distinguish four goals:

- Detection of faults in existing functionality
- Determination of the satisfaction of user acceptance criteria
- Evaluation of the interaction between combined components
- Evaluation of performance

2.3.5.1. Regression Testing

The goal of regression testing is the detection of new faults in existing functionality prior to the delivery of the software system [64].

These faults were possibly introduced due to modifications to the software system. According to our experience, in practice, the only regression tests that are performed, are system tests, and automated unit tests. The tool support for developers, testers, and test engineers for developers and testers in unit tests is well developed. Therefore, we focus on system tests in the remainder of this thesis.

Regression testing only considers existing functionality. For new or changed functionality, usually no regression test cases exist. Therefore, regression testing is suitable after adaptive and preventive maintenance, since they often do not change the system's functionality.

In this thesis, we always refer to the activity regression testing when we use the terms testing or test.

2.3.5.2. User Acceptance Testing

The goal of user acceptance testing is to determine whether a system satisfies its acceptance criteria and to enable the customer to determine whether to accept the system [55, 56].

Before a software system is delivered to the customers, they test it for acceptance. Acceptance has two facets: rather formal criteria that are fixed prior to the test, and more informal aspects like usability or graphical design issues that also can prevent the customer from accepting a system.

For new functionality, where often no test cases exist, the customers are asked to perform exploratory tests to ensure the new functionality conforms to their change requests. From these exploratory tests, test cases can be derived. These new test cases can then be added to the collection of regression tests.

In the case of user acceptance testing, it is the users' task to perform test cases. If testing pursues different goals, as described below, it is the tester's task.

2.3.5.3. Integration Testing

The goal of integration testing is to evaluate the interaction among combined software and/or hardware components [55, 56, 62].

Integration tests also validate functionality, but focus more on the interfaces between components. Additionally, in integration tests, not the whole system is tested necessarily, but groups of components (see also component tests in Section 2.3.4.2).

The tested components interact according to the system's architecture. Therefore, also the architecture of a software system is validated during integration testing.

2.3.5.4. Performance Testing

The goal of performance testing is to evaluate the performance[3] of a system or component [55, 56].

This kind of test does not consider directly what a system does, but how it does it. However, to gain the desired efficiency, the source code of the software system might change towards faster or less resource intensive implementation. The user visible functionality stays the same, except, e.g., the response time. This does often not influence the data observable at the system boundaries.

2.3.6. Regression Test Case Selection

Regression testing is a necessary, but expensive activity in maintenance [65, 66, 67, 68]. Therefore, executing all regression test cases in a test suite may consume many resources [65]. Above that, there may be insufficient resources to execute all regression test cases [67]. Therefore, test engineers choose a subset from the regression test suite, which is to be executed. Their goal is to detect faults introduced by the modifications to the software system in the existing functionality, while not exceeding their resource budget.

Test engineers apply regression test case selection strategies to choose test cases to execute from test suites [66]. Thereby, test engineers focus on testing changed functionality [66], since they suspect more faults in this functionality. Several studies have shown that this assumption is valid [2, 41, 42, 43, 44].

2.3.7. Test Coverage

We define coverage along the existing definition: "The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite." [62]. More exact, we concentrate on *code coverage*, which "determines which parts of the software have been executed (*covered*) by the test suite and which parts have not been executed." [62].

3 Which can be, e.g., resource, or runtime performance.

Note that we do not define *coverage* in relation to functionality, as other definitions do, e.g., "extent to which the test cases test the requirements for the system or software product." [55]

Beneath others, there are several common coverage metrics measuring coverage, and are defined in [62]. We describe these metrics below.

Statement Coverage The percentage of executable statements that have been executed by a test suite.

Branch Coverage The percentage of branches that have been executed by a test suite. 100% branch coverage implies both 100% decision coverage and 100% statement coverage.

Decision Coverage The percentage of decision outcomes that have been executed by a test suite. 100% decision coverage implies both 100% branch coverage and 100% statement coverage.

Condition Coverage The percentage of condition outcomes that have been executed by a test suite. 100% condition coverage requires each single condition in every decision statement to be tested as True and False.

Decision Condition Coverage The percentage of all condition outcomes and decision outcomes that have been executed by a test suite. 100% decision condition coverage implies both 100% condition coverage and 100% decision coverage.

Modified Condition/Decision Coverage The percentage of all single condition outcomes that independently affect a decision outcome that have been executed by a test case suite. 100% modified condition decision coverage implies 100% decision condition coverage.

Path Coverage The percentage of paths that have been executed by a test suite.

Method Coverage In contrast to the aforementioned coverage metrics, we consider coverage on the level of methods[4]. Method coverage is the percentage of methods, in an object oriented sense, that were executed at least partially by a test suite [2]. It does not imply that all statements, branches, decisions, conditions, nor paths of a method were executed by a test suite.

2.4. The Gap between Developers, Product Owners, and Users

It is difficult for product owners and developers to understand the users of the software system under development in the software maintenance process [36, 37, 38]. As a consequence, product owners and developers face the risk of realizing or maintaining useless functionality, which is a waste of resources.

4 In an object oriented sense

But where do the difficulties come from? One reason is a communication gap between product owners, developers, and the users [39, 69]. In this section, we focus on reasons for this gap.

We consider the gap between the technical background of the developers and product owners, and the domain background of the users, as illustrated in Figure 2.5. As it is the product owner's task to communicate with users (see Section 2.2.3), we do not consider communication between developers and users. Note that we consider maintenance projects, which do not apply agile methods. In agile methods, the product owner should be provided by the customer, who also is the user in our context. Therefore, we place the product owner on the side of the developers, and not of the users or customers, which would be the case in agile contexts.

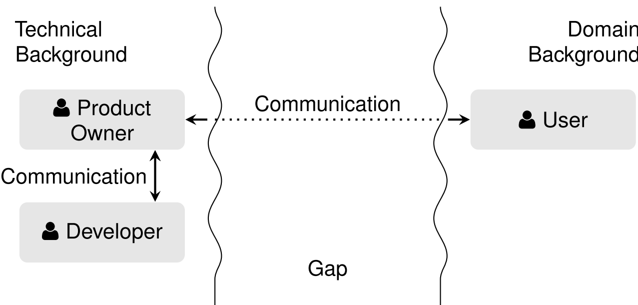


Figure 2.5.: Gap between users, developers, and product owners.

The gap between product owners, developers and users can be categorized into several dimensions [70, 71]. Each of the dimensions has its own reasons for the gap between developers and users to arise.

Perspective Gap: Users might forget that the goal of a system is to serve them and technology is a tool, not an end in itself. Developers and product owners forget that the software system has to provide value for the customers and the developing organization is not the center of the universe [70].

Ownership Gap: Developers and product owners feel that the software and its infrastructure belong to them, while the users feel ownership over the business processes implemented in the software system. This leads to developers and product owners seeing users as reactionists and users perceive developers and product owners as technical elitists [70].

Cultural Gap: Users, developers, and product owners have different values, working behaviors, or priorities. Developers tend to be introverted, analytical, and rational, while users (in a business context) are more extroverted and intuitive [70].

Foresight Gap: Users, developers, and product owners have different capabilities and strategies to make predictions for the future, but are unable to communicate their predictions. Users can foresee better that a proposed solution will not work for them, while developers and product owners have more expertise in predicting technical feasibility and ways of implementing software systems [70].

Communication Gap: One group does not understand what the other group tries to express. One reason for this is different jargon used by each group. The reliance on specifications imposes furthermore a wall between users, developers, and product owners [72]. The reason for this is that specification documents are often handed over from the customer to the product owner and then, the

customers assume the developers have all information that is necessary to develop the software system as intended by the customers [70].

Expectation Gap: Users have unrealistic visions of what the developers are able to do. Reasons are growing technical expertise of users, heroic efforts of developers for delivering software, and developers making overblown claims to what they can deliver [70].

Credibility Gap: The past performance of developers and product owners did not meet the expectations of the users, due to failed projects, or poor customer service. Because of this gap, product owners find user to be overly demanding or unwilling to change [70].

Appreciation Gap: Users, developers, or product owners feel not appreciated by the other group. Developers and product owners feel their hard work goes unappreciated, except of failures [70].

Relationship Gap: Users and product owners do not interact frequently enough to develop a constructive relationship. This can be reinforced by prejudice about the other group [70].

Only some dimensions of the gap between developers and users lead to misunderstandings between product owners and users. However, the perspective, ownership, cultural, foresight, and especially the communication gap lead to product owners understanding the users wrong.

Summarized, the reasons for the gap between users, product owners, and developers, are their different areas of expertise and organizations, and their background. Users have domain knowledge, and know[5], which problems should be solved and which functionality they desire [73]. However, with modern of software systems providing more and more functionality, users do not always know, which functionality they need [74]. Product owners and developers, on the other hand, usually have rather technical knowledge about a software system.

As a result, users, product owners, and developers have different expectations and understanding of the functionality, which a software system has to provide.

2.5. Software Usefulness and Usage

The gap between product owners, developers and users leads to misunderstandings and different expectations of which functionality should be implemented and maintained in the software maintenance process. From the perspective of the users, who are often the customers in our context, *useful* functionality should be implemented and maintained. However, due to the aforementioned gap, developers and product owners can have different expectations about the usefulness of functionality. But what is *usefulness*? In this section, we explain our notion of the terms *usefulness* and its connection to *usage* (see Section 2.2.3).

2.5.1. Usefulness

Legris et al. [7] characterize the term *useful*, and accordingly *usefulness*. They report, that users perceive a software system as useful, if:

- "Using (the system) increases my productivity." [7]
- "Using (the system) increases my job performance." [7]
- "Using (the system) enhances my effectiveness on the job." [7]

5 Possibly, only after they have seen prior versions of their software system

- “Overall, I find the (system) useful.” [7]

The notion of Legris et al. [7] of usefulness comprises complete software systems. However, recent software systems tend to provide lots of user visible functionality. We therefore transfer the notion of Legris et al. [7] from whole systems to user visible functionality. This results in our notion of the terms *useful* and *usefulness*, where using user visible functionality provided by a software system increases the job productivity, job performance, effectiveness on the job of the users. Thus, we consider functionality as *useful*, if the users perceive it as being useful.

However, this notion is subjective. In our context, the users of a software system are specialized in their domain, and the software systems are customized for them. Therefore, we expect users to know and to agree on which functionality is useful for them, at least, once they see it.

2.5.2. Connection of Usefulness and Actual Usage

In this section, we describe that usefulness induces software usage. Literature describes, that the usage of software systems does correlate with what is perceived useful by the users [7, 27, 75, 76, 77]. This literature is based on the Technology Acceptance Model [26, 27] (TAM), as illustrated in Figure 2.6, even though there are several models explaining the usage of software systems by users [78].

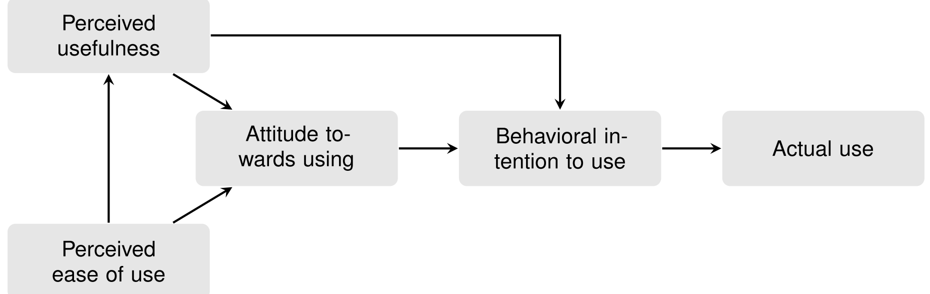


Figure 2.6.: Technology Acceptance Model (TAM) [26, 27, 78].

The TAM was shown to be suitable for explaining why users actually use software [78], while being simple. The TAM explains most of the reasons for users using software. Therefore, we also rely on this model, but adapt it to user visible functionality, not whole software systems.

According to the TAM, actual usage of functionality, which is the act of executing functionality, has two antecedents: *Perceived ease of use* and the *perceived usefulness* of the potential users of the functionality. The actual *behavioral intention to use* functionality is determined by the *attitude towards using* the functionality and its *perceived usefulness*. The *behavioral intention to use* then motivates the *actual use* [78].

Several studies show, that the influence of the perceived usefulness is higher on the actual usage than the influence of the perceived ease of use [7, 27, 75, 76, 77]. Especially, these papers show a statistically significant positive correlation between the perceived usefulness and the actual usage of functionality. That means if the perceived usefulness is high, the actual functionality use is also high.

But reversed, but also if the actual functionality use is low, also the perceived usefulness is low. The latter means that if functionality is not used, it is likely to be little useful in the perception of the users. Figure 2.8 illustrates the latter relation.

The TAM does only consider intrinsic, and not extrinsic motivation of users. Therefore, it does not explicitly cover usage caused by external factors like company

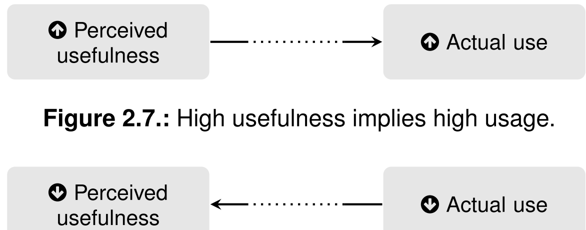


Figure 2.7.: High usefulness implies high usage.

Figure 2.8.: Low usage implies low usefulness.

guidelines. However, according to Taylor and Todd [78], the TAM still explains most of the actual usage, which means that extrinsic motivation plays a smaller role in the usage of functionality.

Due to the specialized users, who know what is useful for them, in our context, and the indicators from literature, we conclude that the usefulness of functionality correlates with its usage.

We are aware, that this model does not present logical implications, but correlations. Therefore, there is no strict implication from usefulness to usage or vice versa. For example, a user might perceive functionality as not useful, but uses it anyways as a substitute, because the functionality he actually needs is not implemented. Another example is, that a user does not use functionality he perceives as useful, because he does not know about it. But, as several studies show, the model explains most of the actual usage of software [27, 7, 75, 76, 77].

2.6. Summary

For gaining insights into the benefit of execution profiles in maintenance and test, we first explained the background of this thesis. We considered the software maintenance process, and the contained activities maintenance and test. In testing, we focused especially on regression testing.

Furthermore, we explained the reasons, why product owners, developers, test engineers, and testers are interested in usage and coverage. The main reason, we identified was a communication gap between users and the aforementioned stakeholders. This gap arises due to different areas of expertise of the users and the other stakeholders. The gap results in product owners not fully understanding what users require, and in test engineers, who do not know, how to align their test cases to the usage patterns of the users.

The goal of product owners, developers, test engineers and testers is to better understand the users. As these stakeholders ask for more knowledge about users, we investigated the connection between usefulness for users, and usage.

CHAPTER 3

Related Work

Contents

3.1. Challenges in Software Maintenance 28
3.2. Involving Users in the Software Maintenance Process 29

In this chapter, we summarize the related work. We thereby concentrate on the challenges and questions in software maintenance mentioned in Chapter 1, that motivate this thesis. We describe them in detail in the first part of this chapter.

These challenges arise due to difficulties in the communication, a gap, between product owners, developers, test engineers, and testers, as described in Section 2.4. But how can we close this gap? The answer of current research is to involve users in the software maintenance process. Therefore, we focus on user involvement in the software maintenance process in the second part of this chapter. Thereby, we focus on how we can gain knowledge about the actual behavior of users to provide knowledge about the actual usage of a software system.

3.1. Challenges in Software Maintenance

As a consequence of the gap between product owners, developers, and users (see Section 2.4), the risk of building a system that does not provide functionality that is useful for the users arises [79, 80]. In this section we describe challenges for product owners, developers, test engineers, and testers reported in literature. These challenges show, that the aforementioned roles are interested in closing the gap to the users.

For closing the aforementioned gap, product owners, developers, test engineers, and testers need to understand the users. However, this imposes challenges, which we describe first in this section. Additionally, literature reports about questions about users, which product owners, developers, test engineers, and testers want to have answered. We explain these questions in the second part of this chapter.

3.1.1. Understanding the User

Heiskari and Lehtola [20] report about challenges during the software maintenance process. These challenges are relate to knowledge about the users using a software system.

- "There is too little user information." [20]
- "There is no feedback from outside the house during development." [20]
- "User information is scattered, unorganized, and difficult to access." [20]
- "What does the customer actually value?" [20]
- "There is very little interaction between the end users and the development organization." [20]
- "How to integrate user knowledge to the existing processes?" [20]
- "No clear processes on understanding the user exists." [20]

These challenges are concerned with gaining knowledge about the users, understanding them, and interpreting knowledge about them. Therefore, the challenges show the difficulty for product owners, developers, test engineers, and testers of understanding the users' correctly.

The product owner's tasks include communicating with users and developers to transfer domain knowledge to the developers. Additionally, the product owner is in charge of identifying and prioritizing changes. The challenges mentioned above therefore concern the product owner, since he faces the difficulty of understanding the users.

There are some other challenges, according to Heiskari and Lehtola [20], e.g., "The big picture needs to be understood before going into details" [20]. However, these target aspects of how to treat knowledge about the users and are not directly related to the software maintenance process. Therefore, we consider these to be out of scope of the thesis.

3.1.2. Questions about Users

In a recent study, Begel and Zimmermann [21], acknowledge the challenges identified by Heiskari and Lethola [20] and report 145 questions, product owners, developers, test engineers, and testers want to have answered. Under the top ten of the most essential[1] questions are four questions that are of interest to all aforemen-

1 As classified by Begel and Zimmermann [21]

tioned roles. All of these questions are concerned with the usage and execution of software systems.

- "How do users typically use my application?" [21]
- "What parts of the software are most used and/or loved by customers?" [21]
- "What are the common patterns of execution in my application?" [21]
- "How well does test coverage correspond to actual usage by our customers?" [21]

All questions consider the usage of software systems. One mutual goal of the questions is to understand what functionality is actually used by users, and how the functionality is used. In this thesis, we concentrate on the question, which functionality is used by the users.

The questions relate to the *change identification prioritization*, *analysis*, and *design* tasks. In these tasks, the product owner uses the answers to these questions to prioritize and identify changes regarding the usage. All aforementioned tasks have an impact on which functionality, and consequently, which source code is maintained. Therefore, these questions also affect the *implementation* task of the developer. With the answers to these questions, product owners and developers can reduce the maintenance in functionality that is *not* used by users.

The fourth question also relates to the *regression/system testing* task, since it considers test coverage. Information about test coverage helps the test engineer to gain insights into, e.g., where modified source code was not tested. With this information, the test engineer can select existing regression test cases that test untested changes to the software system[2].

The literature cited above emphasizes challenges and questions of practitioners. However, we are not aware of research answering these questions in the maintenance of non-agile projects. Furthermore, we are not aware of research targeting the benefits of execution profiles in software maintenance and test. In this thesis, we aim answering these questions by using execution profiles. We provide approaches to the product owners, test engineers, and developers to gain insights into the usage and coverage of the software systems they are responsible for.

3.2. Involving Users in the Software Maintenance Process

The challenges and questions described in Section 3.1 arise due to the gap between users and developers (see Section 2.4). In this section, we describe the state of the art of closing this gap.

3.2.1. User Involvement

Many researchers emphasize the need for involving users into software maintenance (see, e.g., [81, 82, 83, 84, 85, 86, 87]). Seyff et al. suggest "End-user led requirements engineering" [81, 82], where the users are the main sources of input to the change identification/prioritization task of the product owner. Maalej and Pagano [86] propose a process for involving users into the maintenance process. Kujala et al. [87] conduct a survey on user involvement in the requirements elicitation phase of software projects. Their work shows that early user involvement is related to better requirements quality and to project success.

2 As a precondition, appropriate test cases have to exist beforehand.

The approaches vary in the data they collect, and in the contexts they collect the data. Some approaches described above focus on getting feedback that is manually composed by users. Some approaches combine this information with contextual information that is recorded automatically. There are approaches that collect data only for mobile devices [81, 82, 84], whilst others remove this restriction [83, 85, 86]. In all the aforementioned approaches, the users need to take personal action.

Importance of User Involvement: The aforementioned works state that user knowledge is an important source for developers and product owners to understand the users. Based on this assumption, they emphasize the importance of user feedback during the software maintenance process. The aforementioned approaches aim at getting the users' understanding of the functionality explicitly.

Challenges of User Involvement: Bano and Zowghi [88] report some challenges arising from incorporating the feedback of users into the maintenance process. They categorize the shortcomings of user involvement approaches as psychological, managerial, methodological, cultural and political. From a psychological point of view, users are reluctant to get involved into the development process, since they are not motivated or experienced enough. Managers refrain from involving users, mainly because of the resulting additional efforts on the users' and on the developers' side. From a methodological perspective, it is complex to involve users effectively, and from a cultural perspective, users might not want to have a change in their software systems. From a political point of view, there might arise conflicts between users with different opinions, but also between users and developers [88].

The concrete challenges are, according to Bano and Zowghi [88]:

- Users might not be willing to communicate their requirements, due to a lack of motivation, additional efforts, or confidentiality concerns.
- Not all users have the communication skills to formulate their requirements.
- Users are reluctant against changes in their work environment.
- The management does not want to invest efforts (of users and developers).
- Additional costs are implied for training developers and users.
- Collecting data and analyzing it is complex.
- Users might not be represented appropriately in the collected data.

Additionally, Pagano [89] identifies additional challenges that are focused on the collected data:

- The quantity of data can be large (see also [20]).
- The data might miss structure.
- The content and quality of the data varies.
- The data reflects different preferences of users.

In later work, Pagano and Brügge [90] postulate that developers need tool support to deal with large data. However, they only take feedback into account that users write manually. Additionally, Ko et al. [91] state that developers face problems if change requests or bug reports question fundamental assumptions made in the beginning of the software development process.

Many of these challenges arise due to the direct involvement of users and their need for personal interaction. In contrast, in this thesis, we suggest approaches that involve users only indirectly, while still gathering meaningful information. Moreover, we suggest lightweight, and minimal invasive approaches for collecting and interpreting data about user behavior.

3.2.2. Collection of Data about User Behavior

Taking one step back, Begel and Zimmermann [21] report that product owners, developers, test engineers, and testers want to know what functionality users use (see Section 3.1). Data about the behavior of the user shows what the users use. Therefore, we assume that data about the users' behavior helps these stakeholders. This assumption is underpinned by our own observations [1, 2]. Thus, we present an overview of techniques collecting data about user behavior in this section.

3.2.2.1. Taxonomy

In this section, we characterize approaches for user involvement. We base the structure of this section on the taxonomy of Lethbridge et al. [92]. They focus on collection techniques for data in field studies about the behavior of software engineers. Although this is not the focus of this thesis, the characterization of the collection techniques allows for categorizing different kinds of data about user behavior. They characterize data collection techniques into three degrees, depending on how much interaction with the users is necessary to collect data:

First Degree Techniques: require direct access to and interaction with the users.
Examples: Talking to the users face to face[3], equipping their bodies with instruments, or verbally reminding them of thinking aloud.

Second Degree Techniques: need access to the systems the users use, or their environment, but do not require direct interaction with the users.
Examples: Altering the source code of a software systems to produce customized log files, or notifying users via an automated message on their screen that they are monitored.

Third Degree Techniques: require access only to artifacts created during the usage of software systems.
Examples: Analyzing log files generated without altering the systems itself, or files the users create during using a system.

We structure approaches to the collection of data about user behavior from literature along these three degrees, since they will have a significant impact on our choice of collection techniques. Table 3.1 gives an overview of the techniques described in the remainder of this section. It shows the considered techniques for collecting data about user behavior, their degrees, and summarizes the collected data.

The list of techniques we discuss and explain in the following sections is certainly not complete, since there is a plethora of research that focuses on the behavior of software users. However, the next sections give a comprehensive overview of data containing information about user behavior and techniques for collecting this data.

3.2.2.2. First Degree Techniques

First degree techniques require direct interaction with the users. These techniques are often employed in usability testing [93]. Usability testing focuses on how a system is perceived by its users regarding its, e.g., ease of use. However, these techniques often capture data about user behavior, and about what users use. Additionally, some techniques proposed in usability testing are *observational* [94, 92], which means they aim at collecting data by observing users while they use a system [95]. We do not consider *inquisitive techniques*, such as surveys or questionnaires, since these do not observe users while using a system, but only afterwards.

3 Also, when instructing them prior to collecting data about their usage behavior.

Degree	Collection Technique	Collected Data
First	Audio- or videotaping and manual protocols	Performed actions and the users' thoughts in manual protocols, video- and audio recordings
	Eye-tracking and brain computer interfaces	Eye-movement over a screen and current flow through brain
Second	Recording user interface interactions	Automatically collected protocols containing mouse clicks and key strokes, or higher level interactions like opening files.
	Profiling	Automatically collected protocols about the execution of source code
Third	Analysis of log files	Log files, produced by, e.g., web-services
	Analysis of resource usage	Log files, produced by performance and resource monitors
Mixed	Techniques that are composed of techniques from different degrees	Mixture of the above

Table 3.1.: Collection techniques for data about user behavior with their degrees.

We give an overview over the observational techniques in this section that are most widely used in research. These contain manual protocols, often accompanied with audio or video taping. They often require the users to verbalize their thoughts. We add technically more complex techniques, such as eye-tracking and data collection by brain computing interfaces. They require less interaction to gather data, but more interaction to prepare the collection process.

Audio- or Videotaping and Manual Protocols There are several techniques that make use of audio-, videotaping, or manual protocols. These techniques require the user to think aloud to be able to gather protocols [96, 97]. These protocols include data about what a user thinks while using a software system, which tasks the user performs, why the user performs particular steps, or what is difficult for the user. The goal of this technique is to capture the information that is present in the short-term memory of the user.

Eveland and Dunwoody [98] make audio recordings of think aloud protocols of users, video recordings of the users' facial expressions and the images on the screen, while they are browsing the internet. Their goal is to elaborate whether the structure of linked websites is helpful for the users. So, they use audio- and videotapes to record user behavior. During the study, researchers are in the same room with the participants of their study, to make sure the users keep thinking aloud as intended. Furthermore, the users were asked to perform several supervised training tasks.

Van den Haak et al. [97] also record think aloud protocols (audio). They ask the users to constantly verbalize their thoughts. They make video recordings of the computer screen and also take notes of particular events manually. They are in the same room with the participants. Thus, they use audio-, videotaping, and manual protocols about user behavior. The data is collected in a lab, after briefing the participants of their study how to think aloud. They evaluate the usability of an online library catalogue.

The study of Ramal et al. [99] is concerned with the examination of what information is used by software engineers during their maintenance tasks. In that study, they record everything the software engineer says on audio tape. Furthermore, they manually write a protocol about what the software engineer does, and which tasks she performs. Thus, they combine audio recordings with manual protocols about user behavior.

Nørgaard and Hornbæk [100] also use audio recordings, but not to monitor the users, but the usability engineers that lead the users through a session for examining the usability of a software system. This is a less structured approach to collect data about user behavior. This work shows, however, that just recording audio is a possibility for collecting data about user behavior.

There are several more observational first degree techniques, besides the often used think aloud protocols, regardless of the technical monitoring technique, like audio-, videotapes, or manual protocols. In *shadowing*, one would follow the users to record what they do, and in *participant observation*, data is collected while the data collector becomes part of the user team [99]. The latter is only possible, if the users of a software are organized in teams. However, these techniques are similar to the techniques described above and are less covered in literature.

Classification In the aforementioned techniques, often data collectors are in the same room with the participants to remind them of thinking aloud or to take data. At least, the data collectors remind the users to think aloud and give them instructions how to do this. Therefore, all the techniques are first degree techniques, since they always require direct interaction with the users.

Data Collected These approaches collect data about what a user does. Usually, these are coarse grained actions or steps, collected in manual protocols and videotapes. Furthermore, they collect data about what a user thinks and how a user feels in video- and audiotapes.

Remarks It is hardly possible to automate techniques that use audio- or videotaping or involve manual protocols [101]. The reason for this is the collection process: usually we cannot gather data in a way that the data is readable by a computer directly without cleaning, tagging or preparing it in any way, which requires tedious manual work.

Eye-tracking and Brain Computer Interfaces Other approaches involve eye-trackers or brain computer interfaces. These approaches focus on deriving how a user behaves, or what a user thinks by automatically collecting data.

Granka et al. [102] use an eye-tracker to understand what users of a search engine are looking at and reading before selecting documents. They record where a user of a search engine looks at on the results page. Based on this data, they measure how long the users are reading abstracts, how many they read, and how the rank of a document in the search results influences the final selection. Their study shows that the selected document depends on how long a user reads the abstract and that users tend to read the results list from top to bottom. These results show that eye-tracking is suitable to gain insights into the actual behavior of users.

Similarly, Lorigo et al. [103] use eye-trackers to examine which abstracts are viewed on the result page of a search engine and whether they are viewed sequentially or not. To do this, they track the eye movement of the participants over the results page of a search engine. Their study comes to the conclusion, that almost all users only look at the first results page, and scan through the search results from top to bottom. Also, this study shows, that eye-tracking yields insights into user behavior.

In contrast to the aforementioned approaches, Huang and Miranda [104] use a brain computer interface, particularly an electroencephalography device (EEG) to measure the ionic current flow of the brain's neurons. Their goal is to understand brain functions and use this to make self-adaptive systems to react and anticipate

to users' needs while using a system. They were able to deliberately move a car depicted on the computer screen from one side to another, controlled only by the input of the brain computer interface. Therefore, they were able to understand what the user wanted to do, and actually did, with the software system. Thus, they were able to show that this technique is also suitable to gain data about the actual behavior of a user.

Classification Even though these techniques allow for automated data collection, they still require direct interaction with the users. In the case of eye-tracking, the user has to use an eye-tracking device, and therefore, either the user has to come to a lab, or the data collectors have to visit the users. In the case of brain computer interfaces, the user has to wear an EEG, which also requires preparation and manual calibrations that are done by data collectors. Thus, all of these techniques require direct interaction with the users and, therefore, are first degree techniques.

Data Collected The data collected by these techniques can be differentiated into eye movement over a screen, which are basically the positions on a screen a user looks at and when, and into the amplitude in Volts characterizing the current flow through a brain over time. Both kinds of data are collected by a computer and machine readable.

Remarks Even though eye-trackers and brain computer interfaces got cheaper over the last years, these techniques are still expensive compared to audio-, videotaping or manual protocols. The reason is the high price of the equipment for collecting the data. Additionally, even though the data collected is directly machine readable, it still might require manual preparation, like tagging events in an eye-tracking log.

3.2.2.3. Second Degree Techniques

Second degree techniques do only require indirect interaction with the users, or only access to the software systems they use. This means that there is no personal meeting, talking face to face, touching the users, or sending emails to them. However, techniques that display generated messages on the users' screens, or that alter the users' systems to produce log files, fall into the second degree techniques.

Recording User Interface Interactions Some techniques consider recording interactions users with the user interface. The techniques mainly originate from the examination of usability. Many of the works focus on the usability of websites, and only few focus on the behavior of users on other systems.

Atterer et al. [105] focus on fine grained data about user behavior browsing websites. They collect data about page load, resize, focus, blur, unload, mouse click, hover, move, scroll, and key press events. This data contains technical information about the behavior of users. It is collected by instrumenting the users systems, with the goal to examine the usability of a web pages.

Recording clicks and interactions on websites was also used by Hilbert et al. [106, 107, 108, 109]. Interactions range from fine grained, e.g., clicks or keystrokes, or coarsely grained like the completion of input forms. The interactions can be freely defined by the data collector to gather the data suitable for his task. This data is stored in separate logs, which might be sent over the internet to the data collector.

To collect the data, the users' systems are instrumented with software agents capable of recording the interactions, which form the data about the users' behavior.

With the goal of re-engineering common use cases of legacy software systems, El Ramly and Stroulia [110] collect and examine data about interactions with web-based systems. They record coarsely grained interaction data like opening a catalog, searching it, or viewing details in a list, and mine it to discover patterns many users perform. From these patterns, they derive common usage scenarios, which they transform manually into use cases.

Matejka et al. [111] also mine usage data. Their data are Tuples of the form {User, Command, Time}, and therefore uniquely identify users. However, it remains unclear on what level of granularity commands reside. Their work aims at recommending commands to the users of the system AutoCAD by identifying similar users, detect usage patterns of these users, and recommend the commands to the users which were performed by similar users.

Sequences of commands like edit and paste, format bold, etc. are collected by Linton et al. [112, 113]. The instrument Microsoft Word to collect this data and determine the level of expertise of the users. Furthermore, they compare the actions of users to the actions of an expert to derive learning possibilities of the users.

On Microsoft Excel, Horvitz et al. [114] records mouse and keyboard actions, and the current status of the data in spreadsheets. Furthermore, they derive higher level actions, such as access to menus, dialog boxes, selections and drawing actions, by composing several lower level actions to these high level actions. In their notion, these higher level actions represent goals and tasks, users want to achieve.

Murphy et al. [115] monitor the usage of the Eclipse integrated development environment. Their Mylar Monitor captures coarse grained events such as preference changes, selections, or periods of inactivity, and many more. The collected data is stored locally on the computer of the users and sent for analysis to the researchers. The goal of their research is to find out, how developers use the Eclipse integrated development environment.

Classification All these techniques record user interface interactions. They alter the users' software systems for recording data about their behavior. Therefore, all these approaches require access to the software systems of the users. This modification and recording requires, by the law of some countries, a notification of the users. Therefore, we categorize these techniques as second degree techniques.

Data Collected These techniques collect data about interactions users perform with the system. These interactions range from fine grained, lower level interactions like mouse clicks or keystrokes, to coarse grained, higher level interactions, like tasks. These techniques often combine their data into sequences of interactions, to derive information about the user or his behavior.

Remarks These techniques focus on how a user interacts with a system via its user interface. These techniques do not consider the internals of a software system, which can be much more complex than the systems interface. Additionally, all these techniques require further analysis and interpretation of the data collected, but allow a detailed automated analysis of the data.

Profiling In contrast to recording data about actions users perform on the user interface of a software system, profiling aims at the whole software system. Profiling

techniques monitor a system to gather data about execution times, resource usage, and fine grained traces on the level of source code. This section provides an overview over profiling techniques and their goals.

Elbaum et al. [116, 117] present an overview of strategies to profiling software system. They divide profiling strategies into full profiling, targeted profiling, and profiling with sampling:

Full Profiling On a software system, given a set of events to monitor, e.g., method calls, the approach generates a new program that is instrumented and every event in the event set is recorded. This approach negatively affects the software system's size and execution times.

Targeted Profiling Only particular events occurring locations of interest are monitored. This limits the need for instrumentation to particular parts of the software system and therefore reduces the negative impacts of full profiling.

Profiling with Sampling The software system is paused during execution and samples about the events to monitor are collected, e.g., the methods the system is currently executing. This has less negative impacts on the execution times and size of the software system, but yields possibly inaccurate results.

Traub et al. [45] augment the range of profiling techniques by presenting the concept of ephemeral profiling[4]. In contrast to the techniques presented by Elbaum et al. [116, 117], ephemeral profiling only collects the first few events and unhooks the profiling routine afterwards to save performance. Furthermore, Traub et al. present a technique to record profiling data without changing the source code of a software system itself, but by changing the generated machine readable code at runtime to include their recording routines. With this technique, they are able to add and remove their recording code during the runtime of a software system. Traub et al. use this technique to predict execution branches.

Similar to the approach of Traub et al. [45], Duesterwald and Bala [118] predict paths in the control flow of a software system that are most likely to be executed. They collect execution traces on a statement level but only at branches. Therefore, the recorded data resides more on a block level, since branches are usually followed by these blocks, as, e.g., in for or if statements in source code.

Also, Reps et al. [119] record data about which paths in the control flow of a software system are executed. They collect these profiles to detect deviations in the executed paths between different sets of input data. Particularly, they use input sets of data that contains dates before the year 2000 and after, to detect instances of the Y2K problem.

Salah et al. [120] record method invocations and build sequences from it. From this data, they derive usage scenarios from classes (in an object oriented sense). These scenarios are used to show developers how to (re-)use classes in terms of which methods to execute in which order. For example, a common usage scenario for a class writing contents to a file is first, opening the file, second, writing contents to the file, and third, closing the file.

Data at a technically lower level is collected by Narayanasamy et al. [121]. They collect processor register values, the current state of memory and program counters at particular checkpoints in the software system. They use this data to exactly replay the recorded run of the software system for debugging purposes. This data is very complete and suffices to reconstruct a user's run of a software system. However, it generates huge amounts of data. Close to Narayanasamy et al., Fagui et al. [122] record program counter values and load addresses. With this data, they locate bottlenecks causing long execution times in embedded systems.

4 This is the same profiling technique as used in this thesis.

A hybrid technique of profiling and recording actions is proposed by Juergens et al. [35]. They instrument a system with an ephemeral profiler [45], and determine the usage frequency of functionality by the usage of only particular methods, so called feature beacons. With their technique, they record which methods (in an object oriented sense) were called during a certain time interval, but only consider methods that uniquely determine which functionality was used. Based on their data, they compare the actual usage frequency of features with the usage frequency expected by system experts. Their results show that, first, the experts' estimations about usage frequency are diverse, and second, the expectations of the experts do only sparsely match the actual usage frequencies. This implies that system experts do not know what is used how often in their systems. However, the technique of Juergens et al. requires the maintenance of the methods indicating which functionality was used. Looking at the same system two years after their study, the methods used during their study were no indicators for the functionality they tracked anymore. The reason was that the relationship between methods and functionality was not maintained, since this imposed additional maintenance efforts to the developers.

Classification All profiling techniques require an instrumentation of the users' software systems. Therefore, they require access to the systems. However, not all techniques require modifications of the source code.

Data Collected The data collected by profiling techniques is more detailed than the data gathered by the action recording techniques, since it usually contains single method invocations, statements, branches, or processor data. Thus, the aforementioned techniques collect protocols about the execution of source code. However, data about the whole system is recorded, and not only about actions performed consciously by the user.

Remarks Due to the high level of detail of data resulting from profiling techniques, also the amount of data is huge. This imposed issues regarding the analysis of the data in the past. However, these issues can be overcome nowadays because of more computing power available, and the data can usually be processed automatically without manual interference. Additionally, data can be recorded for whole systems and not only for certain parts. This allows for deeper insights into the users' behavior than action recording based techniques. However, many profiling techniques negatively impact the size and execution times of systems. This means for data collection, we have to choose a trade-off between accuracy of the data, and these negative effects.

Others These are other profiling approaches that are more concerned with resources and time aspects. This section gives some examples about second degree collection techniques, which do not fit into the categories described above.

Yang and Padmanabhan [123] collect data about how long users are visiting certain web-pages and domains with their browsers. Based on this data, they identify individual users, and need access to the browsers the users employ to surf the internet.

Froehlich et al. [124] log the usage on mobile devices. They record actions such as phone calls, sending of messages, taking photos and videos, appointments, contacts, and sensor values. With their framework, they want to provide the context of users to researchers doing studies on software for mobile devices. As they instrument

the mobile phones of users, and collect data from them, they need access to their systems.

Classification The described techniques are examples for different kinds of data that can be collected with access to the users' systems users'. They require either access to the systems itself or to systems that supply the users' systems with data.

Data Collected The collected data is manifold, as described before. This shows that data about the users' behavior can be chosen quite freely, if there is access to the users' systems, according to the goal and analyses tasks that should be performed on the data.

Remarks Depending on the collection technique, the data collected tells more or less about the behavior of the users. The contextual data collected by Froehlich et al. [124], gives detailed and complete data about the behavior of the users, whilst the data collected by Yang [123] gives only insights into the browsing behavior of users. This implies that depending on the goals of the data collection, different sources have to be used.

3.2.2.4. Third Degree Techniques

Third degree collection techniques do not require interaction with the users or access to their systems. These techniques either rely on log files that are typically produced by common software systems like web-servers. Or, the techniques require data that can be collected not at the users' software systems, but at sources that are indirectly used by users, e.g., web services that provide data to the users' systems.

Analysis of Log Files A prominent third degree technique is the analysis of log files, especially log files of web-servers. Reasons for this are that web-servers are multi-purpose software systems that play central roles in the internet. Furthermore, web-servers write log files by default, and so, much collectible data is produced [125]. The data usually contains the IP address of a visitor to the web-service, requested files, and a time-stamp. However, the applications of the analysis of web-service log files are manifold, as described below.

Baysal et al. [126] analyze web-server logs to gather more information about which browsers and operating systems are used, where users live, and what the navigation behavior of the users is. They aim at deriving different usage patterns depending on the age and origin of the users.

In contrast to Baysal et al., who are treating all log files end entries equally, Elbaum et al. [127] record the data based on sessions to gain log files for individual users. From this data, they generate test cases that aim to mime real users.

However, gaining data for individuals is, from the data contained in web-service log files, a difficult task, since information about particular users is not distinguishable from the data about other users. Alam et al. [128] consider which files were accessed, how long and in which order. From this data, they cluster the data from the web-service logs by user. Bayir et al. [129], and Srivastava et al. [130] also cluster this data to derive common usage patterns and investigate usage statistics.

Coombs [131] analyzes proxy-server logs to determine which resources are important in an online library. She mines the proxy-server logs for requests to these resources libraries. This data contains, similar to the logs produced by web-servers, the files requested, the requester's IP address, and a time-stamp.

Yu et al. [132] analyze user behavior on a video on demand system. They record user accesses over time and derive daily, and weekly patterns in terms of arrival rates, and session lengths. From this data, they derive design and cache principles for this kind of systems. They found out that there are certain times of the day where users frequently request videos, and that videos are requested frequently only over a limited period of time.

On databases, Fagni et al. [133] analyze queries, which are logged by database servers, from the past to cache the results of database queries more effectively. They only look at queries that are made to a database, and do not consider the requester.

Classification The described techniques all work on the log files the software systems under consideration produce, regardless of the analyses. Therefore, these techniques only require access to the log files, but not to the systems itself.

Data Collected The collected data usually contains which files where requested, when the request happened, and the IP address of the requester. There are slight variations in whether users are identified uniquely and whether more data, like search requests are included in the data.

Remarks Third degree data collection techniques rely on the fact that systems produce log files. The data often can be processed automatically, and does not require interaction with the users. In contrast to web-servers, that often produce log-files, there are lots of systems that do not produce log files that contain valuable information about the behavior of users. If log files are produced, it might still be the case that these log files are not automatically analyzable. Furthermore, the data collected is coarse grained and does not allow for conclusions about what in a system was used, but only, e.g., which files were requested. More fine grained data can only be gathered, if the software system is instrumented.

Analysis of Resource Usage An indirect approach to the collection of data about user behavior is to measure the resource usage their actions imply. With this data and its analysis, conclusions about the behavior regarding resource usage can be drawn.

Network usage data, in terms of bandwidth, and processor usage data is collected by Abdelzaher [134] to determine the quality of service of web-services. This data is not directly connected to the users itself, but to the services the users' software systems communicate with. However, the recorded data is suitable to determine how much resources the users need when communicating with a web-service and therefore indirectly measures the users' behavior.

Focused on resources, Devaraj and Kohli [135] correlate the performance of hospitals, e.g., the mortality rate, with measures about the usage of resources. Their measures include disk I/O, processing time, and opened documents. They conclude that, for example, the number medical reports opened correlates negatively with the mortality rate. This means informally, the more often reports are opened, the less people die in a hospital.

Gaining insights into user behavior is also the goal of Yang et al. [136]. They collect data about network traffic in China and focus on data usage, patterns in the mobility of users, and which applications the users use. They gather data by instrumenting a service provider's network to collect their data, but do not need any access to the users' systems. With this technique, they gain clear insights into the behavior of users regarding the data they record.

Sinha and Chandrakasan [137] monitor the energy consumption of single processor instructions on mobile devices. They record energy consumption in Joule and the current in Ampere. Similarly, Flinn and Satyanarayanan [138] monitor which user processes, user-level procedures, and kernel-level procedures of a modified NetBSD kernel, consume how much energy. They calculate which procedures consume how much power in Joule and Watt. These techniques also allow drawing conclusions about how much energy users need while using a software system and therefore show the behavior of users regarding energy consumption.

Classification All the described techniques do not rely on a certain system or log files to be produced, in contrast, they monitor the physical machine hosting a software system. Therefore, there is no interference with the users, nor with the software used by the users. Thus, these techniques are third degree techniques.

Data Collected Data about every resource a software system needs can be collected. The examples described above focus on network, disk, and processor usage, as well as on energy consumption.

Remarks These techniques do not monitor what a user needs in a software system, but the resources the software system requires. Therefore, these techniques are not or only indirectly suitable for gaining insights into the actual usage or coverage of software systems. However, these techniques also collect data about the actual behavior of users.

3.2.2.5. Mixed Techniques

Also, combinations of techniques residing on different degrees, exist. For example, Joachims et al. combine eye-trackers (first degree) and recording mouse clicks (second degree). They use the data to determine the reliability of data about mouse clicks. They conclude that only data about clicks is not reliable since, even though users look at lower entries of a search engine's results page, they often click on the first result.

Yang and Padmanabhan [123] focus on online security and identify single individuals. They employ a web-server log file analysis (third degree) to build individual user profiles, but combine this with the data about, e.g., how much an individual scrolls (second degree) to gain more accuracy. Furthermore, they use other data like the frequency and pattern of keystrokes and how the mouse is moved (second degree) to enhance their accuracy. They conclude that with this data about the behavior of users, identifying individuals uniquely is possible.

Roehm et al. [139, 140, 141, 142] collect data about user actions using sensors in the software of the users. They suggest that sensors can be implemented in various ways, like framework hooks (second degree), log file monitoring (third degree), special monitoring code (second degree), or byte code instrumentation (second degree) [139]. They suggest to focus on the actual software that should be monitored, but also consider reuse of the data collecting software. However, in parts of their work, they focus only on second degree techniques [140], or on data that is collected by a software system anyways to enable undoing actions [141]. In more recent research, they focus on capturing button clicks, menu selections textual inputs, among others [142]. In all their research, Roehm et al. always collect data similar to other techniques that are recording actions.

3.2.3. Summary

First Degree Techniques require interaction with the users. These techniques can only consider what a user can observe, but not the internals of a software system. The format of the recorded data can either be chosen freely or is very general, like audio data.

Audio- or Videotaping and Manual Protocols These techniques require manual data collection, because the equipment for recording or preparing the data for automated analysis has to be set up manually.

Eye-tracking and Brain Computer Interfaces These techniques produce data that is automatically analyzable. They require manual preparation, like tagging events in the recorded data, when, e.g., the user focused on a task.

Second Degree Techniques aim at just observing users, and therefore, they do not necessarily require direct interaction. These techniques often produce data that can be analyzed automatically. Additionally, the format of the collected data often can be chosen freely.

Recording Actions These techniques require either particular events, like button clicks or keystrokes, to be defined beforehand or collect data at the user interface of a software system. Therefore, they do not include data about the internals of a software system and do therefore not allow drawing conclusions about functionality that is not directly accessible via the user interface. However, these techniques can be integrated into the user interface without being noticed by the users.

Profiling Depending on the collection technique, the impact on the execution times and resource consumption of a software system is huge and noticeable by the users. However, there are techniques that do not impact the system's performance noticeably. Additionally, the data collected by these approaches is fine grained and contains data about the whole source code of the software system.

Third Degree Techniques require additional data to be existing, such as log files, or work products. However, these techniques do not require any interaction with the users. Additionally, these techniques require no change to the software system. All of these techniques can be automated.

Analysis of Log Files Logging usually is concerned with only particular events. In the case of web-servers, requests are usually recorded, but no information about how a request is processed is collected. This is sufficient to collect data about usage for a website that is hosted on the web-service, but does not suffice for drawing conclusions about the web-server itself.

Analysis of Resource Usage These techniques are not concerned with the user behavior in connection with a software system, but with the effects of the user's behavior. Therefore, only very little information about the system itself can be collected.

Relation to this Thesis The aforementioned techniques focus on the collection of data during the execution of software systems. However, only few of these techniques focus on utilizing their collected data in the software maintenance process. More clearly, only Reps et al. [119] relate to this thesis not only due to

the collection of data about the execution of software systems, but they also guide maintenance based on this data, since they direct developers to possible bugs[5].

The related research presented above propose techniques agnostic for what they are utilized. Thus, there is neither guidance on how to use them in the software maintenance process, nor are there estimations about their benefits or disadvantages in the software maintenance process.

In this thesis, we focus on the direct relation of usage and usefulness, and the utilization of data about the behavior of users in the software maintenance process in non-agile maintenance projects. In contrast to the related approaches described above, we focus especially on the benefits and disadvantages of execution profiles in the software maintenance process.

5 In their example, these are Y2K bugs, but they also consider other bugs

CHAPTER 4

Execution Profiles in Software Maintenance and Test

Contents

4.1. Characterization of Execution Profiles **44**
4.2. Relation to Functionality **46**
4.3. Relation to Test Cases **50**
4.4. Conclusions based on Execution Profiles **50**
4.5. Comparison **54**
4.6. Summary **57**

In this thesis, we apply *execution profiles*, to provide product owners and developers with insights the usage, and to provide test engineers and testers with insights into test coverage. In this chapter, we characterize execution profiles and compare them with the techniques for collecting data about the behavior of users described in Section 3.2.

Execution profiles can be collected in the productive, and testing environment of a software system. But which conclusions can we draw on them? Before answering this question in the last part of this chapter, we first explain the relation of execution profiles to functionality and to test cases. Based on these relations, we discuss which conclusions can be drawn from execution profiles dependent on the environment, in which they were collected.

4.1. Characterization of Execution Profiles

To understand the usage and coverage, we measure, which parts of a software system are executed. For this, we use *execution profiles*, inspired by Eisenbarth et al. [143]. Execution profiles show, which methods were executed. In this section, we explain execution profiles and their collection in detail.

4.1.1. Description

Execution profiles express which parts of the source code of a software system were executed in a given period of time. They do *not* express, how often they were executed during that time interval. Execution profiles are collected automatically, and do therefore not require interaction with the users of the software system.

4.1.2. Profiling Technique

Full Profiling In execution profiles, we consider events in the whole source code of a software system. Therefore, collecting execution profiles is a *full profiling* technique (see Section 3.2.2.3)[1].

Ephemeral Profiling In *ephemeral profiling* [45] (see Section 3.2.2.3), only the first few occurrences of particular, predefined events are recorded. After the first few events, recording is stopped for the part of source code, in which the events occurred. These events can be, e.g., method calls or the execution of certain statements. This technique yields less negative impact on the run-time performance of a system than techniques that record not only the first few events, but all. To keep the negative impact on runtime performance low, we chose ephemeral profiling for execution profiles.

First Execution of Methods The event we specify for monitoring is the first execution of each method[2] of a software system. The *just in time compile event* is raised exactly and only at the first execution of a method during a system run. Therefore, we use it to monitor the first execution of each method. This event is triggered by the runtime environment of the software system, e.g., the common runtime environment (CLR) of .NET for C# systems, or the Java virtual machine (JVM) for Java systems. After the first invocation of a method, once the method was compiled, we do not record any further events on this method.

We call the program collecting execution profiles a *profiler*. For the collection of execution profiles in this thesis, we adapt the profiler of Juergens et al. [35], which realizes the described technique. The implementation of the ephemeral profiler targets C# systems, but the principles can be adapted also to Java systems.

Example Figure 4.1 illustrates three exemplary system runs with method invocations. The system, containing three methods, is executed in three system runs. For each system run, we record one execution profile. In the first run, the methods 1 and 2 are executed. We record their first executions. In the second system run, all methods are executed and we record their first executions. In the third system run, method 1 is not executed, but the other methods. Therefore, we record the first

1 In contrast to *targeted profiling* (see Section 3.2.2.3), where events only in parts of the source code are considered.
2 In an object oriented sense

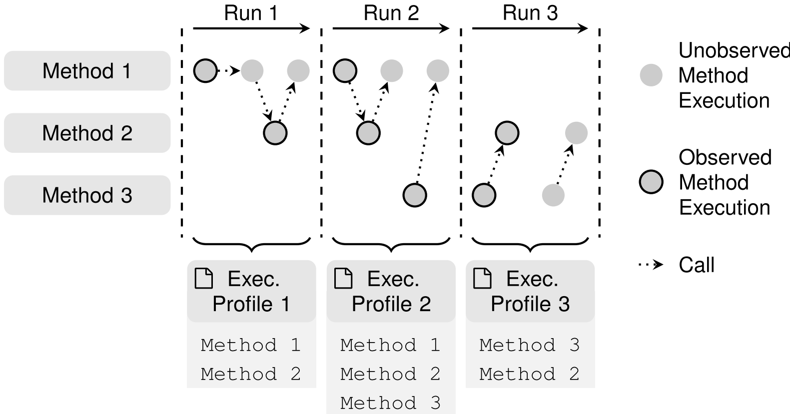


Figure 4.1.: Example of three system runs and recorded method executions.

executions of method 2 and method 3 in the execution profile 3. Note that in one system run, several methods can be triggered by the users via the user interface. In Figure 4.1, these methods do not have incoming arrows. For example, in the third run, a user triggered the execution of method three, which resulted also in the execution of method 2. Possibly another user also triggered method three, which also resulted in a call of method 2.

4.1.3. Collected Data

Table 4.1 shows the data contained in an execution profile. Execution profiles considered in this thesis always contain this data. They contain the start and end time of their recording, which are the start, respectively the end time of the corresponding system run. Furthermore, we record the name of the system, and the environment it was executed in, which can be, e.g., the productive environment or the testing environment. Execution profiles contain data of all methods that were executed during a system run. For each first execution of a method, execution profiles contain the fully qualified name, parameter types, and the number of type parameters of the method.

The example for an execution profile in Table 4.1 was recorded from the 25th of February 2014 to the 26th of February 2014. It was recorded in the productive environment of the system with the name mySystem.

Table 4.1 shows an example for an executed method with the signature, denoted in C# syntax, `myMethod<T1,T2>(int, mySystem.MyDataClass)`. This method is located in the class `MyClass`, which is in the name-space `mySystem`. It has parameters of the types `int` and `MyDataClass`, where the latter type is located in the name-space `mySystem`. The method has two type parameters `T1` and `T2`. With this data, we can uniquely identify, which methods were executed at least once.

We specified the first execution of each method as the event to monitor for our profiler. Therefore, we are not able to assess whether a method was only partially[3] executed. Consequently, we consider all methods as being executed, if any statement of their body was executed.

Throughout this thesis, especially in all contributions, we use execution profiles. We gain information about which parts of a software system were executed, and more important for our approaches, which were not.

3 We consider a method as partially executed, if not all the statements contained in the method body were executed, but a part of them.

Name	Fields	Example	Multiplicity
Context	Name of system Environment	mySystem Productive	1
Time interval	Start time End time	25/02/2014, 15:46:23 26/02/2014, 17:39:19	1
Method signature	Fully qualified name Parameter types Number of type parameters	mySystem.MyClass.myMethod int, mySystem.MyDataClass 2	0..*

Table 4.1.: Data contained in execution profiles.

4.1.4. Data Collectors

Above, we described the data contained in execution profiles, and how we collect them. But who performs the data collection?

Operations[4] staff are responsible for deploying software systems to the productive and testing environment. Moreover, they are responsible for running the hardware and operating system, on which the software system is deployed. Additionally, they are in charge of administering the productive and testing environment by, e.g., managing user accounts.

Our profiler that we use for collecting execution profiles, has to be installed, and the resulting execution profiles need to be collected. This is done by operations, since they are responsible for deploying software, and to administer the underlying hardware and software.

In DevOps [144], the developer is also charged with the tasks of operations. In this case, the developer can gather the execution profiles on their own. This way, they get direct feedback about the usage of their software systems.

4.2. Relation to Functionality

Users use the functionality of business information systems in its productive environment. Execution profiles collected in the productive environment show which methods were executed. During the usage of functionality of a software system, not necessarily all methods of the software system are executed, but just the methods that were triggered by the interactions of the user with the system. Thus, execution profiles show, which methods were executed during the usage of a software system, and which were not. Therefore, we call execution profiles collected in the productive environment *usage data*.

Execution profiles contain information on the level of methods in source code. However, users use functionality via a user interface, not the source code directly. Modern business information systems often provide not one single functionality, but comprise functionality for different tasks of users. In this section, we describe the relation of functionality to methods, and consequently execution profiles.

4.2.1. Relation of Functionality to Source Code

Functionality is realized in source code. Source code is divided into methods. Therefore, functionality is realized in methods. Usually, functionality is realized

4 Not to be confused with operations on a source code level.

in several methods. Vice versa, methods can contribute to the realization of various functionalities, resulting in a many-to-many relations between methods and functionality.

4.2.1.1. Many-to-many Relation of Functionality to Methods

Due to the many-to-many relation between functionality and methods, the mapping of functionality of methods can take various shapes. Functionality can be realized in a *distinct* set of methods, the methods implementing functionality can *overlap* with methods implement other functionality, or be *included* in the set of methods realizing other functionality.

We call methods, which contribute to the realization of exactly one functionality *characteristic* for this functionality (this corresponds to the notion of Juergens et al. [35]). In the example illustrated in Figure 4.3, Method 1 is characteristic for Functionality A, since it does not contribute to the realization of other functionality, whereas Method 2 contributes also to other functionality. At the same time, when Method 1 is executed, we know that Functionality A was used.

Mutually Distinct Functionality: We call functionality *mutually distinct*, if it is realized only in characteristic methods. Figure 4.2 illustrates this case. Functionality A is realized by the methods Method 1 and Method 2. All other functionality of the software system is realized in Method 3 and Method 4. Functionality A does not share source code with other functionality. The methods realizing Functionality A do not overlap with methods implementing other functionality.

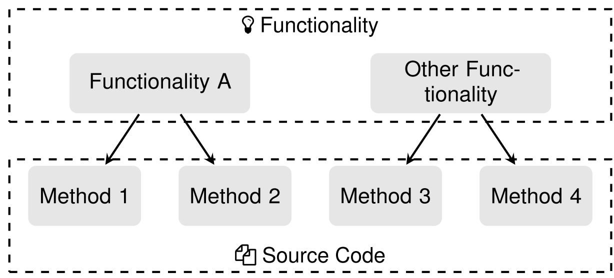


Figure 4.2.: Functionality A realized distinct from other functionality.

Overlapping Functionality: We consider functionality as *overlapping*, if it is realized also with methods that contribute to the realization of other functionality, but also in methods that do not realize other functionality. This case is illustrated in Figure 4.3. Method 2 contributes to the realization of Functionality A, but also to other functionality. Method 1, however, only contributes to the realization of Functionality A. Therefore, the set of methods implementing Functionality A overlaps with the set of methods implementing other functionality. In this example, only Method 1 is *characteristic* for Functionality A.

Included Functionality: One functionality is realized only in methods that also realize other functionality. Figure 4.4 illustrates this case. The methods realizing Functionality A, Method 1, and Method 2, contribute also to the realization of other functionality. Thus, the methods realizing other functionality fully include the methods realizing Functionality A. In this example, no method is *characteristic* for Functionality A.

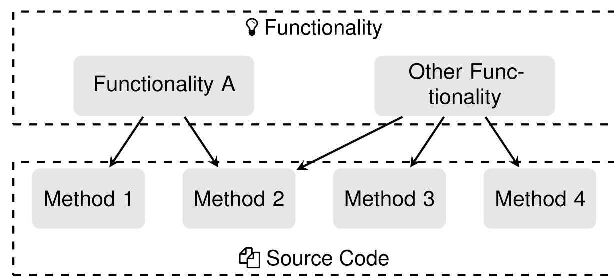


Figure 4.3.: Functionality A realized overlapping with other functionality.

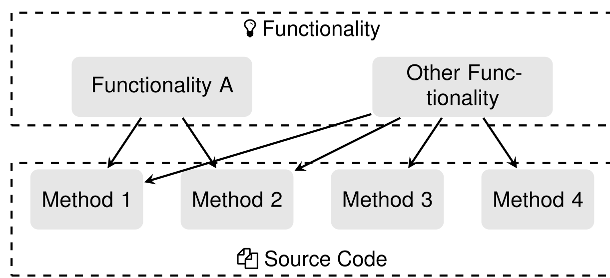


Figure 4.4.: Functionality A realized included in other functionality.

4.2.1.2. Mandatory and Optional Methods

Methods contributing to functionality are either always executed, if the functionality is used, or not. We call methods, which are always executed *mandatory*. We call methods *optional*, if they are not always executed, e.g., because the user can cancel his usage of a functionality after its beginning. Figure 4.5 shows Functionality A that is realized by two *mandatory* methods, Method 1 and Method 2 (indicated by solid arrows), and two *optional* methods, Method 3 and Method 4 (indicated by dotted arrows).

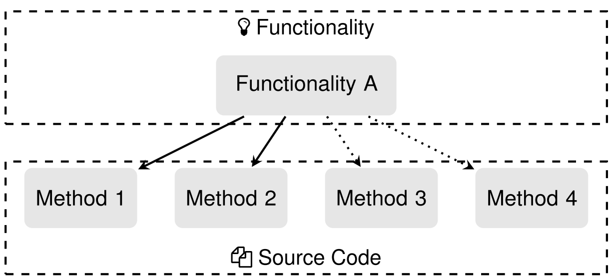


Figure 4.5.: Functionality A realized by mandatory and optional methods.

4.2.2. Relation of Execution Profiles to Usage of Functionality

Users make use of functionality, but execution profiles are collected on the level of methods in source code. So, how do execution profiles relate to functionality? If functionality is used, its realizing *mandatory* methods are executed, and its realizing *optional* methods are executed potentially. In the example illustrated in Figure 4.5, this means that the usage of Functionality A results in the execution of Method 1 and Method 2, and in the potential execution of Method 3 and 4.

Characteristic	Mandatory	Executed	Realized Functionality Used
✓	✓	✓	yes
✓	✓	–	no
✓	–	✓	yes
✓	–	–	unknown
–	✓	✓	unknown
–	✓	–	no
–	–	✓	unknown
–	–	–	unknown

Table 4.2.: Relation of the properties mandatory, characteristic, and executed of methods to the usage of the functionality they realize.

The execution of a characteristic method indicates the usage of the functionality, for which the method is characteristic. However, there are no characteristic methods for *included* functionality. In this case, we cannot conclude about the usage of functionality from the characteristic methods of the particular functionality. This limits the expressiveness of execution profiles on the level of functionality.

The absence of execution of mandatory methods shows that the realized functionality was not used. However, the absence of the execution of optional methods does not allow for this conclusion. This also limits the expressiveness of execution profiles. Additionally, with execution profiles, we cannot draw conclusions about functionality that is realized below the level of methods, for example, in parts of methods. In this case, we only consider the execution of the method itself and cannot distinguish between different functionality.

Table 4.2 summarizes the conclusions that can be drawn from execution profiles to the usage of functionality.

In case characteristic methods are executed, we conclude the functionality was used. If mandatory methods are not executed, we conclude the functionality was not used. However, in the other cases, we cannot infer usage of functionality from the execution of methods.

Especially in the case of included functionality, where no characteristic methods exist for a particular functionality, we cannot draw conclusions about its usage from execution profiles. However, Juergens et al. [35] report they were able to identify methods that were both, characteristic, and mandatory for about 68% of all functionalities in a business information system. Therefore, we are confident, that we can draw conclusions about the usage for most functionality.

To cope with the aforementioned shortcomings, one could consider functionality with unknown usage as unused. This leads to an overestimation of unused functionality. This means that functionality is considered as being unused, which is used. If we search for unused functionality, this strategy consequently leads to lower precision, but higher recall.

Vice versa, if we consider functionality with unknown usage as used, we underestimate unused functionality. For the identification of unused functionality, this leads to higher precision, but lower recall. In this thesis, we consider functionality with unknown usage as being used, to gain the higher precision, since we then can point to useless functionality more accurately.

4.3. Relation to Test Cases

If execution profiles are collected in the testing environment, they show, which methods were executed there. The execution of test cases (manually or automatically) triggers the execution of methods of the software system. During the test of a software system, not all methods are executed necessarily, but just the methods that were triggered by the execution of test cases. This relates to our notion of *method coverage* (see Section 2.3.7), and therefore, we call execution profiles collected in the testing environment *coverage data*.

4.3.1. Relation of Test Cases to Source Code

Figure 4.6 shows a simple example of tests contained in a test suite that execute methods. In this example, the test suite contains two test cases, Test Case A and Test Case B. Test Case A triggers the execution of the methods Method 1 and Method 2. Test Case B triggers the execution of Method 2, and Method 3. So, Method 2 is executed by both test cases, and Method 4 is not executed by any test case. We consider the executed methods as *covered* by a test case, if they are executed during the execution of the test case. Methods are *uncovered*, if they are not executed by any test case. In our example, Method 4 is uncovered.

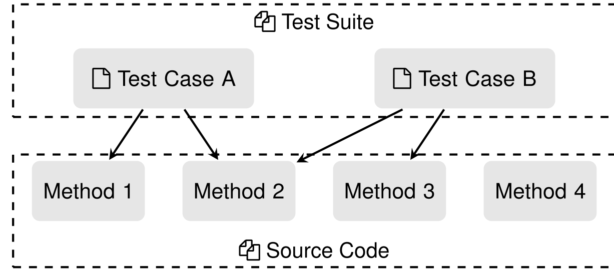


Figure 4.6.: Two test cases that trigger the execution of methods.

4.3.2. Relation of Execution Profiles to Code Coverage

Execution profiles show, which methods were executed during testing, and which were not. This corresponds to *method coverage* (see Section 2.3.7). Thus, we consider execution profiles collected in the testing environment as *coverage data*. In the example illustrated in Figure 4.6, the Method 1, Method 2, and Method 3 were covered by test cases. Method 4 was not covered.

4.4. Conclusions based on Execution Profiles

Execution profiles collected in the productive environment are *usage data* (see Section 4.2). If execution profiles are collected in the testing environment, we consider them as being *coverage data*. In this section, we describe, which conclusions can be drawn on usage data regarding *usefulness* (see Section 2.5). Additionally, we show which conclusions can be drawn about the degree of validation on coverage data. We focus on the expressiveness of execution profiles with respect to the reliability of the conclusions based on them.

4.4.1. Conclusions on Usage Data

Usage data (see Section 4.2) expresses, which functionality is used by the users. In this section, we explain how usage data relates to *usefulness* (see Section 2.5). We base our description on the TAM (see Section 2.5), but revise the connection between usefulness and usage critically.

4.4.1.1. Categories of Functionality

We classify functionality into the categories *implemented*, *used*, *known*, and *useful*. The different categories are illustrated in Figure 4.7 and explained below.

Implemented Functionality that is realized in a software system[5].

Used Functionality that is executed by the users (see Section 2.2.3).

Known Functionality the users are aware of.

Useful Functionality that is perceived as useful by the users (see Section 2.5).

We also consider the counterparts of the categories. For example, functionality is unused, if it does not fall into the category used.

We assume that there is no functionality that is not known, useful, and used. The reason for this is that once a user uses functionality he perceives as useful, the functionality is known. Furthermore, only implemented functionality can be used.

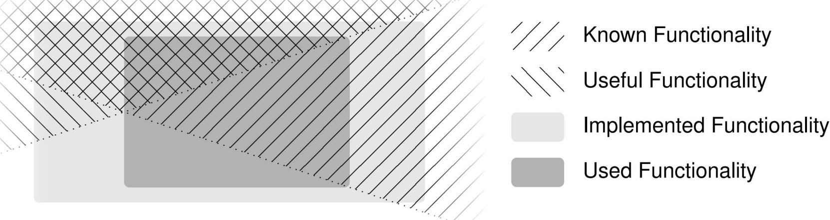


Figure 4.7.: Implemented, used, known, and useful functionality.

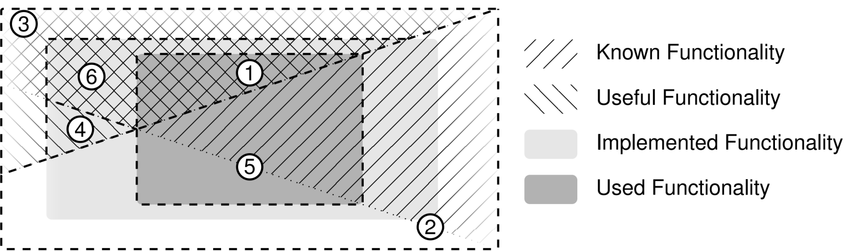


Figure 4.8.: Summarized types of functionality.

We summarize several categories into types of functionality, as depicted in Figure 4.8. The areas outlined by dashed lines are these types. We define the types by the schema in Table 4.3. In this table, the right column shows the representation of the category in Figure 4.8. The left column contains the conditions under which we consider functionality to belong to a certain category. So, for example, the functionality falls into the category *intentionally used*, if it is useful, used and known.

5 We use the terms *implemented* and *realized* interchangeably.

Name and Condition	Description	Reason	
Intentionally used *useful, used, known*	Users perceive functionality, which they know as useful, and therefore use it.	Intention of the users.	1
Intentionally unused *not useful, unused*	Users perceive functionality as not useful, and therefore do not use it.	Intention of the users.	2
Not implemented *useful, not implemented*	Users (would) perceive functionality as useful, whether they know about it or not, but the functionality is not implemented (yet).	There are some explanations for this effect, e.g., rapidly changing demands of users or misguided requirements engineering, which did not identify functionality that users want.	3
Unknown *useful, unused, unknown, implemented*	Users would perceive implemented functionality as useful, but do not know it, since they are not experienced enough with the software system, and therefore do not use the functionality.	Functionality is new and users were not trained for it, the functionality is hard to find, or not reachable by the users.	4
Accidentally used *not useful, used*	Users do not perceive functionality as useful, but use it anyways (whether they know the functionality or not).	Users accidentally click on a wrong button or expect a different functionality when using the software system. Another reason are unnecessarily complex workarounds that replace useful functionality. This is to be expected for inexperienced users.	5
Bad ease of use *useful, unused, known, implemented*	Users perceive implemented functionality they know as useful, but do not use it.	According to the TAM, one reason is low perceived ease of use, which is caused by bad usability of a functionality. An example are functionality that is too complex to use, or functionality that is provided also by other tools the customer has installed.	6

Table 4.3.: Categories of functionality.

Figure 4.7, Figure 4.8, and Table 4.3 show a view on the system that assumes all users use, know, and consider the same functionality as useful. This is an unrealistic assumption. We expect the opinions, knowledge and usage of users to deviate. Thus, one user might use functionality intentionally, but others use it accidentally.

However, we consider these deviations to be small, since we focus on custom software systems that are tailored to specialized tasks of users with similar jobs. Therefore, also the tasks users want to perform are similar or clearly separated. This leads to the conclusion that the differences of users regarding their opinion and usage behavior in the system is rather consistent to other users performing similar tasks and isolated from the users performing different tasks.

4.4.1.2. Critical Reflection of Relation between Usage and Usefulness

Under consideration of the TAM (see Section 2.5), useless functionality is not likely to be used, while useful functionality is more likely to be used. Thus, the user shows,

by using a system, what he perceives as useful. On the other hand, the user shows which functionality is not useful by not using it. However, the relation between usage and usefulness remains fuzzy, due to the reasons explained in Table 4.3.

But, in the context of custom business information systems, the approximation of usefulness by usage (and execution profiles) is accurate enough to be helpful for practitioners [1]. Thus, in the software maintenance process, data about the usage of software can be used to gain insights into the usefulness of functionality.

As illustrated in Table 4.3, there is implemented functionality that is useless. This functionality often provides no value for the users, and therefore, its maintenance bears the risk of being a waste of resources. This functionality is realized because of the reasons given in Section 2.4.

The goal of the product owner is to reduce the waste of resources in maintenance. Therefore, we suggest to question maintenance on functionality that is unused. However, due to inaccuracy of the relation between usage and usefulness, product owners still have to manually inspect whether unused functionality is really useless. But with usage data, at least first insights into the uselessness of features is possible.

There can be functionality that is used only rarely, but is useful for the users. An example in business information systems is the annual accounting. Additionally, custom business information systems have less users than off-the-shelf software or, e.g., well known web services, like search engines. Therefore, new or changed functionality might not be used soon after its realization and deployment. Therefore, we suggest to continuously collect execution profiles for at least one business cycle.

As also shown in Table 4.3, and shortly discussed in Section 2.5.1, not all useful functionality is actually used, nor is all used functionality useful. In the case of *unknown* functionality, the users do not know about implemented functionality they would perceive as useful, and therefore do not use it. On the other hand, in the case of *accidentally used* functionality, the users use functionality even though they do not perceive it as useful. Reasons for this are: in the case that more useful functionality is implemented, the users do not know about it. In the case that more useful functionality is not implemented, that the users use the less useful functionality as a substitute. Another reason is, that users use functionality not on purpose. In the case of *bad ease of use*, the users do not use functionality, they actually perceive as useful, since it is, e.g., too complex or hard to use.

4.4.2. Conclusions on Coverage Data

Coverage data shows, which methods were executed during the execution of test cases, and which were not. Consequently, due to the relation between methods and functionality (see Section 4.2), coverage data shows, which functionality was executed by a test case.

We categorize functionality into the groups *covered*, and *tested* in the context of coverage data. Figure 4.9 illustrates these categories. Only functionality, that is already implemented (see Section 4.4.1.1) in the software system can be covered. And only covered functionality can be tested. But, coverage of functionality does not mean, that it was indeed tested.

Covered Functionality, for which the realizing methods were executed during testing by test cases (see Section 2.3.7).

Tested Functionality that was validated for conformance against the customer's expectations (see Section 2.3.1).

However, there are some limitations to the conclusions we can draw from coverage data. The main reason is that *covered* functionality is not automatically *tested*.

Figure 4.9.: Covered, and tested functionality of a software system.

There are parts in system test cases, which execute a software system, but do not validate its functionality against the customer's expectations. Examples are the setup and tear-down parts, which bring the system into a predefined state before the actual checks begin, and reset the system afterwards. Above that, system test cases often do not compare every single output of a system to an expected result, since this would be too effortful. An extreme case are test cases that do not contain expected outputs at all, and therefore cannot produce a negative verdict. These assumptions are underpinned by, e.g., Inozemtseva and Holmes [145], who report that coverage does correlate with the ability of test suites to detect faults, but is not a good predictor for how many faults are revealed by a test suite. Therefore, we conclude that sole coverage of functionality does not mean it was tested.

But what can we conclude from coverage data? Functionality that is realized by methods that were not executed by test cases cannot be tested. Therefore, uncovered functionality cannot be tested. Thus, we conclude from coverage data, which methods, and consequently, which functionality was certainly *not* tested. So, the expressiveness of coverage data is therefore limited to which functionality is certainly not tested. But, due to the reasons given above, we cannot identify every untested functionality.

Additionally, although coverage data is collected on source code, we cannot infer conclusions about which methods were tested. However, due to the same reasons as described above, we can conclude which source code was definitely not validated against the customer's expectations. But also due to the reasons given above, we cannot identify all untested methods based on coverage data.

4.5. Comparison

The chosen technique for collecting execution profiles has some advantages over, but also disadvantages compared to the techniques presented in Section 3.2.2.

In this section, we compare the collection and analysis of execution profiles to the collection and analysis of other data about the user behavior and execution of software systems (see Section 3.2.2). We focus on the challenges of user involvement reported by Bano and Zowghi [88], and Pagano [89] (see Section 3.2.1).

4.5.1. Advantages of Execution Profiles

In this section, we describe the advantages of using execution profiles in the software maintenance process compared to the related techniques presented in Section 3.2.2.

No User Interaction The collection of execution profiles does not require direct interaction with the users. This is an advantage over first degree techniques, since we do not disturb the users or impose additional work to them. Additionally, the

users might be separated from the *product owner* (see Section 2.2.3), since the work, e.g., in different countries. Therefore, we cannot assume, that the product owner can directly access the. With execution profiles, the product owner does not have to communicate directly with the users. In communication between the product owners and the users the misunderstandings arise (see Section 2.4). Therefore, misunderstandings between users and product owners (see Section 2.4) are less likely when considering execution profiles than in techniques that require direct feedback from the users.

This way, the approach is also applicable in contexts, where users are unwilling or unable to communicate their requirements, as reported by Bano and Zowghi [88].

Seamless Integration Execution profiles are collected in a way that is not, or only hardly noticeable for the users. Therefore, we avoid an impact on the users' work or experience with the system. This is an advantage over other techniques, because users do not have to get used to a changed system, or to data collection techniques, and do not need to be trained for it[6]. However, we suggest informing the users, that they are monitored.

These properties ensure that no efforts have to be invested for user training. Additionally, the users do not have to get used to a different working environment, to which they can be reluctant [88]. Therefore, we consider execution profiles as less invasive than first degree techniques.

Self-Containment Not all software systems produce data about execution or interaction by users. Execution profiles do not depend on any preexisting data. This is an advantage over third degree techniques, because these techniques require particular data, like log files, which might not exist. Additionally, we can freely choose the format of the collected data, by employing our own technique.

This property counteracts the varying structure, content, and quality of data, as reported by Pagano [89].

Little Effort The collection and analysis of execution profiles can be automated to a large extent. Therefore, we argue that the additional effort for the collection and analysis of execution profiles for product owners, developers, test engineers, and testers, is low. This advantage is apparent compared to first degree techniques, that often employ manual data collection, direct interaction with the users, and manual analyses of data (see Section 3.2.2.2). Therefore, we consider execution profiles as lightweight. In first degree techniques, the collection and the analysis of the collected data often requires manual effort. Therefore, first degree techniques impose more effort on product owners, developers, test engineers, and testers.

Low additional efforts reduce the reluctance of the management due to additional costs [88].

Completeness Execution profiles contain data about the whole source code of a software system and all users. Therefore, there is less bias in the data towards single users, compared to techniques that examine the behavior of only few users, like first degree techniques. The second degree technique *recording actions*, usually monitors only particular events on the user interface of a software system and therefore does not consider all parts of a system. The third degree techniques that

6 This could also lead to confounding factors, where observations are caused by the changed environment.

analyze resource usage, can only indirectly draw conclusions about the execution of a software system.

Therefore, execution profiles reflect the preferences of all users together, and not different preferences [89], while containing data about a whole software system.

4.5.2. Disadvantages of Execution Profiles

In the following, we describe disadvantages of execution profiles compared to other approaches (which were presented in Section 3.2.2). The collection of execution profiles are limited to existing source code, which threatens the comprehension of product owners, test engineers, and testers. Furthermore, with execution profiles, we can only gain insights into the execution of existing source code and functionality, but not functionality the users desire besides the realized functionality.

Comprehensibility Execution profiles reside on the level of source code. Therefore, the understanding of execution profiles alone requires knowledge about the source code. However, this is not always the case for product owners, test engineers, and testers. This is the case for all profiling techniques.

In audio- and videotaping techniques, the users are recorded directly. The tapes show the interactions a user performs with the system. We expect this to be comprehensible for all stakeholders. However, eye-tracking data, and EEG-protocols require great understanding of these techniques.

The comprehensibility of recorded actions for product owners, developers, test engineers and testers depends on the granularity of the actions. But, if actions on the level of functionality are recorded, we expect the product owners, developers, test engineers, and testers to understand them. Profiling techniques reside on the level of source code. Product owners, test engineers, and tester often do not have this knowledge. Therefore, we consider profiling techniques as less comprehensible for them.

Third degree techniques do not collect data, but just analyze it. Therefore, the comprehensibility for product owners, developers, test engineers, and testers depends on the presentation of the results of the analyses.

Desired Functionality Additionally, execution profiles contain data only about existing source code, and thus, realized functionality. Therefore, it is impossible to get insights into which source code and functionality *would* be executed, if it *would* exist. In general, second and third degree techniques are bound to existing functionality and source code, since these techniques monitor existing and running software systems.

In audio- and video-taping techniques, users are often requested to verbalize their thoughts. Thus, they can also express what they *would* do, in case other functionality *was* realized, or which functionality they miss. However, in eye-tracking techniques, users can only look at existing data. We found no evidence, that brain computer interfaces are capable of identifying functionality the users desire.

4.5.3. Summary

Table 4.4 illustrates the properties of the different collection techniques for data about user behavior. It is structured by the three degrees of user involvement techniques and follows the structure of our description of related work in Section 3.2.2.

Technique	Degree	No User Interaction	Seamless Integration	Self-Containment	Little Effort	Completeness	Comprehensibility	Desired functionality
Audio- or videotaping and manual protocols	First	–	–	✓	–	–	✓	✓
Eye-tracking and brain computer interfaces		–	–	✓	(✓)	–	–	–
Recording actions	Second	✓	✓	✓	✓	–	(✓)	–
Profiling		✓	(✓)	✓	✓	✓	–	–
Analysis of log-files	Third	✓	✓	–	✓	(✓)	(✓)	–
Analysis of resource usage		✓	✓	–	✓	–	(✓)	–
Execution Profiling		✓	✓	✓	✓	✓	–	–

Table 4.4.: Comparison of techniques for data collection about user behavior.

It shows that execution profiles have some advantages, but also disadvantages compared to other approaches to collect data about the behavior of users.

We conclude that the approach is lightweight, since it does not impose additional efforts to users, and only small additional efforts for product owners, developer, test engineers, and testers. It is furthermore not invasive, since execution profiles do not require direct interaction with the users.

Execution profiles show, which methods are executed in a software system. The execution of methods is triggered by the usage of functionality. The usage of functionality indicates its usefulness. The absence of usage of functionality indicates uselessness. Additionally, the absence of usage leaves methods unexecuted, which is also reflected in execution profiles. Also, execution profiles show, which source code was executed during testing, and which was not. The functionality realized in unexecuted source code was certainly not tested, and therefore, is more likely to contain faults than covered source code.

Due to these relations, and due to the advantages of execution profiles discussed above, we conclude that execution profiles are eligible for product owners, developers, test engineers, and testers to gain insights into usage and coverage.

However, execution profiles cannot give insights into functionality that is desired by users, but not yet implemented. The only technique that is capable of this, is audio- or videotaping, or recording think aloud protocols.

Additionally, execution profiles are not comprehensible per se. We address this issue in our contributions, which are described in the next chapter.

4.6. Summary

We presented our approach to collect information about usage and coverage. There, we explained execution profiles in detail. Additionally, we compared execution profiles to other approaches described in Chapter 3, and focused on advantages and disadvantages of execution profiles over other techniques.

Furthermore, we described, how execution profiles relate to functionality and tests, to be able to explain, which conclusions can be drawn based on execution profiles. We discussed these conclusions afterwards. From execution profiles, we can gain insights into which functionality was not used. Additionally, execution profiles show, which source code was definitely not tested.

CHAPTER 5

Contributions

Contents

5.1. **Unused Source Code in Maintenance** 62
5.2. **Transfer of Unused Source Code to Requirements Artifacts** . . 63
5.3. **Considering Usage Data in Maintenance** 64
5.4. **Uncovered Source Code in Regression Testing** 70
5.5. **Transfer of Uncovered Source Code to Regression Test Cases** . 71
5.6. **Considering Coverage Data in Testing** 72
5.7. **Automatically Linking Source Code with Other Artifacts** 80

In this chapter, we present the contributions of this thesis. All contributions build on *execution profiles* (see Chapter 4) and on the foundations explained in Chapter 3.

Motivated by the challenges reported by Heiskari and Lehtola [20], the questions raised by Begel and Zimmermann [21] (see Section 3.1), and our own observations [1, 2], we state the research objective for contexts, where users are not involved closely in the maintenance process: We explore the benefit of analyses of execution profiles collected for business information systems in software maintenance and test.

But, how can product owners, developers, test engineers, and testers profit from execution profiles?

We answer this question by presenting clarifying studies and approaches that apply execution profiles in the software maintenance process (see Section 2.1). We focus on the activities maintenance and regression testing. Figure 5.1 shows the contributions, and their logical dependencies.

Execution profiles show, which source code was executed, and which was not. However, the presence of execution often does not allow for sharp conclusions (see Section 4.4.1 and Section 4.4.2). But, the absence of execution does allow for more precise conclusions.

To gain insights into the benefits of execution profiles, we first present studies about the existence and extent of unexecuted source code in the productive and testing environment. We present a study about the extent and the effects of unused source code on the activity maintenance (**Publication A**). Additionally, we present a study about the extent and the effects of uncovered source code in regression

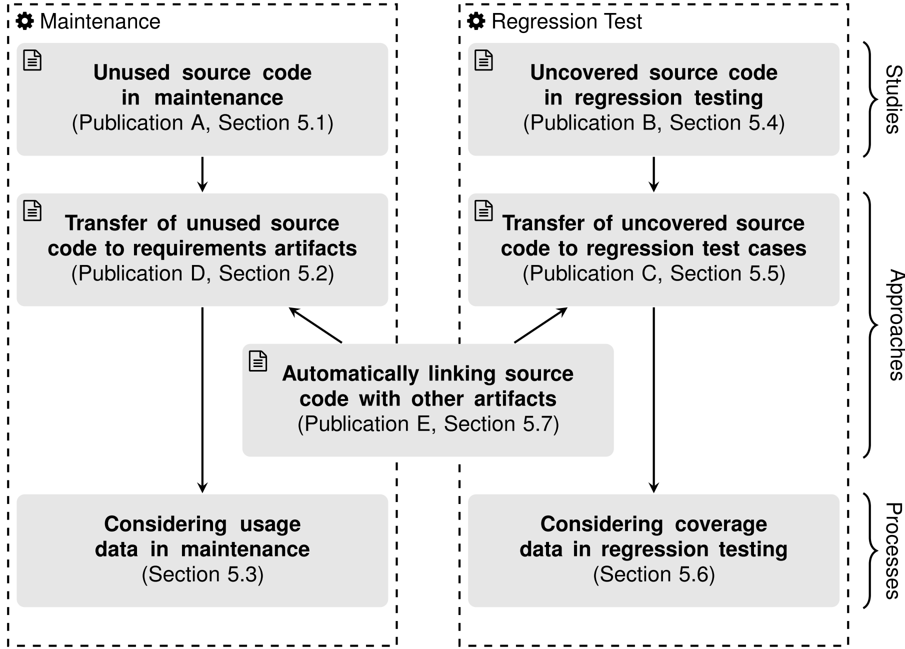


Figure 5.1.: Overview of the contributions.

testing (**Publication B**). The studies show the effects of unexecuted source code in the software maintenance process.

Execution profiles are collected on the level of source code. They contain information about source code. However, product owners, test engineers, and testers often do not know the source code of the software systems, for which they are responsible. Additionally, decisions about change requests, maintenance, and regression testing is are usually not made based on the level of source code, but on the level of functionality.

To enable product owners, developers, and test engineers to use execution profiles, we present approaches that transfer execution profiles from the level of source code to the levels of functionality and test cases, which the stakeholders understand better.

Thus, we present an approach to transfer unused source code to functionality, to make execution profiles usable for product owners in the task *change identification/prioritization* (see Section 2.1), who do not know the source code (**Publication D**). Additionally, we present an approach to select regression test cases that cover previously changed, but uncovered source code (**Publication C**).

The latter approaches use Latent Semantic Indexing [46] (LSI) from the field of information retrieval to link execution profiles to other artifacts like use case documents or test cases.

The goal of LSI is to measure the semantic similarity between documents contained in a document corpus. But, LSI works on the syntax of the documents, while still detecting semantic similarity. Similarity is expressed by a value between -1 and 1, where a higher value means the compared documents are more similar. LSI identifies words belonging to a common concept (e.g., "car" and "automobile") by co-occurrence, enabling it to deal with synonyms.

The accuracy of the results of LSI vary with its configuration. However, to configure LSI, technical knowledge about this technique is needed, and the configuration has to be done manually. Therefore, to improve the applicability of the presented

techniques, we present an approach to fully automate the configuration of LSI (**Publication E**).

But how do the approaches integrate into maintenance and testing? To answer this questions, we extend the maintenance process (see Section 2.1) and regression testing (see Section 2.3.2) with tasks that enable the consideration of execution profiles. The resulting processes show how and where execution profiles and the aforementioned approaches can be brought into the existing processes.

The approaches and processes show that product owners, developers, and test engineers can consider execution profiles in the maintenance process. Product owners are supported in their task *change identification/prioritization*, developers in their task *implementation* and *design* (see Section 2.1). Test engineers are supported during the *creation* and *selection* (see Section 2.3) of regression test cases.

Together with our studies showing the value of execution profiles, we conclude that the aforementioned stakeholders can benefit from execution profiles.

To the best of our knowledge, there is no research about how execution profiles can be used to answer the questions reported by Begel and Zimmermann [21] *after* they arise. We close this gap with this contribution. However, we are aware of agile methods that aim at constructively addressing these questions *before* they arise by closely involving the customers into the development or maintenance process.

5.1. Unused Source Code in Maintenance

Published in [1] at ICSE 2012
acc. rate 16%
10 pages
Full paper

The author of this thesis designed the approach to identify unnecessary maintenance by approximating unnecessary code with unused code, and he led and conducted the studies for evaluating the approximation. Furthermore, he developed the tool support, except the profiler, which was developed by Juergens et al. [35].

Preliminaries In the context of this contribution, we introduce the term *unnecessary* in relation to *maintenance* (see Section 2.2) and source code. We consider maintenance as *unnecessary* if it is a waste of resources. We consider source code as *unnecessary* if it realizes *useless* (see Section 2.5.1) functionality. We approximate *useless* with *unused* code, based on our explanations in Section 2.5.

Goal The goal of this contribution is to foster our understanding of unused source code and its effects on the activity maintenance. Additionally, we evaluate the approximation of unnecessary source code with unused source code.

Case Study We quantify the amount of unused source code and maintenance actions therein. Furthermore, we quantify maintenance in unused code, and examine the extent of unnecessary maintenance therein. By quantifying unnecessary maintenance, the study shows, how much efforts could have been saved by considering usage indicated by execution profiles. Additionally, we qualitatively analyze the helpfulness of knowledge about unused source code by interviews.

We collect execution profiles on an industrial business information system of the reinsurance company Munich Re. We observe the system for two years. We present the execution profiles to developers[1] of the system, who were maintaining the system for several years, and interview the developers.

Results During the examination period, 25% of all methods in the examined software system were not executed, and thus, remained unused. According to the developers, not all unused code was unnecessary, since it also contained, e.g., error handling routines. But, according to the developers considering execution profiles, 48% of the maintenance actions in unused source code were unnecessary. From this, we conclude that unused code often is unnecessary. Thus, the developers found the insights gained into the usage of the software system under development helpful in practice. In our interviews, developers detected bugs, which were caused by unexecuted code that *should* have been executed.

Only 7.6% of all maintenance actions happen in unused source code, and consequently, 92.4% of maintenance efforts are spent on used source code. Obviously, change requests primarily address *useful* functionality (see Section 2.5.1). We explain this by less execution of unnecessary functionality, and it is more likely to find faults or shortcomings in used code as in unused source code.

We found that the concrete maintenance actions in unused code and used code are similar[2]. Consequently, by considering execution profiles, 3.6% of all maintenance actions could have been saved.

Thesis Contribution This contribution confirms by a case study, that maintenance in unused source code can be a waste of resources, and therefore lays a foundation for this thesis.

We aim at addressing the questions reported by Begel and Zimmermann [21] by the retrospective analysis of usage information. To the best of our knowledge, there is no research about how execution profiles can be used to answer these questions *after* they arise. We close this gap with this contribution. However, we are aware of agile methods that aim at constructively address these questions *before* they arise by closely involving the customers into the development or maintenance process.

1 They had responsibilities, which are similar to the responsibilities of a *product owner* (see 2.2.3.).
2 They range from refactorings to more complex operations.

5.2. Transfer of Unused Source Code to Requirements Artifacts

Published in [4] at ICSME 2014
acc. rate 36%
5 pages
Full paper

The author of this thesis developed the approach and the tool support. He also planned, designed, and conducted the case study.

Preliminaries In this contribution, we assume that functionality is documented in use case documents [146]. This contribution makes use of Latent Semantic Indexing [46] (LSI) to detect links between source code and use case documents. We assume, that use case documents exist and are initially created by requirements engineers before the maintenance of the functionality the use cases describe.

Goal The goal of this contribution is the identification of unused functionality from unexecuted source code, to make execution profiles usable for product owners and to point them to unused functionality. Therefore, we develop a technique for detecting links between use case artifacts and source code.

Approach We detect links between source code and use case documents using LSI. We collect execution profiles on the system under consideration in its productive environment. With links between source code and use case artifacts, and execution profiles, we identify the use case documents that describe functionality that was not executed. More exact, we produce a ranking of use case documents, where use case documents describing unused functionality are ranked highest.

Evaluation We evaluate our approach on a custom business information system employed at Munich Re. The system is written in C#, and its functionality was documented in 46 use case documents. We monitored its execution in the productive environment for over one year. We filtered out unit testing code and did not take external libraries into account. We evaluate our findings with the leading architect of the software system. Among his tasks was change identification/prioritization.

With our approach, we detected two use case documents expressing unused functionality (ranked at positions 1 and 2) and two use case documents, which expressed partly unused features (ranked at positions 4 and 5). We draw two conclusions. First, we confirm our finding of publication [1], that there is unused functionality in custom business information systems. Second, we conclude that our approach is capable of identifying use case documents that describe unused functionality, since these use case documents were ranked highest.

We performed this study on use case documents. However, LSI is not specialized nor bound to this kind of documents. Therefore, our approach technically is not limited to use case documents. We are therefore confident that the presented approach is also applicable in contexts, where functionality is documented in other types of artifacts like user stories. In practice, these documents often exist [147].

Thesis Contribution The understanding of unused functionality helps the product owner performing the tasks *change identification/prioritization* (see Section 2.2), since he can prioritize changes in unused functionality low because this has a high probability of being a waste of resources. The contribution helps the developer in maintaining source code because also he can be pointed to unused functionality during the implementation task, to avoid a waste of resources.

Besides the work of Juergens et al. [35], we are not aware of research work eliciting feature level usage information. Their approach requires developers to manually insert "feature beacons" into the source code. But, their technique is tedious, since potentially hundreds of feature beacons have to be inserted manually and become outdated due to changes. Thus, we propose a technique that allows for tracking the execution of features without modifying source code.

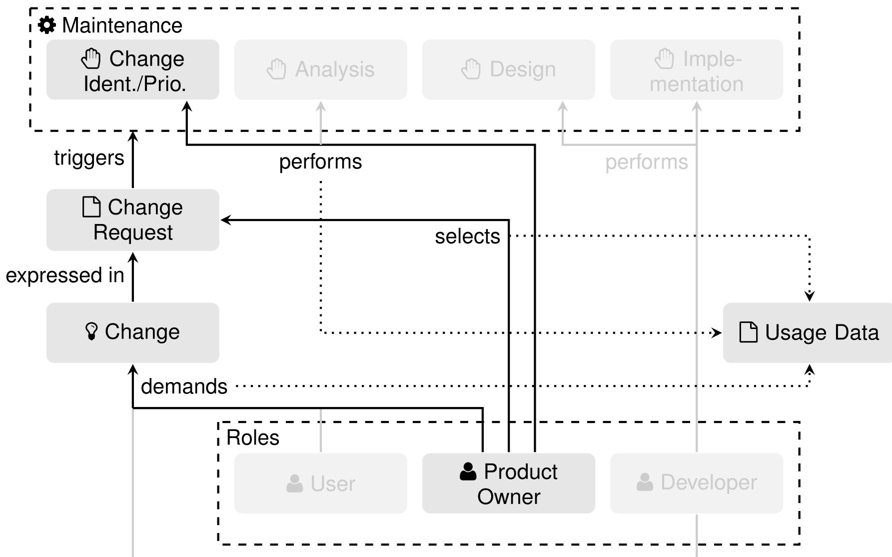


Figure 5.2.: Overview of the tasks of the product owner, where usage data can be considered.

5.3. Considering Usage Data in Maintenance

The product owner and the developers are interested in the actual usage of their software systems (see Section 3.1). But where and how can they profit from usage data? In this section, we describe the areas of application of usage data.

Preliminaries This contribution refers to the techniques presented in publication [1] (see Section 5.1 and publication [4] (see Section 5.2). Thus, the assumptions and limitation of these techniques apply also here.

Goal The goal of this contribution is to integrate the aforementioned techniques into the software maintenance process (see Section 2.2).

Approach We describe in detail, in which tasks and by whom the techniques are used. We focus on the tasks of product owners and developers. Building on the maintenance process described in Section 2.1, we provide an extended maintenance process that integrates tasks, where product owners and developers consider usage data.

Thesis Contribution We embed the techniques described before into the software maintenance process. With this, we provide a deeper understanding of which roles use the presented techniques in which tasks in the maintenance process.

We are not aware of any attempts to embed execution profiles into the software maintenance process. We build upon our own contributions, which were not integrated into the software maintenance process beforehand.

5.3.1. Product Owner

Figure 5.2 shows the tasks of the *product owner* (see Section 2.2), and in which tasks the product owner profits from usage data. Solid black arrows show the tasks, where the product owner profits from usage data. The product owner identifies demanded changes, and selects change requests for implementation during the *change identification/prioritization*.

Performing Change Identification: Demanding Changes Usage data shows the product owner what functionality is used, and what is not. With the approach presented in publication [1] (see Section 5.1), usage data is collected automatically, and with the technique proposed in publication [4] (see Section 5.2), usage data is transferred to documented functionality automatically.

Based on the information the aforementioned techniques yield, the product owner can decide more easily, what functionality is useful, and what is useless. Consequently, the product owner can steer the maintenance efforts to reduce maintenance in useless functionality, which would be a waste of resources.

For useless functionality, the product owner has three possibilities to act. He can demand the removal, preservation, or maintenance of useless functionality. The strategies apply only to the *characteristic* source code (see Section 4.2) that realizes useless functionality.

Removal With its realizing source code, the functionality is removed from the system. The product owner files a change request that requests the removal of the functionality.

Preservation The functionality, and its realizing source code is kept as it is, but it is not maintained. In this case, the product owner does not file a change request. This choice has implications on later change prioritization: the product owner rejects change requests that target the respective functionality, or files new change requests that only other functionality is maintained.

Maintenance The functionality, and its realizing source code are maintained, as if it was useful. In this case, the product owner does not consider usage data.

Exemplary consequences of the application of the aforementioned strategies are explained below.

Removal The functionality and its realizing source code cannot be (re-)used in other functionality, which might be a loss. However, in modern version control systems like SVN or Git, they can be restored, even after removal. Additionally, this strategy prevents a waste of resources caused by maintenance in useless functionality best.

Preservation Old and unmaintained functionality remains in the system. This implies the risk of accidentally maintaining this functionality. Additionally, faults can remain in the system, which may cause faults in case of, e.g., accidental usage by users.

Maintenance If useless functionality is maintained, it can be a waste of resources. However, the functionality can become useful again, due to, e.g., changes in the environment of the system[3]. In this case, the respective functionality is likely to be ready for use.

Removing useless functionality is, among the described strategies, the safest way of preventing a waste of resources. But, also removing functionality causes efforts.

3 For more reasons, see Section 2.2.4

However, the efforts for removing functionality occurs only once. In contrast, if functionality is not removed, it might be maintained more often.

However, removing useless functionality is not always possible. Under the following exemplary circumstances, useless functionality has to be maintained or at least preserved.

Dependencies The source code realizing useless functionality may depend technically on other source code[4]. This may prevent the useless functionality to be removed, because it would impose difficult architectural changes. In these cases, maintaining or preserving functionality can be necessary.

Politics/legislation Due to company politics or legislation, there can be obligatory functionality, regardless of its usefulness. This functionality cannot be removed. In this case, it has to be preserved, but should also be maintained.

The aforementioned strategies steer the maintenance process by triggering changes to the software system, according to the usefulness of functionality. However, this is not the only possibility to react. An alternative to the strategies described above is training the users. Training the users to use other functionality than they actually do can lead to a shift of usefulness in functionality.

The decision for the concrete treatment of unused and useless functionality depends, however, on the project and its context. But, usage data points to functionality that is likely to be useless, and therefore a possible subject to removal.

Performing Change Prioritization: Selecting Change Requests In the change prioritization task, the product owner selects the change requests that are to be implemented. There, the product owner has the option to accept, defer, or reject a change request.

Accept The change request will be implemented as it is.

Defer The change request is not considered immediately, but later.

Reject The change request is not considered for implementation.

Based on usage data about the functionality targeted in the change requests, the product owner can conclude about the usefulness of the respective functionality (see Section 4.4.1). To prevent a waste of resources due to maintenance in useless functionality, we therefore advise the product owner to reject the change request that target unused functionality, since they are likely to provide no value to the users.

Additionally, we suggest to critically review change requests that target unused functionality. The goal of the review is to create new change requests with the actions described for the task *change identification*.

In summary, with usage data, the product owner gains insights into the usefulness of functionality. Based on this information, he can prevent maintenance in useless functionality, to prevent a waste of resources.

Performing Analysis In the *analysis* task, the product owner investigates the feasibility of a change request. This, however, is not influenced by the usage or usefulness of functionality. Therefore, we do not consider this task.

5.3.2. Developer

The tasks of the *developer* comprise the *design* and *implementation* of change requests (see Section 2.2). In these tasks, also the developer can profit from usage data, even

4 E.g., due to interface implementations

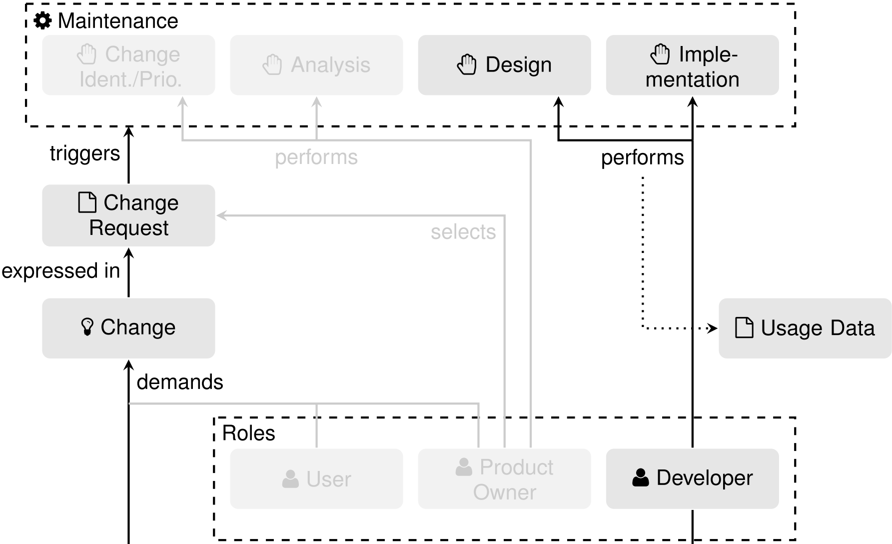


Figure 5.3.: Overview of the tasks of the developer, where usage data can be considered.

if the change requests themselves were assessed based on usage data by the product owner. Figure 5.3 shows the tasks of the developer that are associated with usage data.

Performing Design In the design phase, the developer plans the maintenance caused by a change request in detail. He decides which source code to modify and how to modify it. Usage data, collected by the approach presented in publication [1] (see Section 5.1), shows the developer, what source code is not executed. Unexecuted methods are likely to implement useless functionality [1].

The developer has the choice between modifying source code, or not. Execution profiles give insights into the actual usage of the software system under maintenance. The actual usage indicates, to a certain extent, usefulness. Thus, with insights into the actual usage, the developer has more information about the usefulness of the implemented functionality. Therefore, he can plan which source code to modify, and which not, based on usage data. Therefore, usage data enables the developer to decide, which source code not to change to use less resources. However, due to, e.g., technical dependencies or reuse, maintenance in unexecuted methods can still be necessary.

Performing Implementation Also during the implementation task, the developer faces the risk of maintaining methods that realize useless functionality. With insights into usage data, the developer can decide while modifying the source code, whether the particular modification bears the risk of providing no value to the users.

5.3.3. Process

In this section, we comprise the actions that product owners and developers can take based on usage data. Figure 5.4 outlines a process for considering usage data in the activity software maintenance[5]. Solid arrows indicate the flow through the

5 The publications of this thesis are marked with the symbol 🗎.

process, and dotted arrows illustrate the responsibilities of the roles[6].

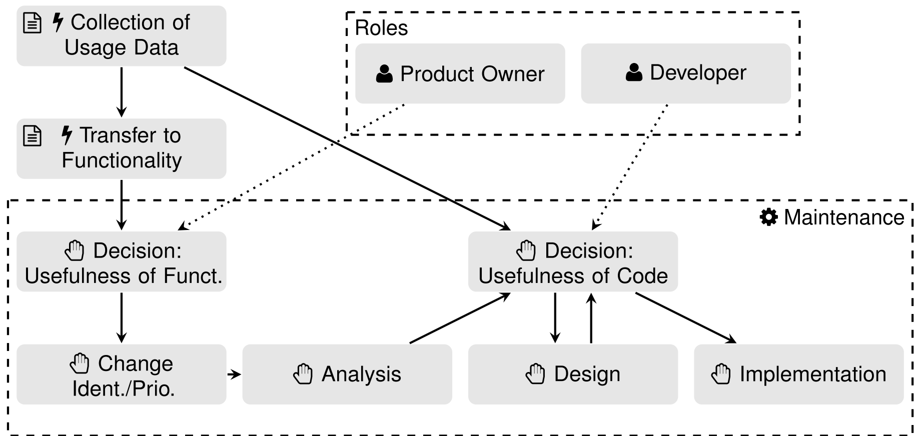


Figure 5.4.: Process and roles for considering usage data in software maintenance.

The process starts with the *collection* of usage data (see Chapter 4 and publication [1] in Section 5.1). After this, usage data is transferred to the level of functionality, using the technique presented in publication [4] (Section 5.2) and publication [5] (Section 5.7). Both aforementioned steps are fully automated.

Based on the insights on usage data on the level of functionality, and his own experience, the product owner *decides about the usefulness of functionality*. Thereby, usage data enables the product owner to gain first insights and an overview of the usefulness of functionality, and to revise his own experience.

The next step is the *change identification/prioritization* task (see Section 2.2). Based on the usefulness of functionality, the product owner identifies changes, and prioritizes changes (as described in Section 5.3.1).

After this, the product owner performs the *analysis* task and examines the feasibility of the changes.

The developer *decides about the usefulness of source code* then, based on usage data on the level of source code, and his experience. Usage data helps the developer to gain first insights into usage, which he usually has not. From missing usage, the developer can revise the usefulness of the maintenance action he plans.

After this, and with knowledge about which methods to maintain, the developer performs the *design* and *implementation task*.

5.3.4. Implications on Resources

In this section, we explain the benefits of considering usage data in maintenance. Thereby, we focus on resources. Our approaches are automated to a large extent. The installation of our profiling technique in the productive environment is, from our experience, done in few hours, and does not require further supervision. Therefore, we assume that the costs of applying our approaches are low.

In the best case, developers and product owners are aware of the usefulness of all unused source code and functionality. Then, they can save the amount of maintenance, which goes into unused, and useless functionality.

6 For the sake of simplicity, we omit the responsibilities for maintenance tasks we already discussed in Section 2.2.

According to publication [1] (see Section 5.1), about half of all maintenance actions in unused source code were providing no value to the users. With 7.6% of all maintenance actions that were performed in unused source code, 3.6% percent of maintenance in the examined business information system were a waste of resources.

But, the additional tasks that are implied by the consideration of usage data, also cause efforts. However, we expect the additional efforts to be small compared to possibly reoccurring maintenance in useless functionality. A deeper investigation is, however, subject to future work.

However, the examined software system was under quality control for several years before our study. The gap between the product owner, developers, and users was small because the system was developed for customers in the same company. Additionally, the developing team was changing only infrequently. In other contexts, e.g., with outsourced development, we expect developing teams to change more often, and the gap between product owners and customers to be larger. Therefore, we expect the amount of savings to be higher in other contexts.

5.4. Uncovered Source Code in Regression Testing

Published in [2] at AST@ICSE 2013
acc. rate 45%
4 pages
Short paper

The author designed and conducted the studies and developed the necessary tool prototypes. He also developed the metric change coverage.

Preliminaries In this contribution, we introduce the metric *change coverage*, which is the fraction of changed and covered methods to all changed methods. If *change coverage* is high for a software system, many changed methods were covered during testing. If *change coverage* is low, few changed methods were covered by tests.

We consider *field faults* as faults occurring in the productive environment of a software system.

Goal We investigate changed source code that was not covered by regression tests with respect to its fault rate. Furthermore, we examine the helpfulness of the metric change coverage, and consequently information about changed, but uncovered methods for test engineers.

Case Study We collect execution profiles in the testing environment of a business information system. Additionally, we collect data about which source code was changed during the activity maintenance, and which source code contributed to field faults.

We investigate the existence and extent of changed source code and uncovered source code. Upon this, we examine how much changed source code was not covered by tests. We explore whether code regions with low change coverage exhibit more field faults than code regions with high change coverage.

The system under consideration is employed at Munich Re, is written in C#, and its size is 340 kLOC. We observed two releases of the system for 14 months in total. We presented the results to a developer, who could also issue the execution of test cases (see the role of the test engineer, Section 2.3).

Results 15% of all methods were changed in both releases respectively of the software system and 34% of all methods were not covered by regression tests. Of the changed methods, 44% in release 1, respectively 45% in release 2, were covered by regression test cases. Thus, 8%, respectively 9% of all methods were changed, but not covered by regression tests. Therefore, we conclude that gaps in the method coverage of changed code exist in the analyzed system.

We observed 23, respectively 10, field faults. Of all field faults, 43% in release 1, and 40% in release 2, were located in modified, but uncovered source code, even though these methods account for only 8%, respectively 9%. Consequently, with 0.53%, respectively 0.21%, the probability of faults is higher in changed-untested methods. This confirms that tested code or code that was not changed during maintenance is less likely to contain field faults.

For several cases, the developer we presented the results to, issued the execution of existing, and the creation of new test cases. Therefore, we conclude that coverage information for changed source code is helpful in practice.

Thesis Contribution This contribution confirms, by a case study, that uncovered, but changed source code contains more faults than other source code, and therefore lays a foundation for this thesis.

It is common knowledge, that changed, but untested source code contains more faults than other source code. However, we are not aware of any empirical evidence for this in our context. We close this gap with this contribution. Existing coverage metrics [148, 149, 150] focus on fine grained coverage information to detect gaps in test coverage, but do not focus on changes. Our approach allows for *assessing* the alignment of the activities regression testing and maintenance with the metric change coverage, which focuses explicitly on changes. To the best of our knowledge, this also closes a gap in literature.

5.5. Transfer of Uncovered Source Code to Regression Test Cases

Published in [3] at AST@ICSE 2014
acc. rate 43%
7 pages
Full paper

The author developed the approach to test regression test case selection. He designed and conducted the evaluating study.

Preliminaries The approach presented in this contribution is also based on LSI (see Section 5.2).

Goal The goal of this contribution is the identification of test cases that cover modified, but yet uncovered methods, to make coverage data better understandable for test engineers, and to help test engineers in their task *regression test case selection*, where they select test cases from existing test suites. In this contribution, we focus on regression test cases, which are denoted in natural language and are executed manually[7] (see Section 2.3).

Approach Using LSI, we detect links between source code and regression test cases. Additionally, we collect execution profiles on the system under test in its testing environment. Based on the links between source code and regression test cases, we identify the test cases for every method, which are likely to cover it. Together with coverage data, we identify test cases that cover modified, but uncovered methods.

Evaluation We evaluate our approach on a custom business information system employed at Munich Re. The system is written in C# and we examined four regression test cases. Among the four test cases, we had one test case with a failing verdict. Additionally, the test cases did not only address the system itself, but also external tools like Excel. The test cases varied in the number of interaction that addressed the software system itself, and other tools.

We focused on the parts of the system that were covered by these test cases. In the examined part of the system, which consisted of 2711 methods, methods were covered by 2.56 test cases on average. Thus, randomly guessing a test case for a method yields a 64% chance of hitting a test case that actually covers the method.

Our approach identified 1.75 test cases for each method on average. For 2444 methods, at least one covering test case was identified. Thus, in 90% of all methods, our approach identifies a test case that covers the respective method, which outperforms randomly guessing.

We furthermore evaluated whether the characteristics of the test cases influence the results. The failing verdict did not influence the results. However, test cases containing more interactions with the system under test, and less with external tools, can be mapped with higher accuracy to methods using our approach.

We conclude that our approach is capable of identifying test cases that cover modified, but yet uncovered method. But, we rely on existing test cases. In case of new or heavily modified functionality, the approach is not applicable.

Thesis Contribution The presented technique selects test cases. Therefore, it directly helps the test engineer in the task *test case selection* (see Section 2.3). Since we identify test cases for methods that are changed, and we propose to change methods that realize useful functionality, we consequently suggest test cases for useful functionality.

Many techniques for regression test case selection rely on logical properties like pre and post conditions in source code, or coverage data from prior test runs [153, 154, 155, 156, 157, 158, 159]. The approach presented in this contribution is based only static analyses of the test suite and source code, and does not need any of the aforementioned information.

7 There is some evidence for manual testing being widely applied in practice [151, 152].

5.6. Considering Coverage Data in Testing

The test engineers show interest in the coverage of their software systems (see Section 3.1). But where and how can test engineers and testers profit from coverage data contained in execution profiles? In this section, we describe the areas of application of coverage data provided by execution profiles.

Preliminaries This contribution refers to the techniques presented in publication [2] (see Section 5.4) and publication [3] (see Section 5.5). Therefore, the limitations and assumptions of these techniques also apply here.

Goal The goal of this contribution is to integrate the techniques presented in publication [2] (see Section 5.4) and publication [3] (see Section 5.5) into the activity regression testing (see Section 2.2).

Approach In this contribution, we describe where and how test engineers profit from *coverage data* provided by execution profiles (see Section 4.3) collected during *regression testing* (see Section 2.3). We provide an extended testing process that integrates the techniques mentioned before.

Thesis Contribution We embed the techniques described before into the regression testing task. With this, we provide a deeper understanding of which roles use the techniques in which activities in the maintenance process.

We are not aware of any attempts to embed execution profiles into the regression testing task. However, we build upon our own contributions, which were not integrated into the regression testing task beforehand.

5.6.1. Test Engineer

In Figure 5.5, we illustrate the tasks of the *test engineer* (see Section 2.3), where coverage data can be beneficial. The test engineer is in charge of the tasks *management* of test suites by adding or removing tests, *creation and modification* of test cases, and the *selection* of test cases for execution.

Coverage data shows the product owner, what was executed during test runs, and what was not (see Section 4.4.2 and publication [2] in Section 5.4). Additionally, with the publication [3], the test engineer gains insights into which regression test cases should be executed to close gaps in test coverage (see Section 5.5).

Managing Test Suites: Adding Tests Based on information about existing test cases that close gaps in the code coverage, the test engineer can add existing test cases to existing test suites. The effect is a larger test suite that covers more source code.

Creating and Modifying Test Cases The approach presented in publication [3] (see Section 5.5) selects only existing test cases. However, for new or heavily modified source code and functionality, test cases might not exist. However, the approach still suggests test cases that are at least likely to cover the source code in question. The test engineer can use this information to modify existing test cases, or create new test cases that eventually lead to the coverage of the respective source code.

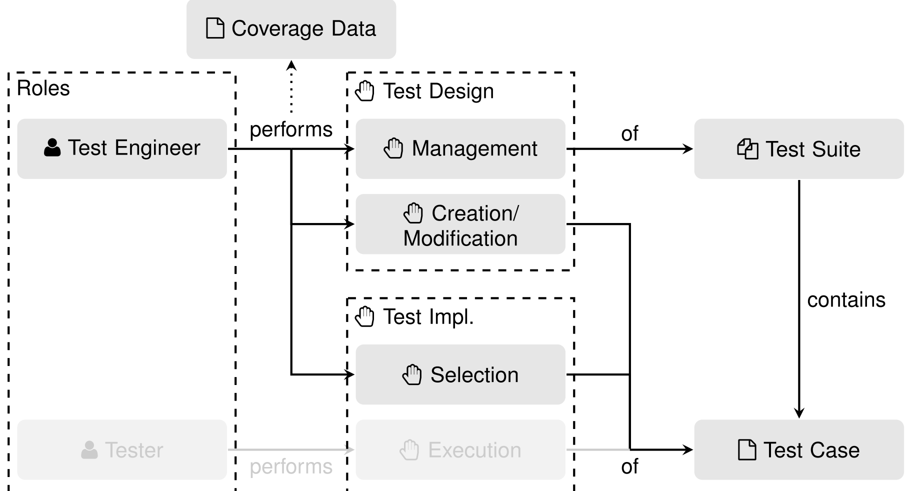


Figure 5.5.: Overview of the tasks of the test engineer, where coverage data can be considered.

Selecting Test Cases Based on coverage data, the approach presented in publication [3] (see Section 5.5) selects test cases that are likely to cover previously uncovered source code. Therefore, the test engineer can use the suggestions for test cases of the approach to select test cases for execution.

5.6.2. Process

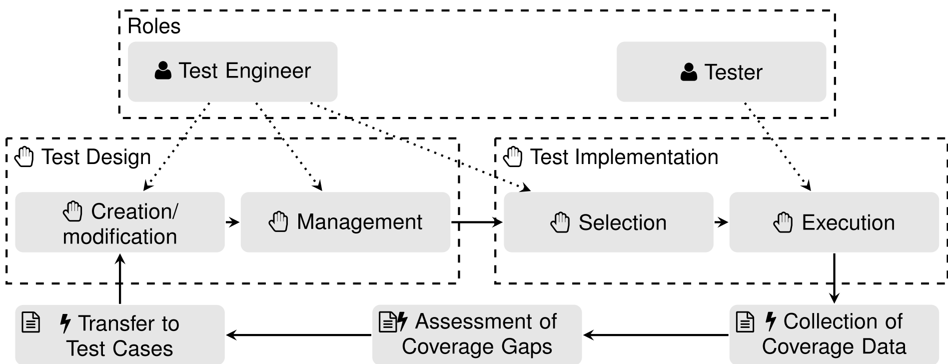


Figure 5.6.: Process and roles for considering coverage data in testing.

Figure 5.6 illustrates the process for considering coverage data in regression testing[8]. Solid arrows indicate the flow through the process, and dotted arrows illustrate the responsibilities of the roles

The process forms a cycle that enables testing in iterations. In the first iteration, coverage data cannot be considered, since no test cases were executed. The process starts with the tasks of the test engineer *creation and modification* of test cases, continues with the *management* of test suites, and the *selection* of test cases. Afterwards, the tester executes the selected test cases (see Section 2.3).

8 The publications of this thesis are marked with the symbol 🗎.

During the *execution* of test cases, coverage data is collected, using the techniques for collecting execution profiles (see publication [1], Section 5.1, Chapter 4). Then, with the techniques of publication [2] (see Section 5.4), gaps in the coverage of source code are *assessed* automatically. Based on these gaps, the approach proposed in the contributions [3] (Section 5.5) and [5] (Section 5.7) automatically *transfers the coverage information to test cases* and suggests existing test cases that are likely to cover the previously uncovered source code.

Based on the information about gaps in coverage, and which test cases are likely to close the gaps, the test engineer *creates or modifies* test cases. The test cases must validate the functionality that is realized in the uncovered methods. These test cases can then be added to test suites during their *management*, with the goal to permanently close the coverage gaps. The *creation or modification* task is especially important, if no test cases exist that cover new or modified functionality.

The test engineer *selects* test cases from the test suites, also with the goal to close gaps in coverage. Thereby, he focuses on test cases that were not executed previously, since performing the same tests on an unmodified system is likely to cover the same source code.

After this, the tester *executes* the test cases that were selected on the basis of coverage information. After this, the cycle starts again, until the desired coverage is reached. The desired coverage heavily depends on the project context and the domain. However, we suggest to at least cover the changes made to the source code during maintenance.

5.6.3. Implications on Resources

In publication [2] (see Section 5.4), we encountered that the probability of faults in changed, but uncovered methods is higher than the probability of faults in changed, and covered methods. In the study presented in the aforementioned contribution, we examined two releases of the same software system. In the following, we explain how many faults were, and could have been found by raising the coverage of changed source code.

5.6.3.1. Assumptions

We assume, that the probability of introducing a fault into a system during maintenance does not depend on whether the method is tested during regression testing afterwards.

But, we assume that testing methods after their modification reveals faults. Additionally, we assume that executing a method during testing yields a chance of revealing a fault. We assume that this probability is the same for all methods.

Therefore, we assume that the detection of faults (test suite effectiveness) scales linearly with the number of covered methods.

5.6.3.2. Approach

We divide the methods of a software system into covered and uncovered methods. The absolute number of covered methods is denoted as m_c, and the absolute number of uncovered method by m_u. The absolute number of faults in covered methods is denoted as f_c and the absolute number of faults in uncovered methods is f_u. The probability of faults in covered methods is p_c, and in uncovered methods p_u. The latter values are calculated by the formulas:

$$p_c = \frac{f_c}{m_c}$$

$$p_u = \frac{f_u}{m_u}$$

The number of faults that would occur in the covered methods, if they were not covered is denoted as f'_c. The absolute number of faults in uncovered methods, if they were covered is denoted as f'_u. We calculate the values by multiplying the absolute number of the methods in the respective category with the probability of faults in the other category:

$$f'_c = m_c \cdot p_u = m_c \cdot \frac{f_u}{m_u}$$

$$f'_u = m_u \cdot p_c = m_u \cdot \frac{f_c}{m_c}$$

The difference in the total number of bugs actually contained in covered methods (f_c) compared to the number of bugs if the methods were uncovered (f'_c) is denoted as d_c. And vice versa, the difference of faults for uncovered methods is denoted as d_u. They are calculated by the following formulas:

$$d_c = f'_c - f_c$$

$$d_u = f_u - f'_u$$

d_c shows, how many more faults would be in the covered methods, if they were uncovered. This means, that d_c shows, how many faults were revealed by covering these methods. We expect $d_c > 0$, since $p_c < p_u$, as reported in publication [2] (see Section 5.4). On the other hand, d_u shows, how many faults could have been found, if all uncovered methods were covered.

From the values above, we compute the total number of faults contained in the considered methods, *before* testing. We denote the total number of faults as F. We compute the total number of faults by adding the number of faults that occurred in uncovered methods (f_u) to the number of faults that would have occurred if the covered methods had been uncovered (f'_c):

$$F = f_u + f'_c$$

Additionally, we calculate the number of faults that could have been detected if all methods were covered F'.

$$F' = d_c + d_u$$

From the total number of faults contained in the considered methods, we calculate the share of faults that could have been detected by covering all methods S.

$$S = \frac{F'}{F}$$

Considered Methods	Release	Covered	m	f	p	f'	d	F	F'	S
all	1	✓	8395	5	0.0006	10.99	−5.99	28.99	15.79	0.54
		–	13 755	18	0.0013	8.19	9.81			
	2	✓	14 138	3	0.0002	11.54	−8.54	18.54	13.72	0.74
		–	8574	7	0.0008	1.82	5.18			
changed	1	✓	1469	5	0.0034	7.81	−2.81	17.81	6.41	0.36
		–	1880	10	0.0053	6.40	3.60			
	2	✓	1553	3	0.0019	3.21	−0.21	7.21	0.48	0.07
		–	1934	4	0.0021	3.74	0.26			
unchanged	1	✓	6899	0	0	4.65	−4.65	12.65	12.65	1
		–	11 875	8	0.0007	0	8			
	2	✓	12 585	0	0	5.69	−5.69	8.69	8.69	1
		–	6640	3	0.0005	0	3			

Table 5.1.: Model instantiations for all and only changed methods, for two releases.

5.6.3.3. Instantiation

We instantiate the described model for *all methods* contained in the study object that we also used in publication [2] (see Section 5.4). The study objects maintenance and test did *not* follow the proposed approaches and processes. In these contributions, we examined two releases of a business information system. We structure the instantiation of our model along these releases. For each release, we present the numbers for the aforementioned variables in Table 5.1.

Considering all Methods In the first release, 23 faults occurred in the productive environment after testing. Five faults occurred in the covered methods, and 18 faults occurred in the uncovered methods. We conclude that about 29 faults existed in the system. From these, about 16 faults could have been detected by covering all methods with tests. Thus, 54% of the faults would have been detected by covering all methods.

In the second release, 10 faults occurred in the productive environment. Three faults occurred in the previously covered methods, and seven faults in uncovered methods. We expect, that about 19 faults were in the system in total. If all methods were covered by tests, about 14 faults, which account for 74% of all faults, would have been detected.

Considering only Changed Methods Our process for considering coverage data suggests considering especially changed methods. Therefore, we instantiate our model for only the methods, that were changed during maintenance. Thereby, we use the same study object as above, which we also used in publication [2] (see Section 5.4).

In the first release, five faults occurred in changed, and covered methods. Ten faults occurred in changed, but uncovered methods. We derive, that about 18 faults reside in the changed methods. By covering all changed methods, about 6 faults could have been detected, which would account for 36% of the faults in changed methods.

In the second release, three faults were located in covered methods, and four in covered methods. The probability of faults in either group of methods is quite similar. This means, that whether methods were covered or not, did not influence the number of faults. By applying our model, we derive, that no fault would have been found by covering all changed methods in this particular release. This becomes apparent by considering the numbers for the expected faults and the faults that actually occurred. While we expect about three faults to be located in the covered methods, three faults actually occurred there. This means, that testing did not reveal these faults.

Considering only Unchanged Methods Considering only unchanged methods, in both releases, there were no faults in covered methods. However, in both releases, there were faults in the uncovered, and unchanged methods. According to our model, the probability of faults in covered methods is $p_c = 0$. Therefore, by applying our model, we assume, that covering unchanged methods detects all faults. This leads, according to our model, to the conclusion, that 100% method coverage leads to the detection of all faults in the unchanged methods of the examined software system. As this statement does not hold in general [145], this shows inaccuracies in our model, which we discuss below in the threats to validity.

5.6.3.4. Threats to Validity

In this section, we describe the threats to validity for the findings produced by our model described above.

External Validity We measured the numbers of covered and uncovered methods and faults therein in one business information system. This prevents generalizability. However, the maintenance and test processes, were typical for business information systems. Additionally, the size and complexity of the examined system was usual for business information systems. Therefore, we believe, that we did not examine a special case.

Internal Validity Our assumptions might be wrong. Especially, we assume that test suite effectiveness scales linearly with method coverage. This assumption is not always valid, since the coverage is not a good predictor for the number of faults revealed [145] (see Section 4.4.2). But, Inozemtseva and Holmes [145] report that a larger number of test cases does influence the number of faults found. In our process to close gaps in coverage, we suggest critically revising the gaps in coverage and to create test cases that close these gaps and *validate* the realized functionality. Therefore, we are confident that increasing coverage with our approach increases the ability of a test suite to reveal faults.

We furthermore assume that the probability of revealing a fault is the same for all covered methods. For the changed methods, we saw that the probability of revealing a fault in covered methods is lower than for all methods together, and for unchanged methods, it is higher. Therefore, the assumption does not hold. This limits the expressiveness of our study. To act against this threat, we presented the numbers for all methods together, and only changed, and unchanged methods, to illustrate the effect.

Construct Validity In the instantiation, we consider only faults that actually occurred in the productive environment of the examined business information

system. Not all faults might have become apparent in the productive environment and still remain unnoticed. Therefore, we underestimate the number of faults. However, the faults that occur and are revealed in the productive environment are the ones that actually influence the user visible behavior. We argue, that these faults are the ones that hurt, and therefore, we consider them.

5.6.3.5. Reflection

The results above indicate differences between all methods, changed methods, and unchanged methods, regarding the probabilities of faults in covered and uncovered methods. In changed methods, coverage is a bad indicator for the absence of faults, while in all methods, the indicator works better, and in unchanged methods, the indicator works, according to our results, perfect. So, the results vary greatly. But where do these differences come from?

Considering all methods, we examine also unchanged functionality. For this functionality, regression test cases exist. However, for the changed functionality, there might be no test cases. Just covering the changed code with existing test cases does not consider the changes. The changes might be subtle, and therefore, they do not even become apparent. Consequently, also faults are not revealed by testing. But testing still covers methods, which are possibly changed. This underpins the observations of Inozemtseva and Holmes [145], who show that coverage is a bad indicator for the effectiveness of test suites.

For unchanged methods, there are test cases that correspond to the realized functionality. Additionally, these test cases are likely to be executed earlier, since they are regression test cases. Therefore, faults in the executed methods are likely to be removed before. However, in unchanged methods, where no test cases exist, there are still faults. These faults cannot be detected in regression testing without covering the realizing methods. However, our model suggests, that covering unchanged methods would detect all faults. However, this is probably too optimistic. But, since faults were either in the unchanged system since the beginning of maintenance, or due to faults introduced by maintenance, there must have been a point in time, where these methods were changed. From this point on, the fault could have been detected, but was not, and remained in the software system.

Therefore, it is important, to critically revise gaps in the coverage of changed methods. It is not enough to run additional existing regression tests. The test engineers have to, based on the results our approaches yield, define new test cases for changed functionality or modify existing test cases accordingly. This emphasizes the importance of the activity *creation/modification* in the process proposed before (see Section 5.6.2).

Additionally, the estimation of the resources that can be saved by considering coverage data according to our approaches and processes, is difficult.

The costs of faults occurring in the productive environment are hardly predictable. They depend on the context of a software system and its domain. For example, in insurance companies, contracts over large amounts of money are made in their systems. Faults can become equally expensive. In safety-critical systems, even people can be harmed.

Since we can hardly estimate the costs of faults, the implications on resources are not predictable. However, since we suggest creating and executing additional tests, this rises the expenses for testing. From our experience, only few tests (about one to five) are created and selected for execution due to the presented analyses. With regression test suites containing ten times this number, or even more, the increase of expenses for testing is small, since all regression tests are usually executed.

The small increase of expenses, and the possibly large costs due to failures, give our approaches the character of a cheap insurance against expensive faults that would have been missed without our approaches.

5.7. Automatically Linking Source Code with Other Artifacts

Published in [5] at RET@ICSE 2015
acc. rate NA
7 pages
Full paper

The author of this thesis developed the approach for configuring LSI. He also designed and conducted the studies.

Preliminaries LSI detects links between artifacts in a document corpus. Its output is a matrix (documents $\times$ documents) containing their similarity (ranging between -1 and 1). We call this matrix *link matrix*.

The contributions [3] and [4] are based on LSI. However, LSI has many configuration options. The accuracy of the links between documents that are detected by LSI heavily varies with the configuration. Additionally, the configuration of LSI has to be done manually by experts for this technique. This limits the applicability of LSI and consequently the applicability of the aforementioned approaches in practice.

Furthermore, there is often no knowledge about correct links in the document corpus in our context, since then we would not need to detect the links between documents automatically.

Goal The goal of this contribution is to provide an automatic approach to configure LSI, which does not rely on knowledge about correct links between documents, so that it produces accurate results.

Approach In a pre-study, we learn that there are heuristic metrics, which can be collected on the link matrices produced by LSI, and which correlate with their accuracy. These metrics do not rely on any knowledge about correct links between documents. Based on the heuristic metrics, we filter out inaccurate link matrices, and by doing so, also configurations, that yield inaccurate results.

Evaluation We evaluate the approach on seven document corpora from industry (Munich Re, NASA) and academia. The contained documents are high- and low-level requirements, use cases, test cases, change requests, and defect reports. We compare the accuracy of the configuration selected by our approach fully automatically to the accuracy of the best possible configuration, and to the accuracy of the configuration chosen manually by other researchers[9].

In one case, our approach selects the best possible configuration. In the other six cases, the approach selects a configuration that yields results with an accuracy that is close to the best possible configuration.

Compared to the configurations chosen by other researchers on the same document corpus, our approach selects also configurations yielding either better, the same, or only slightly worse accuracy.

LSI is not always capable of detecting links between documents accurately. But, our approach selects the configurations that yield nearly the most accurate possible results using LSI.

Thesis Contribution This contribution forms an extension to the contributions [3] and [4]. It provides a mechanism for further automating the approaches presented in these contributions and therefore improves their applicability.

Existing approaches to configuring LSI rely on links between documents that have to be created by system experts [47, 48, 160, 161]. These approaches select the configuration for LSI that reflects these manually created links best. In contrast to these approaches, we present a technique that does not rely on manually created links.

9 Often with knowledge about correct links

CHAPTER 6

Conclusions

Contents

6.1. Summary 82
6.2. Limitations 85
6.3. Future Work 89

To conclude this thesis, we first give a summary of this thesis and describe key takeaways. We furthermore discuss limitations of the presented approaches and give an overview of future work.

6.1. Summary

Context In this thesis, we concentrated on custom business information systems. These systems are "software systems to support forecasting, planning, control, coordination, decision making, and operational activities in organizations" [6], and are developed for users, who are specialized in their working domain. Business information systems provide value to their users by supporting their work-flows in their daily business. In this context, users perceive functionality as useful, if using it increases their productivity, performance, or effectiveness on the job [7].

Business information systems are often in use over years, or even decades [8]. During these long periods of time, the users, environments, or business models change. Therefore, the software systems themselves have to be adapted to these changes. These changes are realized in the maintenance process.

Motivation In this process, product owners, developers, test engineers, and testers are interested in usage and coverage of the business information system, for which they are responsible [21, 20]. Our own observations underpin these questions: we observed, that nearly half of the maintenance actions done in unexecuted source code were a waste of resources [1]. Additionally, we observed that modified, but uncovered methods contain more faults than other methods [2].

We focused on the question: How can product owners gain insights into the usage and test engineers into the coverage of software systems in a lightweight and minimal invasive way?

We presented *execution profiles* to measure usage and coverage. Execution profiles express, which methods (on the level of source code) were executed during a given period of time. If execution profiles are collected in the productive environment of a software system, they express, which methods were executed by the users. Methods implement functionality, and therefore, their usage expresses, which functionality was used. Execution profiles, which are collected in the testing environment of a software system, show, which methods were covered by testing. Execution profiles are lightweight, since they do impose only little additional efforts on product owners, developers, test engineers, and testers. Execution profiles are minimal invasive, since their collection is hardly noticeable and their interpretation integrates seamlessly into existing processes.

Research Objective Motivated by the desire of product owners, developers, test engineers, and testers to gain insights into the usage and coverage, we investigated the research objective: We explore the benefit of analyses of execution profiles collected for business information systems in software maintenance and test.

6.1.1. Contributions

To approach our research objective, we presented two case studies that clarified the extent and effect of unexecuted source code. We furthermore developed three approaches for utilizing execution profiles in the maintenance process. Furthermore, we proposed two processes that integrate execution profiles in the activities maintenance and regression testing. Figure 6.1 shows the contributions of this thesis, separated by the activities software maintenance and regression testing.

Studies To understand the extent and effects of unexecuted source code, we conducted two studies. One about unexecuted source code in the productive environment, and one about unexecuted source code in testing.

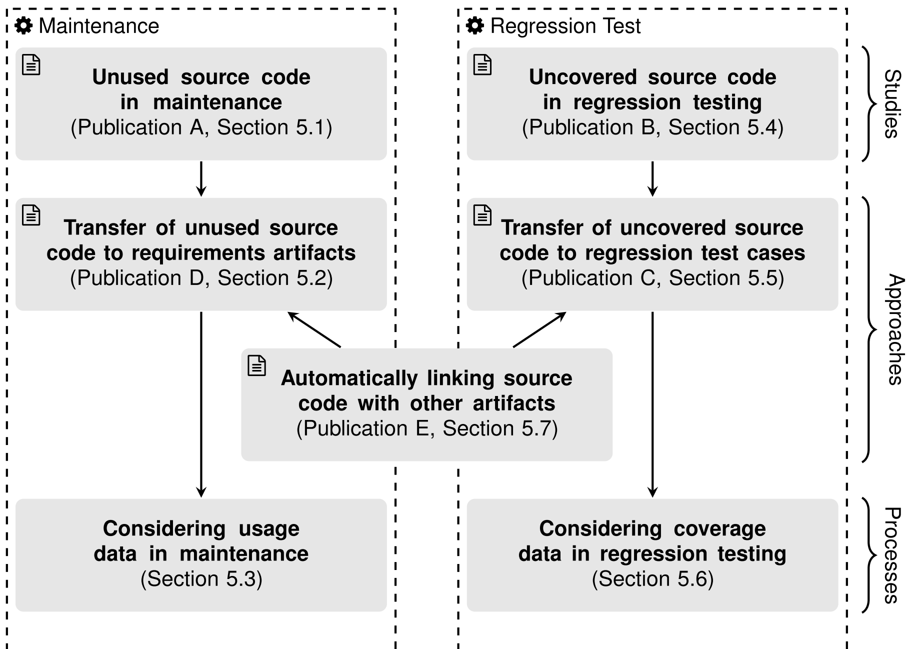


Figure 6.1.: Overview of the contributions.

Our study about unexecuted source code in maintenance (**Publication A**) shows, that one quarter of all methods in the examined systems were not executed over two years. In these methods, 7.6% of the maintenance actions take place. From the maintenance in unexecuted source code, about half of it was a waste of resources. Consequently, 3.6% of the maintenance efforts could have been saved by considering execution profiles. However, the examined software system was developed in house by the same developers for years and the developers were familiar with the domain. Therefore, the developers could estimate the usage of their software system to a certain extent. However, this is not the case, when, e.g., developers are unfamiliar with the software system or its domain, or do not know the software system as good due to fluctuation in the developing teams. Thus, we expect the amount of maintenance that can be saved by considering execution profiles to be higher in other maintenance projects.

Our study about unexecuted source code in regression testing (**Publication B**) revealed that, in the examined software system, about one third of all methods were not executed, and therefore uncovered during regression testing. From the methods that were changed in maintenance, almost half of them were covered by regression tests. But, almost 10% of all methods were changed, but uncovered. In the changed, but uncovered methods, we observed about almost half of the faults that became apparent in the productive environment afterwards. This shows, that in the examined business information system, the probability of faults is higher in changed, but uncovered methods than in other source code. Thus, we suggest that test engineers reflect gaps in the coverage of changed methods critically to avoid faults. This underpins common knowledge. However we are not aware of other research that examined the probability of faults with respect to changes and coverage.

Approaches Building on the insights on unexecuted source code, we provided approaches for product owners, developers, and test engineers, which enable them use execution profiles.

For product owners and developers, we present an approach to identify unused functionality (**Publication D**). We thereby rely on functionality to be documented in requirements artifacts. We employ the information retrieval method Latent Semantic Indexing [46] (LSI) to identify the requirements artifacts that describe functionality, which is not used. Our evaluation of the technique showed, that the transfer of execution profiles to use case documents produces accurate results on the examined business information system. With the results, product owners can prevent the waste of resources caused by maintenance in useless functionality, by critically revising maintenance in unused functionality.

For test engineers, we proposed an approach, based on LSI to automatically select regression test cases that are likely to cover changed, but yet uncovered source code (**Publication C**). Our evaluation showed, that the technique produces accurate results. However, the technique relies on existing test cases. For new or changed functionality, often no test cases exist.

The latter contributions are based on LSI. However, LSI is not a fully automated approach, since it needs to be configured manually. But the accuracy of the results of LSI vary heavily with its configuration. Therefore, we proposed an approach for configuring LSI automatically (**Publication E**). The evaluation of the approach showed, that the configurations selected by the approach yield accurate results.

These approaches show, how product owners, developers, test engineers, and testers can analyze execution profiles automatically and profit from them.

Processes Our goal is to enable product owners, developers, testers, and test engineers to benefit from execution profiles. Therefore, we weave our approaches into the existing maintenance and testing activities.

In maintenance, we propose a process to consider usage data. We weave the automated approaches for collecting usage data (**Publication A**) and the transfer of the information about unexecuted source code in execution profiles to functionality (**Publication D**) into the existing tasks in software maintenance. Based on the results of our automated approaches, product owners and developers decide upon the usefulness of functionality and source code.

In testing, we extend the existing process by adding our automated approaches for collecting coverage data (**Publication A**), assessing gaps in method coverage (**Publication B**), and the transfer of these gaps to test cases (**Publication C**). We conclude, that the test engineer has to decide which tests need to be created or modified to validate functionality that remained uncovered.

6.1.2. Key Takeaways

We formulate the key takeaways of this thesis in a short overview.

There is a gap between users and other stakeholders. Due to the different areas of expertise of the users of custom business information systems and product owners, developers, test engineers, and testers, there is a gap between stakeholders in non-agile maintenance projects. This gap leads to misunderstandings between the stakeholders in charge of maintaining the software system and its users.

Execution profiles are a possibility to gain knowledge about the execution of software systems. With execution profiles, we measure, which source code was executed (*usage data*). If execution profiles are collected in the productive environment, they indicate usage. In case the execution profiles are collected in the testing

environment, they indicate coverage (*coverage data*). Product owners, developers, test engineers and testers can use this information to answer the questions reported by Begel and Zimmermann [21].

Data about usage and coverage is interesting. Product owners, developers, test engineers, and testers want to understand which source code and functionality, and how functionality is executed in their systems [21, 20]. With this knowledge, product owners and developers can understand the actual usage of their software systems. Thus, they understand better what is useful for the users of their software systems (see Section 5.1 and Section 5.2). But also for test engineers and testers, the execution of software systems is of interest. With data about the execution in software systems, they can measure test coverage. With knowledge about test coverage, test engineers and testers have insights into where testing efforts should be spent (see Section 5.4 and Section 5.5).

Considering execution profiles yields benefits. Source code realizes functionality and is covered by test cases. With usage data, we can derive, which functionality was not used (see Section 4.2). Maintenance in unused functionality bears a high risk of being a waste of resources (see Section 5.1). With usage data, product owners can detect unused source code and functionality (see Section 5.2 and Section 5.3). Thus, and since product owners are in charge of the economic success of a software system, usage data is of interest to them. With usage data, they can reduce the waste of resources.

Source code that was not covered by tests is certainly not tested (see Section 4.3). It is the goal of test engineers and testers to reveal faults. With coverage data, they can detect gaps in coverage, especially in *change coverage* (see Section 5.4). Based on this information, test engineers can select or modify existing test cases (see Section 5.5), or create new test cases, with the goal to test modified source code (see Section 5.6). However, we advise that just covering source code does not mean that it was validated against the customer's expectations.

Thus, we conclude, that product owners, developers, test engineers, and testers benefit from the consideration of execution profiles. If the questions reported by Begel and Zimmermann [21] occur[1], we advise to use execution profiles to gain insights into the actual usage and coverage of the system under maintenance or test, because with execution profiles, product owners, developers, test engineers, and testers can at least partially answer the questions of Begel and Zimmermann [21].

6.2. Limitations

Beneath the benefits of execution profiles, there are also some limitations, which we reflect in this section.

6.2.1. Limitation to Existing Artifacts

Execution profiles can only be collected on existing source code. Furthermore, our approaches to transfer the information contained in execution profiles to functionality and test cases rely on existing artifacts. Thus, it is not possible with the proposed approaches, to e.g., suggest test cases that do not yet exist.

1 We expect these question to be more likely to occur in non-agile contexts.

Limitation to Existing Functionality In execution profiles, users do not directly express, what is useless for them. But, from execution profiles, we suggest to measure the usage of functionality, and then draw conclusions about their usefulness. This limits the scope of the proposed approaches to existing functionality. So, it is not possible to derive new functionality from the usage of software systems.

Additionally, to identify unused functionality, we select existing requirements documents, that describe unused functionality. However, the functionality of a software system is not necessarily documented. In this case, our approaches cannot identify unused functionality. Additionally, the proposed approaches are not capable of identifying functionality that *would* be useful for the users, if it *was* realized in the software system under examination.

Limitation to Existing Test Cases Based on gaps in coverage, we identify existing test cases, which are likely to cover the source code in question. However, if there is no such test case, our approaches are not capable of suggesting a new test case. Therefore, our approaches are limited to existing test cases. The field of test suite augmentation focuses on extending existing test suites to, e.g., gain higher coverage, or more effective test suites [162, 163, 164]. Due to the limitation to existing test cases, our approaches cannot be used for test suite augmentation directly. The approaches can be used for giving hints, for which functionality test cases should be created.

6.2.2. Limited accuracy

The relation of execution, or usage and coverage, to usefulness and validation is usually not straightforward. We rely on a correlation between these properties, but, the relation is inaccurate. Additionally, we cannot conclude about single users, the frequency of usage, or the order in which source code or functionality is used.

Usage and Usefulness The relation of usefulness and usage is described in the TAM (see Section 2.5). There is a correlation between the usage and the usefulness of functionality. However, the relation is not necessarily causal or strictly logical and usefulness is not the only reason for usage (see Section 4.4.1).

Our own study showed, that almost half of the maintenance in unused source code was still important (see **Publication A**, Section 5.1). This shows the limitations of accuracy when inferring usefulness from usage. Therefore, we require product owners and developers to decide manually about the usefulness of functionality.

We discussed some examples where usage does not indicate usefulness (see Section 4.4.1). Users might not know functionality they would perceive as useful and therefore not use it, and even use other functionality as a substitute. Additionally, users might use functionality, even though they perceive it as not useful either accidentally, or as a substitute for missing functionality.

Business information systems support the workflows of users in their daily business. However, if the workflows itself are designed badly, also the software systems that implement the workflows do not support the users well. This leads to less usefulness of these software systems in the perception of the users. However, the users might be obliged to use functionality. In this case, and in general, where users are obliged to use functionality[2], usage does not allow for conclusions about usefulness. Additionally, in these cases, usage might even lead to wrong conclusions about usefulness.

2 This would be extrinsic motivation, in contrast to the more intrinsic view of the TAM.

Coverage and Validation Plain coverage of source code does not mean the realized functionality was tested. An extreme example are test cases that do execute source code, but do not check for any results. Inozemtseva and Holmes [145] report that coverage is not a good indicator for the effectiveness of test suites. However, in this thesis, we use missing coverage as an indicator for what was certainly not tested.

The relation between covered and tested source code depends on whether test cases actually validate the functionality of the covered source code. We acknowledge that covered source code is not necessarily tested. Therefore, just raising coverage might not yield the desired result of revealing more faults. Therefore, we suggest monitoring the effectiveness of test cases.

However, with coverage data, we can identify methods, that were not covered. Therefore, these methods are definitely untested.

6.2.3. Assumptions

In this thesis, we stated several assumptions and narrowed down the context. We reflect these assumptions critically.

Custom Business Information Systems We considered only custom business information systems. These systems are employed in the working environment of users. There, they serve the users in tasks that are specialized to their domain. This excludes off-the-shelf software like text processing systems or operating systems. In these systems, users are far more diverse than in business information systems. Additionally, the experience of the users in the functionality of these systems varies. So, for example in text processing, there can be expert users, who master the area of text processing, whilst other users are beginners. These users use the software systems differently.

Linton et al. [112, 113] present a technique, where they monitor the interactions of users to *compare* them with the interactions of expert users. They found out, that expert users of a text processing tool use the functionality, which novice users use, with the same frequency. However, expert users extend their repertoire of functionality. This means, that the amount of unused functionality is reduced by expert users. However, the functionality that is not used by novice users, but expert users is *not* useless. However, business information systems also different users. They vary not as much in their expertise with the system, but in the tasks they perform. For the different users, different functionality can be useful. Therefore, we expect the relation of usage and usefulness in off-the-shelf software is comparable to the relation in business information systems.

However, the limitation to custom business information systems excludes the majority of installed software systems, which impacts the generalizability of our findings negatively.

Systems with User Interfaces We assume, that users decide, which functionality they use, by directly interacting with software systems via a user interface. However, not all business information systems are built for this way of usage. There are, e.g., batch systems, which are triggered remotely by other software systems. These systems are either not triggered by users at all (fully automated batch jobs), or only indirectly, if the user executed functionality that triggers another software system. However, execution profiles also in these systems show usage and coverage. Therefore, the aforementioned techniques can also be applied to these systems, with

the limitation, that not the usefulness for users is in the focus, but the usefulness for underlying business purposes.

Long Living Software Systems that are Actively Maintained We developed our techniques for long living software systems that are actively maintained. This is often the case for business information systems [8]. However, not all business information systems are long living, or are actively maintained. In these systems, the resources that are spent on maintenance are less. Therefore, in shorter living systems, or systems that are not actively maintained, the benefit of the proposed approaches is less. However, the approaches can still be applied in shorter living software systems that are actively maintained. In systems that are not maintained, there are no stakeholders to perform the tasks required by our approaches (see Section 5.3 and Section 5.6). Therefore, our techniques cannot be applied there. But, if the systems are not maintained, there is also no risk of wasting resources in their maintenance.

Existing Processes We rely on existing process as described in Section 2.1, Section 2.2, and Section 2.3. However, different processes and approaches to maintenance can be established. One example are agile methods. Agile methods focus on the reduction of waste in maintenance [12, 13]. Thereby, these methods focus on frequent discussions of developing stakeholders and users or customers. However, also in agile contexts, the risk of implementing obsolete requirements exists [22]. Thus, we suggest to complement the discussions between developing stakeholders and customers or users with the proposed approaches. One example, how this can be done, is to retrospectively consider, whether the functionality that was implemented is used as expected. With the techniques presented in this thesis, product owners can detect obsolete requirements retrospectively.

An emerging paradigm in software development and maintenance are DevOps [165, 166]. As we already described (see Section 4.1.4), operations are responsible for collecting execution profiles. With DevOps, developers and operations cooperate closely, or these roles are even fulfilled by the same person. Thus, developers are able to gain feedback about usage rapidly, which enables them to bring in this knowledge into discussions with the product owner or other stakeholders.

Manual System Tests We assume, that regression testing is often done manually. However, there is a trend towards automating regression tests, also on the system level. However, there is some evidence, that there are still software systems, which are tested manually [151, 152]. Therefore, we regard this assumption as valid. But, if tests are performed automatically, their execution costs drop, and, e.g., all regression test cases can be run in one night. In this case, our technique for selecting regression test cases (**Publication C**, Section 5.5) does not need to be applied. However, there can still be gaps in the coverage of changed source code. These gaps still yield a higher probability of faults. Test engineers aim at detecting faults. Therefore, we conclude that our approach to investigate gaps in the coverage of changed source code, which is presented in **Publication B** (see Section 5.4) is beneficial of test engineers.

Functionality is Implemented in Methods We assume, that functionality is distinguishable by the methods that implement it. However, this might be wrong, and therefore, execution profiles on the level of methods too coarse grained. But, we rely on the findings of Juergens et al. [35], who report they were able to find methods that were suitable for characterizing functionality in most cases.

6.2.4. Generalizability

We performed the major part of our studies in the context of the reinsurance company Munich Re, and made several assumptions as described above. This limits the generalizability of our findings we gathered in our studies and evaluations (see Section 6.1.1). However, Munich Re is an experienced company in software development and applies common techniques, processes, and strategies for developing software. Thus, we expect Munich Re to be similar to other software developing companies. Therefore, we are confident that our findings are transferable also to other companies developing custom business information systems, which fulfill our assumptions.

6.3. Future Work

Building on this thesis, there arise some new topics in research and practice. Many of these topics arise from the limitations we presented before, and augment the applicability of our approaches.

Combination with other Techniques Execution profiles and analyses of them are optimized to cause only small additional efforts, but reduce the efforts in the software maintenance process. This gives room for more extensive user involvement. To overcome the limitations of our approaches to existing and documented functionality, our analyses can be combined with user involvement techniques, which require direct interaction with users (see Section 3.2.2), to gain insights in the usefulness of functionality that is not yet implemented into a software system.

Agile Processes The reduction of waste is central for agile processes. However, to the best of our knowledge, there are no reliable reports about whether agile processes really avoid waste, or how much waste they avoid. Especially, when it comes to *extra features* [167], no evidence for the effectiveness for agile methods is given. Since one of the potential benefits of the application of usage data in maintenance is the elimination of extra features, we suggest to examine agile methods for their capabilities regarding the reduction of extra features. This would also allow for a more detailed comparison of the techniques presented in this thesis and agile methods.

Usage of Data The approaches we proposed are limited to source code. Therefore, they are not able to consider the *data* that is viewed or edited by users. To gain insights into the usage of data, for example, log file analysis techniques can be employed (see Section 3.2.2): by analyzing log files of servers providing data, product owners can gain insights into which data is viewed or edited. However, with a technique for collecting information about the usage of data, we expect our approaches to provide more detailed information.

Usage of Maintenance Artifacts One particular instance of considering data is monitoring the artifacts used during the software maintenance process. Software maintenance not only produces source code, but also other artifacts like requirements documents or test cases. These artifacts aim at facilitating activities like project management, maintenance, or testing. However, in practice, artifacts can be created that are not relevant for any activity[3]. For example, test cases might be

3 Agile methods also focus on the reduction of irrelevant artifacts [167].

created that are never executed. The creation of these artifacts is consequently a waste of resources. Therefore, the identification of these unused artifacts can also be helpful in practice to reduce a waste of resources.

Bibliography

[1] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Prommer, "How much does unused code matter for maintenance?" in *International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1102–1111, Reprinted with permission.

[2] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer, "Did we test our changes? assessing alignment between tests and development in practice," in *International Workshop on Automation of Software Test (AST)*. IEEE, 2013, pp. 107–110, Reprinted with permission.

[3] S. Eder, B. Hauptmann, M. Junker, R. Vaas, and K.-H. Prommer, "Selecting manual regression test cases automatically using trace link recovery and change coverage," in *International Workshop on Automation of Software Test (AST)*. ACM, 2014, pp. 29–35, Reprinted with permission.

[4] S. Eder, H. Femmer, B. Hauptmann, and M. Junker, "Which features do my users (not) use?" in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 446–450, Reprinted with permission.

[5] ——, "Configuring latent semantic indexing for requirements tracing," in *International Workshop on Requirements Engineering and Testing (RET)*. IEEE, 2015, pp. 27–33, Reprinted with permission.

[6] P. Bocij, A. Greasley, and S. Hickie, *Business information systems: technology, development and management*. Pearson Education, 2008.

[7] P. Legris, J. Ingham, and P. Collerette, "Why do people use information technology? a critical review of the technology acceptance model," *Information & Management*, vol. 40, no. 3, pp. 191–204, 2003.

[8] Z. Durdik, B. Klatt, H. Koziolek, K. Krogmann, J. Stammel, and R. Weiss, "Sustainability guidelines for long-living software systems," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 517–526.

[9] IEEE, "IEEE standard for software maintenance (IEEE Std 1219-1998)," 1998.

[10] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap," in *Future of Software Engineering (FOSE)*. ACM, 2000, pp. 73–87.

[11] F. Paetsch, A. Eberlein, and F. Maurer, "Requirements engineering and agile software development," in *International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 2003, pp. 308–313.

[12] M. Poppendieck and T. Poppendieck, *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley, 2007.

[13] C. Ebert, P. Abrahamsson, and N. Oza, "Lean software development," *IEEE Software*, vol. 29, no. 5, pp. 22–25, 2012.

[14] M. Fowler, "Yagni," http://martinfowler.com/bliki/Yagni.html, 2015, accessed: March 28 2016.

[15] S. Arnold and P. White, "Leaner software development using devops," http://www.ibm.com/developerworks/rational/library/leaner-software-development-with-the-aid-of-collaborative-lifecycle-management/, 2014, accessed: March 28 2016.

[16] J. Johnson, "Roi, it's your job," Keynote at XP, 2002.

[17] M. Cohn, "Are 64% of features really rarely or never used?" https://www.mountaingoatsoftware.com/blog/are-64-of-features-really-rarely-or-never-used, 2015, accessed: March 28 2016.

[18] R. Bergman, "Embracing nihilism as a software development philosophy and the birth of the big book of dead code," in *Agile Conference (AGILE)*. IEEE, 2012, pp. 86–91.

[19] R. Lichty, "The most convincing reason to change from waterfall to agile," http://ronlichty.blogspot.de/2013/07/the-most-convincing-reason-to-change.html, 2013, accessed: March 28 2016.

[20] J. Heiskari and L. Lehtola, "Investigating the state of user involvement in practice," in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2009, pp. 433–440.

[21] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 12–13.

[22] K. Wnuk, T. Gorschek, and S. Zahda, "Obsolete software requirements," *Information and Software Technology*, vol. 55, no. 6, pp. 921–940, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584912002364

[23] T. Dyba and T. Dingsoyr, "What do we know about agile software development?" *IEEE Software*, vol. 26, no. 5, pp. 6–9, 2009.

[24] W. Raschke, M. Zilli, J. Loinig, R. Weiss, C. Steger, and C. Kreiner, "Where does all this waste come from?" *Journal of Software: Evolution and Process*, vol. 27, no. 8, pp. 584–590, 2015.

[25] M. Ikonen, P. Kettunen, N. Oza, and P. Abrahamsson, "Exploring the sources of waste in kanban software development projects," in *Conference on Software Engineering and Advanced Applications (EUROMICRO)*, 2010, pp. 376–381.

[26] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Quarterly*, vol. 13, no. 3, pp. 319–340, 1989.

[27] ——, "User acceptance of information technology: system characteristics, user perceptions and behavioral impacts," *International Journal of Man-Machine Studies*, vol. 38, no. 3, pp. 475–487, 1993.

[28] J. Kramer, S. Noronha, and J. Vergo, "A user-centered design approach to personalization," *Communications of the ACM*, vol. 43, no. 8, pp. 44–48, 2000.

[29] B. W. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.

[30] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, 2000.

[31] R. L. Glass, "Maintenance: Less is not more," *IEEE Software*, vol. 15, no. 4, pp. 67–68, 1998.

[32] D. Yeh and J.-H. Jeng, "An empirical study of the influence of departmentalization and organizational position on software maintenance," *Journal of*

Software Maintenance and Evolution: Research and Practice, vol. 14, no. 1, pp. 65–82, 2002.

[33] H. Rombach, B. T. Ulery, and J. D. Valett, "Toward full life cycle control: Adding maintenance measurement to the sel," *Journal of Systems and Software*, vol. 18, no. 2, pp. 125–138, 1992.

[34] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett, "Understanding and predicting the process of software maintenance release," in *International Conference on Software Engineering (ICSE)*. IEEE, 1996, pp. 464–474.

[35] E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. Prommer, "Feature profiling for evolving systems," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2011, pp. 171–180.

[36] M. J. Gallivan and M. Keil, "The user–developer communication process: a critical case study," *Information Systems Journal*, vol. 13, no. 1, pp. 37–68, 2003.

[37] J. Coughlan and R. D. Macredie, "Effective communication in requirements elicitation: A comparison of methodologies," *Requirements Engineering*, vol. 7, no. 2, pp. 47–60, 2002.

[38] H. Rombach and B. Ulery, "Improving software maintenance through measurement," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 581–595, 1989.

[39] G. Fischer, "User modeling in human-computer interaction," *User Modeling and User-Adapted Interaction*, vol. 11, no. 1–2, pp. 65–86, 2001.

[40] B. P. Lientz and E. B. Swanson, "Problems in application software maintenance," *Communications of the ACM*, vol. 24, no. 11, pp. 763–769, 1981.

[41] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *International Conference on Software Engineering (ICSE)*. IEEE, 2005, pp. 284–292.

[42] N. Nagappan, B. Murphy, and V. R. Basili, "The influence of organizational structure on software quality: An empirical case study," in *International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 521–530.

[43] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[44] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," *SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 86–96, 2004.

[45] O. Traub, S. Schechter, and M. D. Smith, "Ephemeral instrumentation for lightweight program profiling," School of engineering and Applied Sciences, Harvard University, Tech. Rep., 2000.

[46] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *Journal of the Association for Information Science and Technology*, vol. 41, no. 6, pp. 391–407, 1990.

[47] M. Lormans and A. van Deursen, "Can LSi help reconstructing requirements traceability in design and test?" in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2006, pp. 10–56.

[48] S. K. Sundaram, J. H. Hayes, and A. Dekhtyar, "Baselines in requirements tracing," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–6, 2005.

[49] R. B. Bradford, "An empirical study of required dimensionality for large-scale latent semantic indexing applications," in *Conference on Information and Knowledge Management (CIKM)*. ACM, 2008, pp. 153–162.

[50] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, *Factors Impacting the Inputs of Traceability Recovery Approaches*. Springer, 2012, pp. 99–127.

[51] A. Garron and A. Kontostathis, "Applying latent semantic indexing on the trec 2010 legal dataset," in *Text Retrieval Conference (TREC)*. National Institute of Standards and Technology, 2010.

[52] G. Bavota, A. De Lucia, R. Oliveto, A. Panichella, F. Ricci, and G. Tortora, "The role of artefact corpus in LSI-based traceability recovery," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. IEEE, 2013, pp. 83–89.

[53] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," in *International Conference on Program Comprehension (ICPC)*, vol. 0. IEEE, 2008, pp. 103–112.

[54] IEEE, "IEEE/eia standard industry implementation of international standard ISO/IEC 12207: 1995 (ISO/IEC 12207) standard for information technology software life cycle processes (IEEE/EIA 12207.0-1996)," 1998.

[55] ——, "Systems and software engineering – vocabulary (ISO/IEC/IEEE 24765:2010(E))," 2010.

[56] ——, "IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990)," 1990.

[57] F. Deissenboeck, "Continuous quality control of long-lived software systems," Ph.D. dissertation, Technische Universität München, 2009.

[58] J. M. Bass, "Agile method tailoring in distributed enterprises: Product owner teams," in *International Conference on Global Software Engineering (ICGSE)*. IEEE, 2013, pp. 154–163.

[59] C.-P. Lin and A. Bhattacherjee, "Extending technology usage models to interactive hedonic technologies: a theoretical model and empirical test," *Information Systems Journal*, vol. 20, no. 2, pp. 163–181, 2010.

[60] V. Rajlich, "Software change and evolution," in *International Conference on Software Engineering (ICSE)*. ACM, 1999, p. 695.

[61] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON)*. ACM, 2008, pp. 304–318.

[62] ISTQB, "Glossary of testing terms," 2014.

[63] B. Hauptmann, M. Junker, S. Eder, C. Amann, and R. Vaas, "An expert-based cost estimation model for system test execution," in *International Conference on Continuous Software Process Improvement (ICSSP)*. ACM, 2014, pp. 159–163.

[64] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke, "Regression test selection of manual system tests in practice," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 309–312.

[65] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[66] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, 1997.

[67] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[68] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 184–208, 2001.

[69] U. Abelein and B. Paech, “Understanding the influence of user participation and involvement on system success — a systematic mapping study,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 28–81, 2015.

[70] J. Mann, “It education’s failure to deliver successful information systems: Now is the time to address the it-user gap,” *Journal of Information Technology Education: Research*, vol. 1, no. 1, pp. 253–267, 2002.

[71] D. Tuffley, “Exploring the it-user gap: towards developing communication strategies,” in *Qualitative Research in IT & IT in Qualitative Research (QualIt)*. Griffith University (IIIS), 2005.

[72] J. Grudin, “Interactive systems: Bridging the gaps between developers and users,” *IEEE Computer*, vol. 24, no. 4, pp. 59–69, 1991.

[73] A. Tiwana and M. Keil, “The one-minute risk assessment tool,” *Communications of the ACM*, vol. 47, no. 11, pp. 73–77, 2004.

[74] G. Mrenak, “Evolving concepts, or why users often don’t recognize the software they asked for,” in *Washington Ada Symposium on Ada (WADAS)*. ACM, 1990, pp. 17–22.

[75] D. A. Adams, R. R. Nelson, and P. A. Todd, “Perceived usefulness, ease of use, and usage of information technology: a replication,” *MIS quarterly*, vol. 16, no. 2, pp. 227–247, 1992.

[76] G. H. Subramanian, “A replication of perceived usefulness and perceived ease of use measurement*,” *Decision Sciences*, vol. 25, no. 5–6, pp. 863–874, 1994.

[77] P. J. Hu, P. Y. Chau, O. R. L. Sheng, and K. Y. Tam, “Examining the technology acceptance model using physician acceptance of telemedicine technology,” *Journal of management information systems*, vol. 16, no. 2, pp. 91–112, 1999.

[78] S. Taylor and P. A. Todd, “Understanding information technology usage: A test of competing models,” *Information Systems Research*, vol. 6, no. 2, pp. 144–176, 1995.

[79] L. Damodaran, “User involvement in the systems design process-a practical guide for users,” *Behaviour & Information Technology*, vol. 15, no. 6, pp. 363–377, 1996.

[80] M. Harris and H. Weistroffer, “A new look at the relationship between user involvement in systems development and system success,” *Communications of the Association for Information Systems*, vol. 24, no. 1, pp. 739–756, 2009.

[81] N. Seyff, F. Graf, and N. Maiden, “Using mobile re tools to give end-users their own voice,” in *International Requirements Engineering Conference (RE)*. IEEE, 2010, pp. 37–46.

[82] N. Seyff, G. Ollmann, and M. Bortenschlager, “Appecho: A user-driven, in situ feedback approach for mobile platforms and applications,” in *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. ACM, 2014, pp. 99–108.

[83] N. A. Qureshi, N. Seyff, and A. Perini, “Satisfying user needs at the right time and in the right place: a research preview,” in *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*. Springer, 2011, pp. 94–99.

[84] K. Schneider, S. Meyer, M. Peters, F. Schliephacke, J. Mörschbach, and L. Aguirre, “Feedback in context: supporting the evolution of it-ecosystems,” in *International Conference on Product Focused Software Process Improvement (PROFES)*. Springer, 2010, pp. 191–205.

[85] W. Maalej, H.-J. Happel, and A. Rashid, “When users become collaborators: Towards continuous and context-aware user input,” in *Conference on Object*

Oriented Programming Systems Languages and Applications (OOPSLA). ACM, 2009, pp. 981–990.

[86] W. Maalej and D. Pagano, "On the socialness of software," in *International Conference on Dependable, Autonomic and Secure Computing (DASC)*. IEEE, 2011, pp. 864–871.

[87] S. Kujala, M. Kauppinen, L. Lehtola, and T. Kojo, "The role of user involvement in requirements quality and project success," in *International Requirements Engineering Conference (RE)*. IEEE, 2005, pp. 75–84.

[88] M. Bano and D. Zowghi, "A systematic review on the relationship between user involvement and system success," *Information and Software Technology*, vol. 58, no. 0, pp. 148–169, 2015.

[89] D. Pagano, "Towards systematic analysis of continuous user input," in *International Workshop on Social Software Engineering (SSE)*. ACM, 2011, pp. 6–10.

[90] D. Pagano and B. Brügge, "User involvement in software evolution practice: A case study," in *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 953–962.

[91] A. J. Ko, M. J. Lee, V. Ferrari, S. Ip, and C. Tran, "A case study of post-deployment user feedback triage," in *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2011, pp. 1–8.

[92] T. Lethbridge, S. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empirical Software Engineering*, vol. 10, no. 3, pp. 311–341, 2005.

[93] J. S. Dumas and J. Redish, *A practical guide to usability testing*. Intellect Books, 1999.

[94] M. Maguire, "Methods to support human-centred design," *International Journal of Human-Computer Studies*, vol. 55, no. 4, pp. 587–634, 2001.

[95] M. Maguire and N. Bevan, *User Requirements Analysis*. Springer, 2002, pp. 133–148.

[96] K. A. Ericsson and H. A. Simon, "How to study thinking in everyday life: Contrasting think-aloud protocols with descriptions and explanations of thinking," *Mind, Culture, and Activity*, vol. 5, no. 3, pp. 178–186, 1998.

[97] M. van den Haak, M. D. Jong, and P. J. Schellens, "Retrospective vs. concurrent think-aloud protocols: Testing the usability of an online library catalogue," *Behaviour & Information Technology*, vol. 22, no. 5, pp. 339–351, 2003.

[98] W. P. J. Eveland and S. Dunwoody, "Examining information processing on the world wide web using think aloud protocols," *Media Psychology*, vol. 2, no. 3, pp. 219–244, 2000.

[99] M. Ramal, R. de Moura Meneses, and N. Anquetil, "A disturbing result on the knowledge used during software maintenance," in *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2002, pp. 277–286.

[100] M. Nørgaard and K. Hornbæk, "What do usability evaluators do in practice?: An explorative study of think-aloud testing," in *Conference on Designing Interactive Systems (DIS)*. ACM, 2006, pp. 209–218.

[101] M. Y. Ivory and M. A. Hearst, "The state of the art in automating usability evaluation of user interfaces," *ACM Computing Surveys*, vol. 33, no. 4, pp. 470–516, 2001.

[102] L. A. Granka, T. Joachims, and G. Gay, "Eye-tracking analysis of user behavior in WWW search," in *International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 2004, pp. 478–479.

[103] L. Lorigo, M. Haridasan, H. Brynjarsdóttir, L. Xia, T. Joachims, G. Gay, L. Granka, F. Pellacini, and B. Pan, "Eye tracking and online search: Lessons learned and challenges ahead," *Journal of the American Society for Information Science and Technology*, vol. 59, no. 7, pp. 1041–1052, 2008.

[104] S. Huang and P. Miranda, "Incorporating human intention into self-adaptive systems," in *International Conference on Software Engineering (ICSE)*, vol. 2. IEEE, 2015, pp. 571–574.

[105] R. Atterer, M. Wnuk, and A. Schmidt, "Knowing the user's every move: User activity tracking for website usability evaluation and implicit interaction," in *International Conference on World Wide Web (WWW)*. ACM, 2006, pp. 203–212.

[106] D. M. Hilbert and D. F. Redmiles, "An approach to large-scale collection of application usage data over the internet," in *International Conference on Software Engineering (ICSE)*. IEEE, 1998, pp. 136–145.

[107] ——, "Agents for collecting application usage data over the internet," in *International Conference on Autonomous Agents (AGENTS)*. ACM, 1998, pp. 149–156.

[108] D. M. Hilbert, "Large-scale collection of application usage data and user feedback to inform interactive software development," Ph.D. dissertation, University of California, Irvine, 1999.

[109] D. M. Hilbert and D. F. Redmiles, "Large-scale collection of usage data to inform design," in *Conference on Human-Computer Interaction (INTERACT)*. Ios Press Inc., 2001, pp. 569–576.

[110] M. El-Ramly and E. Stroulia, "Mining software usage data," in *International Workshop on Mining Software Repositories (MSR)*. IEEE, 2004, pp. 64–8.

[111] J. Matejka, W. Li, T. Grossman, and G. Fitzmaurice, "Communitycommands: Command recommendations for software applications," in *Symposium on User Interface Software and Technology (UIST)*. ACM, 2009, pp. 193–202.

[112] F. Linton, D. Joy, and H.-P. Schaefer, "Building user and expert models by long-term observation of application usage," in *International Conference on User Modeling (UMAP)*. Springer, 1999, pp. 129–138.

[113] F. Linton and H.-P. Schaefer, "Recommender systems for learning: Building user and expert models through long-term observation of application use," *User Modeling and User-Adapted Interaction*, vol. 10, no. 2–3, pp. 181–208, 2000.

[114] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse, "The lumière project: Bayesian user modeling for inferring the goals and needs of software users," in *Conference on Uncertainty in Artificial Intelligence (UAI)*. Morgan Kaufmann Publishers Inc., 1998, pp. 256–265.

[115] G. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the elipse ide?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[116] S. Elbaum and M. Hardojo, "An empirical study of profiling strategies for released software and their impact on testing activities," *SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 65–75, 2004.

[117] S. Elbaum and M. Diep, "Profiling deployed software: Assessing strategies and testing opportunities," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 312–327, 2005.

[118] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: Less is more," *SIGPLAN Notices*, vol. 35, no. 11, pp. 202–211, 2000.

[119] T. W. Reps, T. Ball, M. Das, and J. R. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *European Software Engineering Conference (ESEC/FSE)*. Springer, 1997, pp. 432–449.

[120] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. Vokolos, “Scenariographer: a tool for reverse engineering class usage scenarios from method invocation sequences,” in *International Conference on Software Maintenance (ICSM)*. IEEE, 2005, pp. 155–164.

[121] S. Narayanasamy, C. Pereira, and B. Calder, “Software profiling for deterministic replay debugging of user code,” *Frontiers in Artificial Intelligence and Applications*, vol. 147, no. 0, p. 211, 2006.

[122] L. Fagui, L. Shengwen, X. Ran, and L. Chunwei, “A low-overhead method of embedded software profiling,” in *International Colloquium on Computing, Communication, Control, and Management (CCCM)*, vol. 4. IEEE, 2009, pp. 436–439.

[123] Y. C. Yang and B. Padmanabhan, “Toward user patterns for online security: Observation time and online user identification,” *Decision Support Systems*, vol. 48, no. 4, pp. 548–558, 2010.

[124] J. Froehlich, M. Y. Chen, S. Consolvo, B. Harrison, and J. A. Landay, “Myexperience: A system for in situ tracing and capturing of user feedback on mobile phones,” in *International Conference on Mobile Systems, Applications and Services (MobiSys)*. ACM, 2007, pp. 57–70.

[125] R. Cooley, B. Mobasher, and J. Srivastava, “Web mining: information and pattern discovery on the world wide web,” in *International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 1997, pp. 558–567.

[126] O. Baysal, R. Holmes, and M. Godfrey, “Mining usage data and development artifacts,” in *Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 98–107.

[127] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher, “Leveraging user-session data to support web application testing,” *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 187–202, 2005.

[128] S. Alam, G. Dobbie, and P. Riddle, “Particle swarm optimization based clustering of web usage data,” in *International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. IEEE, 2008, pp. 451–454.

[129] M. A. Bayir, I. H. Toroslu, A. Cosar, and G. Fidan, “Smart miner: A new framework for mining large scale web usage data,” in *International Conference on World Wide Web (WWW)*. ACM, 2009, pp. 161–170.

[130] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, “Web usage mining: Discovery and applications of usage patterns from web data,” *SIGKDD Explorations Newsletter*, vol. 1, no. 2, pp. 12–23, 2000.

[131] K. A. Coombs, “Lessons learned from analyzing library database usage data,” *Library Hi Tech*, vol. 23, no. 4, pp. 598–609, 2005.

[132] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng, “Understanding user behavior in large-scale video-on-demand systems,” *SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 333–344, 2006.

[133] T. Fagni, R. Perego, F. Silvestri, and S. Orlando, “Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data,” *ACM Transactions on Information Systems*, vol. 24, no. 1, pp. 51–78, 2006.

[134] T. Abdelzaher, “An automated profiling subsystem for QoS-aware services,” in *Real-Time Technology and Applications Symposium (RTAS)*. Springer, 2000, pp. 208–217.

[135] S. Devaraj and R. Kohli, “Performance impacts of information technology: Is actual usage the missing link?” *Management Science*, vol. 49, no. 3, pp. 273–289, 2003.

[136] J. Yang, Y. Qiao, X. Zhang, H. He, F. Liu, and G. Cheng, "Characterizing user behavior in mobile internet," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 1, pp. 95–106, 2015.

[137] A. Sinha and A. Chandrakasan, "Jouletrack-a web based tool for software energy profiling," in *Design Automation Conference (DAC)*. IEEE, 2001, pp. 220–225.

[138] J. Flinn and M. Satyanarayanan, "Powerscope: a tool for profiling the energy usage of mobile applications," in *Workshop on Mobile Computing Systems and Applications (HotMobile)*. IEEE, 1999, pp. 2–10.

[139] T. Roehm, B. Bruegge, T.-M. Hesse, and B. Paech, "Towards identification of software improvements and specification updates by comparing monitored and specified end-user behavior," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 464–467.

[140] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej, "Monitoring user interactions for supporting failure reproduction," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 73–82.

[141] T. Roehm and B. Bruegge, "Reproducing software failures by exploiting the action history of undo features," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 496–499.

[142] T. Roehm, S. Nosovic, and B. Bruegge, "Automated extraction of failure reproduction steps from user interaction traces," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2015, pp. 121–130.

[143] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, 2003.

[144] J. Roche, "Adopting devops practices in quality assurance," *Communications of the ACM*, vol. 56, no. 11, pp. 38–43, 2013.

[145] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 435–445.

[146] A. Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2000.

[147] J. Mund, D. M. Fernandez, H. Femmer, and J. Eckhardt, "Does quality of requirements specifications matter? combined results of two empirical studies," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.

[148] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[149] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2007.

[150] Y. Malaiya, M. Li, J. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 420–426, 2002.

[151] B. Hauptmann, M. Junker, S. Eder, E. Juergens, and R. Vaas, "Can clone detection support test comprehension?" in *International Conference on Program Comprehension (ICPC)*. IEEE, 2012, pp. 209–218.

[152] B. Hauptmann, M. Junker, S. Eder, L. Heinemann, R. Vaas, and P. Braun, "Hunting for smells in natural language tests," in *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1217–1220.

[153] J. Bible, G. Rothermel, and D. S. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 149–183, 2001.

[154] V. Channakeshava, V. K. Shanbhag, A. Panigrahi, R. Sisodia, and S. Lakshmanan, "Safe subset-regression test selection for managed code," in *India Software Engineering Conference (ISEC)*. ACM, 2008, pp. 137–138.

[155] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo, "Testtube: A system for selective regression testing," in *International Conference on Software Engineering (ICSE)*. IEEE, 1994, pp. 211–220.

[156] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for java software," *SIGPLAN Notices*, vol. 36, no. 11, pp. 312–326, 2001.

[157] D. Willmor and S. Embury, "A safe regression test selection technique for database-driven applications," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2005, pp. 421–430.

[158] G. Rothermel and M. J. Harrold, "Selecting regression tests for object-oriented software," in *International Conference on Software Maintenance (ICSM)*. IEEE, 1994, pp. 14–25.

[159] ——, "Selecting tests and identifying test coverage requirements for modified software," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 1994, pp. 169–184.

[160] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 4, pp. 1–49, 2007.

[161] A. Kontostathis, "Essential dimensions of latent semantic indexing (LSI)," in *Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2007, pp. 73–73.

[162] Z. Xu and G. Rothermel, "Directed test suite augmentation," in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2009, pp. 406–413.

[163] R. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso, and M. Harrold, "Test-suite augmentation for evolving software," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2008, pp. 218–227.

[164] Z. Xu, Y. Kim, M. Kim, and G. Rothermel, "A hybrid directed test suite augmentation technique," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2011, pp. 150–159.

[165] J. Wettinger, U. Breitenbücher, and F. Leymann, "Standards-based devops automation and integration using tosca," in *International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2014, pp. 59–68.

[166] J. Humble and J. Molesky, "Why enterprises must adopt devops to enable continuous delivery," *Cutter IT Journal*, vol. 24, no. 8, pp. 6–12, 2011.

[167] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*. Addison-Wesley, 2003.

APPENDIX A

Publication A [1]

Venue: ICSE 2012
Acceptance rate: 16%
Length: 10 pages
Type: Full paper
Reviewed: Peer reviewed

How Much Does Unused Code Matter for Maintenance?

Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann
Institut für Informatik
Technische Universität München, Germany
Garching b. München, Germany
{eders,junkerm,juergens,hauptmab}@in.tum.de

Rudolf Vaas, Karl-Heinz Prommer
Munich Re
München, Germany
{rvaas,hprommer}@munichre.com

***Abstract*—Software systems contain unnecessary code. Its maintenance causes unnecessary costs. We present tool-support that employs dynamic analysis of deployed software to detect *unused* code as an approximation of *unnecessary* code, and static analysis to reveal its changes during maintenance. We present a case study on maintenance of unused code in an industrial software system over the course of two years. It quantifies the amount of code that is unused, the amount of maintenance activity that went into it and makes the potential benefit of tool support explicit, which informs maintainers that are about to modify unused code.**

***Keywords*-Software maintenance, dynamic analysis, unnecessary code, unused code**

I. Introduction

Many software systems contain unnecessary functionality. In [1], Johnson reports that 45% of the features in the analyzed systems were never used. Our own study on the usage of an industrial business information system [2] showed that 28% of its features were never used.

For consumer software, speculative or even unnecessary features might be justified to lure new customers into buying a product. For custom developed software, such as the business information systems developed and maintained at Munich Re, a re-insurance company in Germany, however, they provide no value at all.

For such systems, maintenance of unnecessary features is a waste of development effort. To avoid such waste, maintainers must know which code is still used and useful, and which is not. Unfortunately, such information is often not available to software maintainers. In our own study [2], expected and actual usage frequency deviated from 40% to 53% of the cases (depending on which stakeholder was involved in the evaluation). The picture was even clearer for entirely unused features: for over 70% of them, it surprised the stakeholders that they were not used at all.

Whether unnecessary code causes maintenance efforts depends on whether it actually needs to be adapted during maintenance. On the one hand, we expect *perfective* and *corrective* change requests [3] to mostly arise for features that are important to their users—otherwise they would not complain about bugs or demand changes. For unnecessary features, perfective and corrective maintenance effort can thus be expected to be low. On the other hand, however, *preventive* and *adaptive* maintenance [3] regularly affect code independent of the functionality it implements. Examples include the migration of a software system to a new programming language or platform, or the replacement of a component, such as the underlying database. Such changes also affect unused code. Both adaptive and preventive maintenance are regularly performed during software evolution. Extensive studies on maintenance effort reported that perfective and corrective maintenance constitute merely 48% [4], 64% [5], and 75% [6] of all change requests. The remaining changes comprise adaptive and preventive maintenance. We thus have to expect unnecessary code to be subject to maintenance, too. To perform maintenance tasks cost-effectively, maintainers must know which code is still necessary, and which is not.

Whether code is still necessary or not is determined by the function it fulfills for its users. The value of code is thus not an inherent property, but determined by its context. One way to approximate usefulness of code to its users is to monitor its usage in production, and compare it to its expected usage. If code is never used, and does not implement infrequently used functionality such as failure recovery, maintainers should investigate if the effort for its modification is justified.

However, the recording of usage information and consideration of unused code is not an integral part of software maintenance practice. Based on our experience and that of our industrial partners, we see two reasons for this. First, we have little empirical data on how much unused code exists in software and how strongly it affects maintenance. Second, we lack suitable tool support to capture usage data in production and present it in a way suitable to maintainers. As a consequence, it remains unclear how important unused code for cost-effective maintenance really is in practice. Given the amount of unused code in industrial software, we consider this precarious both for practice and for education.

Problem: Real-world software contains unnecessary code. Its maintenance is a waste of development resources. Unfortunately, we lack tool support to identify unnecessary code and empirical data on the magnitude of its impact on

maintenance effort. As a consequence, it is unclear how harmful unnecessary code is for software maintenance.

Contribution: In this paper, we present tool support to collect code-level usage information in a production environment, to approximate unnecessary code. We contribute a case study that analyzes the usage of an industrial business information system over the period of over 2 years. The study quantifies maintenance effort in unused code and shows the potential benefits of the tool support we propose.

II. Outline

The paper is structured as follows: In the next section, we define important terms. Afterwards, we introduce our tool support in detail, followed by a description of the case study. We then discuss our results and related work. We conclude with an overview of future work and a summary of our results.

III. Terms

Method: Units of functionality of a software system. Methods consist of a signature and a body.

Method genealogy: List of methods that represent the evolution of a single method over different versions of a software system. The list contains all versions of one method in chronological order.

Modified: A method genealogy is modified if not all its methods are equal with respect to their signatures and bodies. A method is modified if it is part of a genealogy which is modified.

Unused: A method genealogy is *unused*, if none of its methods is executed in a productive environment. This is not necessarily useless or dead code, but code that was just not executed in a considered time frame.

Unnecessary: A method is *unnecessary*, if it is not needed to fulfill the system's intended purpose and could be removed. Domain and development knowledge is necessary to decide whether a method is unnecessary.

IV. Tool Support

The proposed tool support is divided into four steps: the collection of usage data, the analysis of the program structure, the combination of both in a data repository, and the generation of statistics that can be used by developers. The tool chain is illustrated in Figure 1.

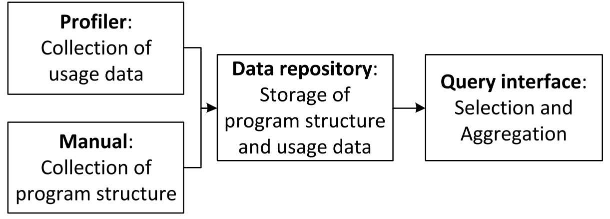


Figure 1. Schematic illustration of the proposed tool chain.

For collecting usage data, we use a profiler based on the .NET profiling API that logs method invocations.

For each program version that gets deployed, we collect assemblies (compilation units in .NET) manually to obtain the structure and functionality of the program. The reason why we work on the binary level and not directly on the source code level is that the system includes several components which are developed by different teams, have different release cycles and are only integrated in binary form. It is thus non-trivial to determine the complete source code for a program version that ran in the productive environment.

Usage data, as well as the software system's structure and functionality are stored in a central data repository. For calculating statistics, we provide a query interface. In the following sections, we explain the different steps in more detail.

A. Profiling Usage Data

When collecting usage data, we need to minimize the impact on the productive system while still providing enough accuracy to gain valuable information. Therefore, we use an ephemeral [7] profiler that records which methods were called within a certain time interval. The profiler does not record how often a method was called, just if it was called in a given time interval. For our study, we set the time interval to one day, to gain data that is accurate enough but produces very low performance impact. More information on the profiler can be found in [2].

Every method is instrumented with a profiling hook at (re-)start of the software system. This hook is removed after the first call of the method and therefore yields no performance impact on later invocations. This technique may miss methods that were inlined by the just-in-time compiler for performance issues, and, thus, we also instrument methods for the inlining event and count this event as an invocation. This is valid, because inlining is performed just in time by the virtual machine. The resulting data is written to a file at every shutdown of the system. Our approach is based on .NET, but not limited to it. It can also be applied to other environments with a virtual machine, such as Java.

B. Data Repository

To store usage data and the structure of every version of the examined software system, we use a database.

For every program version, the structure of the program is stored hierarchically. Program versions are decomposed into assemblies. Every assembly contains types (e.g., classes), which are themselves decomposed into methods. Types and assemblies only carry their names, whereas methods carry their signature and body. Figure 2 illustrates our data model.

After storing the program structure and usage data for all of the program versions, we map methods from one version to the next. Typically, one program version is succeeded by another version including bug fixes and change requests.

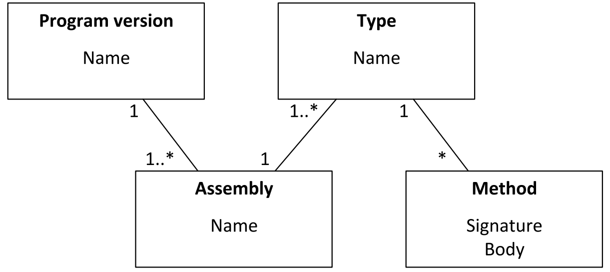


Figure 2. Data model of the program structure.

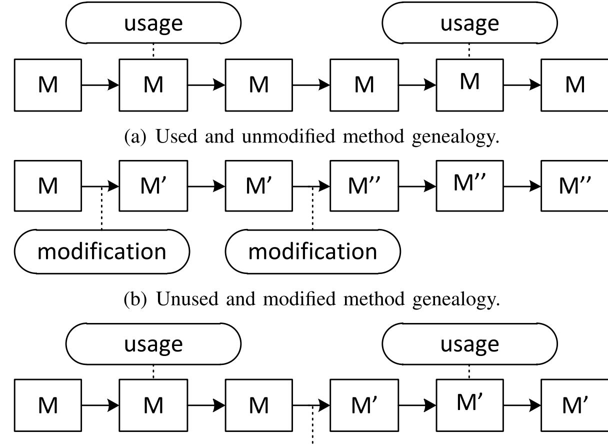
These changes are reflected in differences in method bodies and signatures, as well as in the structure of types and assemblies and their names. Because of these changes and in order to gain accurate usage data, methods have to be mapped from one program version to another. We perform this mapping by comparing the methods of succeeding program versions with respect to their signatures, bodies, enclosing types, and assemblies. We find the most accurate match for each method by first looking in the original type in the original assembly of the next program version. Types and assemblies are matched based on their names and contained methods. We then rate the similarity of methods, whereas the maximum similarity is given, if a method in the next program version is found in the same type and assembly with an exact match in the method name and parameters. The confidence in the similarity is hampered, if parameters or the enclosing type of a method have changed. The confidence is even lower, if only the parameter types of two methods are the same. We map methods to the most accurate match in the next version. This enables us to build lists of methods of different program versions that evolved from each other. We manage to map about 98% of all methods from one program version to the next. The list follows the ordering of the software system's versions. We call these lists *method genealogies*, as defined in Section III.

Maintenance between two versions is detected by comparing two consecutive methods in the same genealogy. There are three possible actions, a developer could have performed: Add, remove, or change methods.

All of the three actions can be detected in genealogies. If a method was added during the program evolution, its genealogy does not reach to the first program version. If a method was removed, its genealogy does not reach to the last program version and if a method was changed, the body or signature changes in the method genealogy.

Figure 3 depicts the most important kinds of method genealogies. The first genealogy is used twice, but never modified. The second genealogy is never used, but modified twice. The third genealogy is used twice and modified once.

Figure 4 illustrates the sets of method genealogies and their relationships. The method genealogies, that are interesting for our analysis, are *unused* and *modified*. These methods then can be split up into two sets again: Methods that were modified necessarily and unnecessarily.



(a) Used and unmodified method genealogy.

(b) Unused and modified method genealogy.

(c) Used and modified method genealogy.

Figure 3. Different types of method genealogies.

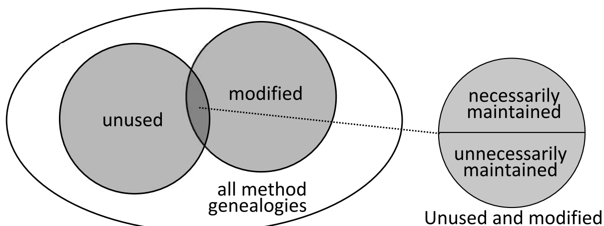


Figure 4. Analyzed sets of method genealogies and their composition.

Having the time interval a program version was productive, we can reconstruct the possible time interval in which a method was changed: Between the start of its own program version and the beginning of the next. We used this information in order to retrieve the source code that was affected by a modification and to discuss it with the developers. If a method was modified, each of its maintenance actions is recorded as single event.

C. Query Interface for Developers

To support developers in maintenance, we provide a query interface for the analysis results. This interface allows generating statistics about the percentage of unused method genealogies, the number of maintenance actions that have been performed on unused method genealogies, and the development of both over time. Thus, we use the query interface for obtaining the relevant number for the case study.

Furthermore, developers can search for methods they are maintaining and retrieve the usage frequency of these

methods. With the help of this information, developers can direct their maintenance effort.

V. Case Study

In this section, we explain the case study we conducted to quantify the impact of unused code on maintenance.

A. Research Questions

We formulate our research objective using the Goal-Question-Metric approach from [8]. The research objective is defined using the goal definition template as proposed in [9]:

> We analyze *usage and maintenance of a large industrial software system* for the purpose of *exploring the role of unused code* with respect to its *effect on maintenance* from the viewpoint of *maintenance engineers and developers* in the context of *industrially hosted business information systems*.

We infer the following research questions:

RQ1: How much code is unused in industrial systems? This question targets the existence and extent of unused code in industrial systems. If there is no unused code, our study would be irrelevant.

RQ2: How much maintenance is done in unused code? Having identified unused code, we answer the question about the existence and extent of maintenance effort that is spent on unused code.

RQ3: How much maintenance in unused code is unnecessary?
This research questions targets the existence and extent of maintenance that gets spent on unnecessary code. This question determines the severity of the problems caused by maintenance actions in unused code and the potential of savings of maintenance effort.

RQ4: Do maintainers perceive knowledge of unused code useful for maintenance tasks?
This question determines the usefulness of the proposed analysis. It is especially interesting whether the analysis helps developers to direct their maintenance effort.

B. Study Object and Subjects

We evaluated the research questions with respect to a business information system being in production at Munich Re Group. Munich Re Group is one of the largest reinsurance companies in the world and employs more than 47,000 people in over 50 locations. For their insurance business, they develop a variety of custom supporting software systems. The analyzed business information system implements damage prediction functionality and supports about 150 expert users in over 10 countries. An overview is shown in Table I.

Table I
Study object.

Language	C#
Age (years)	8
Size at beginning (kLOC)	360
Engineers (max)	9 (16)
Min. # Methods (size at beginning)	13908
Max. # Methods (size at end)	21664
# Versions	19

We chose this system as study object for several reasons. First, the system has been in successful use for 8 years and is still actively used and maintained. Understanding the impact of unnecessary code on maintenance is thus likely to decrease maintenance costs. Second, the development and usage context is typical for the Munich Re Group. Its users are distributed across different countries. The software engineers are from different companies (some are employed by Munich Re, some by software suppliers) and work at different sites. This distribution of users and engineers complicates communication inside and across the stakeholder groups and could thus lead to a lack of usage information. Third, it is a web application. Its server offers a single point for usage data collection. Our study subjects are two maintainers of the system. Both have been working in the system for 8 years actively. Thus, they have deep knowledge about the system.

C. Study Design

We conduct our study in two major steps. First, we collect program and usage data. Second, we analyze the data in four steps oriented at our research questions.

RQ1: Amount of unused code: We answer RQ1 using the profiling data. We calculate the fraction of the number of unused method genealogies off the overall number of method genealogies for all individual program versions as well as in total.

RQ2: Maintenance in unused code: For RQ2, we need to identify modifications in method genealogies. Modifications in a genealogy occur between two program versions. Therefore, we compare successive methods and determine if they differ. This way, we find all modifications for a genealogy. If a modification took place in an unused genealogy, we conclude that the maintenance effort for this modification was spent on unused code.

RQ3: Amount of unnecessary maintenance and RQ4: How does the analysis help the developers In order to answer RQ3 and RQ4, we discuss our findings with the developers of the system. There are typically large parts of a system's code that are systematically unused in production such as unit tests or code related to batch jobs that are not executed in the productive environment. In order to get meaningful results, it is important to exclude such code from the analysis. Therefore, in a first round, we select unused, but

maintained methods in a way that every part of the system that exhibited unused code is represented in the sample. Additionally, we select methods that seem to be noticeable (e.g., methods with a large change in size or methods whose names suggest unit tests). This results in a set of 24 methods. We present this sample to a developer and use the results to improve the filters in our analysis. These filters are also applied for the measurements to answer RQ1 and RQ2. In a second round, we take a random sample of cases, which we discuss in detail with a different developer of the system in order to elicit the reasons why the code was not used and to quantify the fraction of unnecessary maintenance. We are able to discuss 27 cases with this developer. Furthermore, we investigate how the developer would have acted with knowledge about the unused code and discuss if a tooling as proposed would be helpful for supporting maintenance tasks.

D. Execution

During the analysis period of two years, we gathered usage data of 19 different program versions. Figure 5 shows the distribution of the program versions over time. The uncovered time intervals exist due to missing data that was lost because of technical errors.

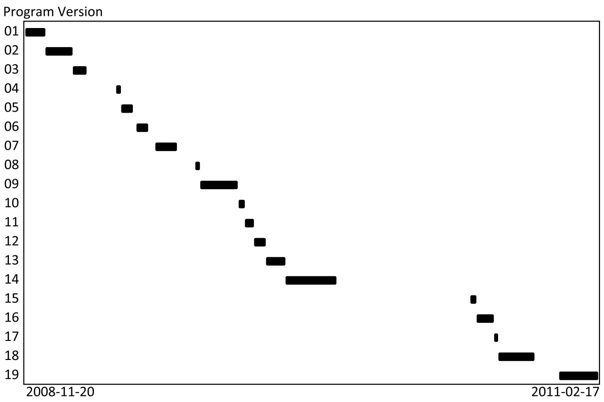


Figure 5. All program versions with their time span of deployment.

In the following, we discuss the concrete procedure for answering our research questions.

RQ1: The investigation of RQ1 requires some prerequisites. The program structure, consisting of assemblies, types, and methods, is extracted from the binary versions of the software system by using the .NET reflection technique. We then store the whole structure of the program in a relational database to perform the method mapping. We map methods based on several heuristics that consider the signature and location of a method, the name of its containing type and assembly.

This results in a multistage mapping procedure to find methods in the succeeding program version. At first, the algorithm maps assemblies and types in the next program version based on their names and contents. Based on the found relations between assemblies and types, methods are mapped. Methods are preferably matched if they have the same location and the same signature. If no completely matching method is found at the exact location, methods are compared based on the method's signature (name, return type and argument types). This way, we can match methods with arguments added and removed or with a changed name. If no match was found at this stage, the whole software system is searched for the method based on the same heuristics as before. This way, we find methods that were moved.

If iterated over all pairs of consecutive program versions, this procedure leads to method genealogies that reach at most from the first program version to the last.

RQ2: Maintenance actions can be derived by comparing method signatures (name, return type, and argument types) and bodies of consecutive methods. For method bodies, we check intermediate language code of methods for equality. If signatures or bodies differ within one genealogy, we count this as a modification.

RQ3: To obtain information about how much code is unnecessary, we present a random sample of method genealogies from the unused, but modified, method genealogies to maintainers. With the help of the maintainers, we split our sample into two groups: A set of maintained and unused, but necessary, method genealogies and a set of maintained, but unnecessary, methods.

RQ4: The interviews we conducted to answer RQ3 also provided information for RQ4. In the interviews, we asked the developers, how useful and interesting the provided information was. Furthermore, having identified the unnecessarily maintained method genealogies, we quantify the accuracy of our analysis by comparing the set of unused, but maintained, method genealogies with the set of unnecessarily maintained method genealogies.

Technical details: We conducted the study using a machine with two 2.4 GHz processor cores and dedicated 4 GB of RAM. We were using a relational database for storing the program structure and usage data. The complete evaluation toolkit is written in Java. Inserting the program structure and usage data of all 19 program versions into the database took about 7 hours. Mapping methods and generating the results took about 5 minutes.

Table II
DISTRIBUTION OF MODIFIED AND UNMODIFIED METHOD GENEALOGIES, DEPENDING ON USAGE.

	Used	Unused	Total
Unmodified	53.3%	22.9%	76.2%
Modified	21.7%	2.1%	23.8%
Total	75%	25%	100%

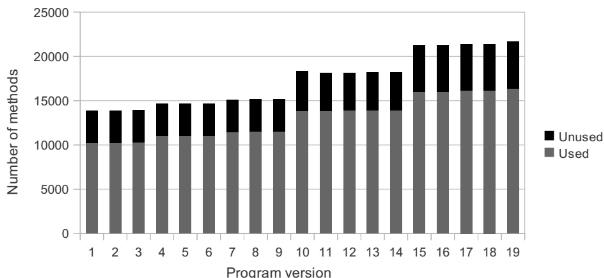


Figure 6. Number of used and unused methods per program version.

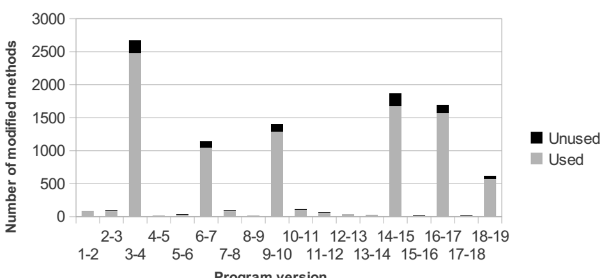


Figure 7. Number of methods that were modified from one program version to the next. The grey bar shows the part of the methods that are used; the black bar shows the part of methods that are unused.

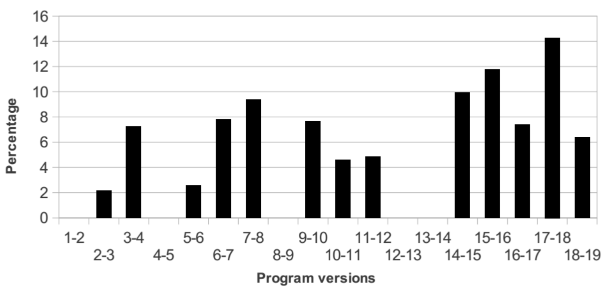


Figure 8. The fraction of the modified and unused methods of all modified methods.

E. Results

In the following, we present the results of our study by providing numbers for our research questions. The numbers are based on the method genealogies we obtained after filtering. That means unit tests and code related to batch jobs that are not executed in production are not included.

In the system, we identified 25,390 method genealogies. Of these, 6,028 were modified with a total of 9,987 individual modifications. This means that considerable maintenance effort took place during the analysis period.

RQ1: Amount of unused code: Table II shows the distribution of used and unused method genealogies. We found that 25% of all method genealogies were never used during the complete period. The fraction of unused methods is roughly stable across program versions, as illustrated by Figure 6.

RQ2: Maintenance in unused code: We first compared the degree of maintenance (i.e. percentage of maintained genealogies) between used and unused method genealogies. We found that 40.7% of the used method genealogies were maintained, but only 8.3% of the unused method genealogies. That means, unused methods were maintained less intensively than used methods. The unused genealogies account for 7.6% of the total number of modifications. Figure 7 shows how the modifications are distributed over the program versions in absolute numbers and Figure 8 shows the percentage of the number of unused methods off the number of modified methods for each program version.

RQ3: Amount of unnecessary maintenance: We reviewed the examples of unused maintenance with the developers. By inspecting the affected code and researching the reason why it is not used, we found that in 9 of 27 (33%) cases, the unused code was indeed unnecessary. In another 4 cases (15%) the code in question was no longer existent as it was either deleted or moved. That means that in nearly every second case unused methods were either unnecessary or potentially deleted from the system. The exact actions could not be retrieved from the versioning system.

RQ4: How does the analysis help the developers: In both discussion rounds, we encountered great interest in the analysis results, especially in the cases in which unused methods were maintained. Often, the developers were surprised that the respective method was not used. When investigating the reason why a particular method was not used, in some cases the developers discovered a bug (e.g., a business function was not correctly hooked into the UI). Regarding the tool support we got the feedback that information about unused methods would be helpful to support maintenance.

Experiences with the mapping of methods: In general, we found that our procedure to identify genealogies works well in the majority of cases. The fraction of methods that could be mapped from one program version to the next usually was around 98%. Only in two cases it was significantly lower (90% and 93%). As it is sensible to assume that a certain number of methods are deleted without replacement due to refactoring or changed requirements, the fraction of methods genealogies that are erroneously terminated should be very small. To support this claim we manually inspected 20 genealogies. As far as we could tell all of them were correct.

F. Interpretation

This section presents an interpretation of the results based on the research questions.

RQ1: Amount of unused code: In our case study, 25% of the implemented code has never been executed and, therefore, can be considered as unused. However, this

amount does not only consist of code of unused features, but also of error handling routines such as exception handlers.

RQ2: Maintenance in unused code: Bringing together the code usage with the locality of the changes, it is noticeable that most of the modifications have been done in code which has been executed. Only 7.6% of the system's changes affect methods that have never been executed. This means that most of the maintenance (92.4%) has been spend on actually used code. For this, we have the following explanation. Change requests primary address the key functionality of a system. Functionality, which is rarely or never executed, is less likely affected by change requests. Furthermore, it is more likely to find bugs in executed code as in code which is not executed at all. Additionally, developers are aware that some code is seldom or never executed and focus their maintenance effort on actually used code.

RQ3: Amount of unnecessary maintenance: Based on our measurements, 7.6% of all maintenance modifications have been performed in unused code regions. To decide whether modifications are unnecessary or not, we can combine the answers from the developer interviews. Therefore, 48% of all modifications performed in unused code can be considered unnecessary. The cleanup of all unused code would cause unreasonable effort. Thus, we suggest tool support to warn developers in case they are performing maintenance tasks in an unused code region. With this, the developer can decide individually if the planned maintenance modifications are necessary or not. During our interviews with the developers, we found that the actions taken to maintain unused and unnecessary code are very similar to the actions that are taken to maintain used code (from refactoring to more complex adaptions). This implies that the actual effort spent on unnecessary code is comparable to the aforementioned numbers.

RQ4: How does the analysis help the developers: Connecting the amount of maintenance in unused code (7.6%) with the ratio of unnecessary maintenance from the interviews (48%), 3.6% of the overall maintenance is needless. However, using usage data during maintenance, a developer can be sure in 92.4% that the performed changes are definitely necessary, whereas in the remaining 7.6% which are not used there is a 48% chance to avoid unnecessary changes. Furthermore, during the developer interviews, several bugs have been detected which are directly related with unused code.

G. Threats to Validity

In this section, we discuss the threats to validity in our study. We structure the threats by internal, external and construct validity.

Internal validity: We consider genealogies as used, if they were used at an arbitrary time during the examination period. If a method was modified after its last usage, we are still considering the method genealogy as used and modified. Therefore, we are missing method genealogies that are unnecessarily maintained after their last usage, because they are not used in the future. This results in an underapproximation of the amount of maintenance actions in unused code and implicates more conservative estimations, which we see, however, as a minor threat.

There are two missing time intervals in our study due to technical errors as shown in Figure 5. Because of that, we compared program versions that did not follow each other directly. However, we are able to map nearly as many methods between these program versions as between all other versions. Thus, we consider this to be a minor threat. The missing data also affects usage data. This results in method usages, which are not considered in our analysis. The discussion with the maintainers did not show any methods that were used in the maintainers' opinion. Thus, we consider this as a minor threat.

External validity: We examined only one system with our analysis. To gain more general numbers about how much unnecessary code is maintained during the software life cycle, the investigation of our research questions on more systems is needed. This allows for a generalization of our findings, whereby we currently plan similar studies on other systems.

Construct validity: Another threat to validity is that method mapping may produce false method genealogies. This effect can be divided into two classes: Methods are set into relation that should not be and methods are not set into relation that should be. This can cause under- and overestimation of the maintenance actions on unused code.

These imprecisions arise due to the lack of information about the exact history of methods, types, and namespaces and the resulting estimation of relationships of different methods. We minimize these effects by implementing a rather conservative search algorithm, which matches method based on several heuristics, considering all possible successors and matching methods with the highest probability. In addition, we manually analyze further random samples of method genealogies. Since we did not find any errors, this strengthens our confidence in a low overall amount of errors.

VI. Discussion

Our analyses, as well as the interviews with the maintainers show results that exceed our research questions. In this section, we discuss these results.

The analysis shows that 3.6% of the maintenance was unnecessary. We expect this low number to be caused by the structure of the development team. Most of the developers are maintaining and developing the system since the beginning and are experts for the domain, as well as for the system. In an environment where developers change more frequently, it is likely that there is less knowledge about the actual usage of the program and, thus, more maintenance in unnecessary code.

According to the developers, the main causes for maintenance of unused code are:

- Exception handling
- Interfaces that had to be implemented
- Code for future use
- Code for testing

We also detected code that was about to be removed or was removed shortly after our examination interval. This means that our analysis was able to identify unnecessary code the developers were aware of.

However, the maintaining developers found the information our analysis provided very useful. In 48% of our findings for maintenance of unused code, the developers did not know why the method was not used. Knowledge about the usage frequency would significantly have changed the behavior of the developers regarding maintenance. In the sample set of methods, the most interesting findings pointed to bugs. For example, we found a method that checked certain conditions that validated a data set. With this check not being performed, it was possible to insert inconsistent data into the system's database. According to the maintainers, the unnecessarily maintained methods are undergoing deeper investigation. These methods are either used in the future or subject to removal.

With our analysis, we narrowed the set of methods to look for unnecessary maintenance from 6369 (all unused method genealogies) to 529 methods (maintained method genealogies that were not used). This makes it a lot easier for maintainers to identify misdirected maintenance effort. This effect was also confirmed by the maintainers. These results and their interpretation as well as the feedback of the developers, points out that this analysis is useful for maintainers in practice.

VII. Related Work

To the best of our knowledge, other approaches that use usage information to find out unused code to support maintenance do not exist. However, our approach builds on existing work from several areas. We relate it to *remote analysis of deployed software*, *program profiling*, *code coverage testing*, *diff and semantic diff*, as well as *unnecessary code elimination*.

Remote analysis of deployed software: has been proposed by several researchers. Hilbert [10] proposes to employ agents to collect usage information in deployed software to support usability engineering. Orso et al. [11] investigate means to distribute monitoring tasks across users to reduce associated impact. Liblit et al. [12], [13] propose remote program sampling to isolate bugs. Elbaum and Diep [14] survey existing approaches to support testing by profiling deployed software. Haran et al. [15] present approaches to classify execution data gathered during remote program analysis in support of further analysis. These approaches were a valuable inspiration for our work and provide general indication for the feasibility of profiling deployed software. However, to the best of our knowledge, none of them are targeted at usage analysis and maintenance.

Program profiling: [16] is an established practice in performance engineering to identify problematic code. Existing approaches can be categorized into exact and statistical profilers. While exact profilers yield precise results, their potentially devastating impact on performance inhibits their application on production machines. Statistic approaches sacrifice precision to reduce performance impact and, thus, can be applied to continuous profiling of deployed software [17]. Ephemeral profiling [7], as we employ it, combines exact results with minimal impact, thus, combines the advantages of both approaches.

Code coverage testing: In software testing, metrics such as instruction, branch, path, or condition coverage are used to measure the quality of test suites. These metrics describe to which degree a program has been tested based on its control flow graph. Many testing tools track the control flow during test execution by either injecting additional measurement code or using a system's debugging interface. Both affect the execution time and memory consumption in a negative way. Since our test object was under productive use, none of these techniques could be applied. We focused on tracking just method executions in a lightweight way, which affects the system's run time behavior in a minimal way. We injected measurement code, which is executed just once for every first execution of a method. Since the application has been restarted every day, method usage could be tracked on a day time precision.

Syntactic and semantic diff: Differences between programs can be determined on several levels. There are approaches comparing two versions of a program on the syntactic, as well as on the semantic level. The UNIX diff tool, for example, performs a lexical analysis of two text sources. Even little textual changes, which have no effect on the compilation, such as removing unnecessary line brakes or spaces, will be detected as changes. Some work has been done in finding differences based on abstract syntax trees (AST) of programs [18], [19], [20]. By comparing programs on their abstract structure, lexical changes, which do not affect the behavior, are ignored. Another approach is to compare programs on the semantic level. Semantic Diff [21], for example, creates local dependency graphs to compare their observable input-output behavior. LSdiff [22] uses logical structural deltas to detect and understand similarities in Java code. Since compiling source code already filters little changes of the source code, which does not change the system's semantics, we focused on comparing the binary representation of the system. However, during the compilation of .NET applications, some information, which is necessary to understand the intention of the developer, gets lost.

Unnecessary code elimination: Most compilers perform optimizations to remove unnecessary code. Code, which does not affect the applications result (dead code) or cannot be executed at all (unreachable code), is detected and not included in the compilation result [23], [24], [25]. In some development environments, this analysis is already performed during coding which helps the developer to remove this code. Since all this is performed before executing the system, the decision whether code is unnecessary is based on the static information available at compile time. In our approach, we perform a dynamic analysis using the actual usage data, which exists not until the execution of the system. To this end, we can detect code, which is technically reachable, but still never executed.

VIII. Future Work and Conclusion

In this section, we provide a conclusion, a short summary of our work, and an overview of the results of our case study. Afterwards, we outline our future research plans.

A. Conclusion

Real-world software systems typically contain unnecessary code. Maintenance of this code is unnecessary and produces unnecessary maintenance costs.

To understand the impact of unused code on maintenance, we monitored the usage and maintenance actions of an industrially hosted business information system for over two years. We quantified the amount of unused code and measured how often such code is maintained. Furthermore, we investigated to what extent maintenance tasks on unused code are unnecessary. We conducted our study by using the presented tool support.

From our analysis, we draw two main conclusions: A large portion of the code has not been used over the analysis period of two years (25% of all methods). However, a surprisingly low amount of maintenance (7.6% of all maintenance actions) is spent on this fraction of the software system. Therefore, unused code is not a severe problem in the maintenance of the examined system. But nearly 50% of the maintenance actions that were performed on the unused parts of the system affected methods that were unnecessary and caused unnecessary maintenance effort or even bugs. The information received during interviews with the maintainers of the examined software system indicates that our analysis is helpful for them.

We believe that our analysis would show a greater amount of unnecessary maintenance for projects with a different structure of the maintaining team. We are optimistic that this analysis helps directing maintenance efforts more effectively.

B. Future Work

Motivated by the results of our study, we plan a number of improvements and validations for the proposed analysis in the future.

Increase accuracy: At this stage, we are not able to map functionality of methods that is migrated into other methods. In order to gain more precise statistics, we plan to improve our mapping mechanisms. Moreover, we are working on more elaborated statistics and metrics that measure unnecessary maintenance effort.

Another possibility to increase the accuracy of our analysis is to employ procedures for filtering exception handlers and similar parts of the code from the set of unused and maintained methods. We are optimistic to raise the ratio of unnecessarily maintained code in our findings.

Test control: Our tooling can also be applied to test systems. Combined with knowledge about changes to an underlying system, we can point testers to methods that were changed, but not tested afterwards.

Representative study: In this study, we only observed one large software system. In the future, we will monitor more systems to gain more general results about the maintenance of unused code. Furthermore, the presented tool support only targets systems written in C#. As our approach is not limited to this programming language, we plan to adopt it also for other systems that work on virtual machines, for example, Java.

Acknowledgment

We are grateful to Markus Herrmannsdoerfer for his help with the execution of the study. We also thank Lars Heinemann, Markus Herrmannsdoerfer, and Daniel Méndez Fernández for their helpful comments.

References

[1] J. Johnson, “Roi, it’s your job,” Keynote at XP ’02.

[2] E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. Prommer, “Feature profiling for evolving systems,” in *ICPC ’11*, 2011.

[3] IEEE, “IEEE standard glossary of software engineering terminology,” Standard, 1990.

[4] D. Yeh and J.-H. Jeng, “An empirical study of the influence of departmentalization and organizational position on software maintenance,” *J. Softw. Maint. Evol. Res. Pr.*, 2002.

[5] H. D. Rombach, B. T. Ulery, and J. D. Valett, “Toward full life cycle control: Adding maintenance measurement to the SEL,” *J. Syst. Softw.*, 1992.

[6] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett, “Understanding and predicting the process of software maintenance release,” in *ICSE ’96*, 1996.

[7] O. Traub, S. Schechter, and M. D. Smith, “Ephemeral instrumentation for lightweight program profiling,” School of engineering and Applied Sciences, Harvard University, Tech. Rep., 2000.

[8] V. Basili, G. Caldiera, and H. Rombach, “The Goal Question Metric Approach,” *Encyclopedia of Software Engineering*, vol. 1, 1994.

[9] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: An introduction*. Kluwer Academic Publishers, 2000.

[10] D. M. Hilbert, "Large-scale collection of application usage data and user feedback to inform interactive software development," Ph.D. dissertation, University of California, Irvine, 1999.

[11] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: Continuous evolution of software after deployment," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, 2002.

[12] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *SIGPLAN Notices '03*, vol. 38, no. 5, 2003.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI '05*, 2005.

[14] S. Elbaum and M. Diep, "Profiling deployed software: Assessing strategies and testing opportunities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, 2005.

[15] M. Haran, A. Karr, M. Last, A. Orso, Alessandro d A. Porter, A. Sanil, and S. Fouche, "Techniques for classifying executions of deployed software to support software engineering tasks," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, 2007.

[16] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *SIGPLAN Notices '82*, 1982.

[17] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" *ACM Trans. Comput. Syst.*, vol. 15, no. 4, 1997.

[18] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *PLDI '90*, 1990.

[19] W. Yang, "Identifying syntactic differences between two programs," *Softw., Pract. Exper.*, vol. 21, no. 7, 1991.

[20] J. E. Grass, "Cdiff: A syntax directed differencer for C++ programs," in *UXENIX C++ '92*, 1992.

[21] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *ICSM '94*, 1994.

[22] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE '09*, 2009.

[23] R. U. J. D. Aho, Alfred V.; Sethi, *Compilers - Principles, Techniques and Tools*. Addison Wesley, 1986.

[24] A. Appel, *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[25] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, 2000.

APPENDIX B

Publication B [2]

Venue: AST@ICSE 2013
Acceptance rate: 45%
Length: 4 pages
Type: Short paper
Reviewed: Peer reviewed

Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice

Sebastian Eder, Benedikt Hauptmann, Maximilian Junker
Technische Universität München, Germany

Elmar Juergens
CQSE GmbH, Germany

Rudolf Vaas, Karl-Heinz Prommer
Munich Re Group, Germany

***Abstract*—Testing and development are increasingly performed by different organizations, often in different countries and time zones. Since their distance complicates communication, close alignment between development and testing becomes increasingly challenging. Unfortunately, poor alignment between the two threatens to decrease test effectiveness or increases costs.**

In this paper, we propose a conceptually simple approach to assess test alignment by uncovering methods that were changed but never executed during testing. The paper's contribution is a large industrial case study that analyzes development changes, test service activity and field faults of an industrial business information system over 14 months. It demonstrates that the approach is suitable to produce meaningful data and supports test alignment in practice.

***Index Terms*—Software testing, software maintenance, dynamic analysis, untested code**

I. Introduction

A substantial part of the total life cycle costs of long-lived software systems is spent on testing. In the domain of business-information systems, it is not uncommon that successful software systems are maintained for two or even three decades. For such systems, a substantial part of their total lifecycle costs is spent on testing to make sure that new functionality works as specified, and—equally important—that existing functionality has not been impaired.

During maintenance of these systems, test case selection is crucial. Ideally, each test cycle should validate all implemented functionality. In practice, however, available resources limit each test cycle to a subset of all available test cases. Since selection of test cases for a test cycle determines which bugs are found, this selection process is central for test effectiveness.

A common strategy is to select test cases based on the changes that were made since the last test cycle. The underlying assumption is that functionality that was added or changed recently is more likely to contain bugs than functionality that has passed several test cycles unchanged. Empirical studies support this assumption [1], [2], [3], [4].

If development and testing efforts are not aligned well, testing might focus on code areas that did not change, or—more critically—substantial code changes might remain untested. Test alignment depends on communication between testing and development. However, they are often performed by different teams, often located in different countries and time-zones. This distance complicates communication and thus challenges test alignment. But how can we assess test alignment and expose areas where it needs to be improved?

Problem: We lack approaches to determine alignment between development and testing in practice.

Proposed Solution: In this paper, we propose to assess test alignment by measuring the amount of code that was changed but not tested. We propose to use *method-level change coverage* information to support testers in assessing test alignment and improving test case selection.

Our intuition is that changed, but untested methods are more likely to contain bugs than either unchanged methods or tested ones. However, our intuition might be dead wrong: method-level churn could be a bad indicator for bugs, since methods can contain bugs although they have not changed in ages.

Contribution: This paper presents an industrial case study that explores the meaningfulness and helpfulness of method-level change coverage information. The case study was performed on a business information system owned by Munich Re. System development and testing were performed by different organizations in Germany and India. The case study analyzed all development changes, testing activity, and all field bugs, for a period of 14 months. It demonstrates that field bugs are substantially more likely to occur in methods that were changed but not tested.

II. Related Work

The proposed approach is related to the fields of defect prediction, selective regression testing, test case prioritization, and test coverage metrics. The most important difference to the named topics is the simplicity of the proposed approach and the fact that change coverage *assesses* the executed subsets of test suites, but does not give hints to improve them.

Defect prediction is related to our approach, because we identify code regions that were changed, but remained untested, with the expectation that there are more field bugs.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

There are several models for defect prediction [5]. In contrast to these models, we measure only changes in the system and the coverage by tests and do not predict bugs, but assess test suites and use the probability of bugs in changed, but untested code as validation of the approach.

The proposed approach is related to [6], which uses series of changes "change bursts" to predict bugs. The good results that were achieved by using change data for defect prediction encourage us to combine similar data with testing efforts.

Selective regression testing techniques target the selection of test cases from changes in source code and coverage information. [7], [8], [9].

In contrast to these approaches, the paper at hand focuses on the assessment of already executed test suites, because often experts decide which tests to execute to cover most of the changes made to a software system [10]. However, their estimations contain uncertainties and therefore possibly miss some changes. Our approach aims at identifying the resulting uncovered code regions. Therefore, our approach can only be used if testing activities were already performed.

Compared to [11], we are validating our approach by measuring field defects, and do not take defects into account that were found during development.

Test coverage metrics give an overview of what is covered by tests. Much research has been performed in these topics [12] and there is a plethora of tools [13] and a number of metrics available, such as statement, branch, or path coverage [14]. In contrast to these metrics, we focus on the more coarse grained method coverage. Furthermore, we do not only consider static properties of the system under test, but changes.

Empirical studies on related topics focus to the best of our knowledge mainly on the effectiveness of test case selection and prioritization techniques [9], [15]. In our study, we assess test suites by their ability to cover changes of a software system, but do not consider sub sets of test suites.

III. Context and Terms

In this work, we focus on *system testing* according to the definition of IEEE Std 610.12-1990 [16] to denote *"testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements"*. System tests are often used to detect bugs in existing functionality after the system has been changed. In our context, many tests are executed manually and denoted in natural language.

Our study uses *methods* as they are known from programming languages such as Java or C#. Methods form the entities of our study and can be regarded as units of functionality of a software system. They are defined by a signature and a body. To compare different releases of a software system over time, we create *method genealogies* which represent the evolution of a single method over time. A genealogy connects all releases of a method in chronological order [17].

In the context of our work, the life cycle of a software system consists of two alternating phases (see Figure 1). In the *development phase*, existing functionality is maintained or new features are developed. Development usually occurs in *iterations* which are followed by *test runs* which are the execution of a selection of tests aiming to test regressions as well as the changed or added code. A development phase is completed by a *release* which transfers the system into the *productive phase*. In the productive phase, functionality is usually neither added nor changed. If critical malfunctions are detected, *hot fixes* are deployed in the productive phase.

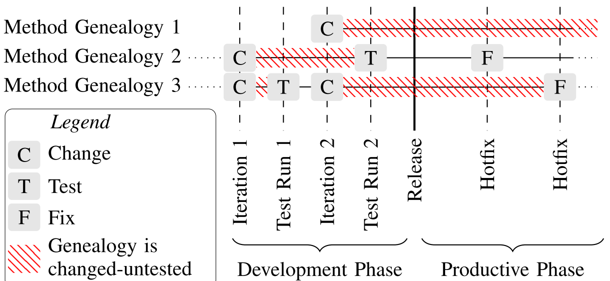


Fig. 1. Development life-cycle

We consider a method as *tested* if it has been executed during a test run. If a method has been changed or added and been tested afterwards before the system is released we consider it as *changed-tested*. If a method change or addition has not been tested before the system is transferred in the productive phase, we consider the method as *changed-untested* (see genealogy 1 and 3 in Figure 1).

IV. Change Coverage

To quantify the amount of changes covered by tests, we introduce the metric *change coverage (CC)*. It is computed by the following formula and ranges between [0,1].

$$\text{change coverage} = \frac{\text{\#methods changed-tested}}{\text{\#methods changed}}$$

A change coverage of 1 ($CC = 1$) means that all methods which have been changed since the last test run have been tested after their last change. On the contrary, a coverage of 0 ($CC = 0$) indicates that none of the changed methods have been covered by a test.

V. Case Study

A. Goal and Research Questions

The goal of the study is to show whether change coverage is a useful metric for assessing the alignment between tests and development. We formulate the following research questions.

RQ 1: ***How much code is changed, but untested?*** The goal of this research question is to investigate the existence of changed, but untested code, to justify the problem statement of this work. Therefore, we quantify changed and untested code.

RQ 2: ***Are changed-untested methods more likely to contain field bugs than unchanged or tested methods?*** The goal of this research question is to decide whether change coverage can be used as a predictor for bugs in large code regions and is

therefore useful for maintainers and testers to identify relevant gaps in their test coverage.

B. Study Object

We perform the study on a business information system at Munich Re. The analyzed system was written in C# and its size are 340 kLOC. In total, we analyzed the system for 14 months. The system has been successfully in use for nine years and is still actively used and maintained. Therefore, there is a well implemented bug tracking and testing strategy. This allows us to gain precise data about which parts of the system were changed and why they were changed.

We analyzed two consecutive releases of the system. Release 1 was developed in five iterations in two months, and release 2 was developed in ten iterations in four months. Both releases were deployed to the productive environment due to hot fixes five times and were in productive use for six months. Note that one deployment may concern several bugs and changes in the system. The system contained 22123 (release 1) respectively 22712 (release 2) methods.

For both releases, test suites containing 65 system test cases covering the main functionality were executed three times.

C. Study Design and Execution

For all research questions, we classify methods according to the categories shown in Figure 2: Tested or untested, changed or unchanged, and whether methods contain field bugs.

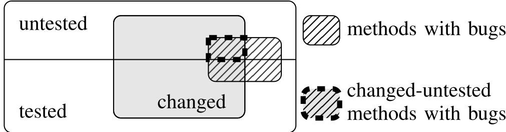


Fig. 2. Method categories used to evaluate change coverage

Study Design: First, we collect coverage and program data, then we answer RQ 1 and RQ 2 based on the collected data.

For answering RQ 1, we build method genealogies and identify changes during the development phase and relate usage data to these genealogies. With this information, we identify method genealogies that are changed-untested.

For answering RQ 2, we calculate the probability of field defects for every category of methods by detecting changes in the productive phase of the system in retrospective. This is valid for the analyzed system, since only severe bugs are fixed directly in the productive environment, which is defined by the company's processes.

We gain our results by identifying methods that are changed in the productive phase, which means they were related to a bug. We then categorize methods by change and coverage during the development phase. Based on this, we calculate the bug probability in the different groups of methods.

Study Execution: We used tool support, which consists of three parts: An ephemeral [18] profiler that records which methods were called within a certain time interval, a database that stores information about the system under consideration, and a query interface that allows retrieving coverage, change, and change coverage information. The same tool support was used in earlier studies [17], [19].

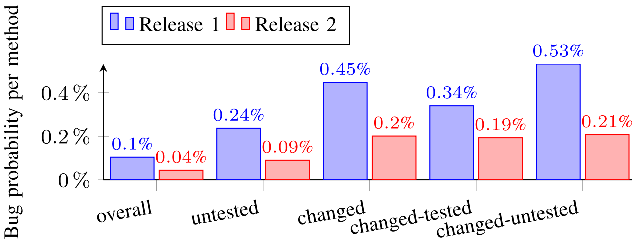


Fig. 3. Probability of fixes in both releases

Validity Procedures: We focus on validity procedures and not on threats to validity due to space limitations.

We conducted manual inspections to ensure that every bug that is identified by our tool support is indeed a bug.

To confirm the correctness of method genealogies we build based on locality and signatures, we conducted manual inspections of randomly chosen method genealogies. We found no false genealogies and have therefore a high confidence in the correctness of our technique. We also used the algorithm in our former work [17], which provided suitable results as well.

D. Results

RQ 1: Untested methods account for 34% in both releases we analyzed. 15% of all methods were changed during the development phase of the system, also in both releases. The equality of the numbers for both releases is a coincidence.

8% respectively 9% of all methods were changed-untested. Considering only changed methods, only 44% were tested in release 1 and 45% of these methods were tested in release 2. These numbers constitute that there are gaps in the test coverage of changed code in the analyzed system.

RQ 2: We found 23 fixes in release 1 and 10 fixes in release 2. The distribution of the bugs over the different change and coverage categories of methods is shown in Table I. The biggest part of bugs occurred in methods categorized as changed-untested with 43% of all bugs in release 1 and 40% of all bugs in release 2. In both releases, there are considerably less bugs in unchanged regions than in changed regions.

The probabilities of bugs are shown in Figure 3. With 0.53% in release 1 and 0.21% in release 2, the probability of bugs is higher in the group of methods that were changed-untested. This confirms that tested code or code that was not changed in the development phase is less likely to contain field defects.

E. Discussion

RQ 1: With 15% of all methods being changed and 34% of all methods being not tested, untested code and changed code plays a considerable role in the analyzed system. The high amount of changed methods results from newly developed features, which means that many methods were added during the development phase of both releases.

TABLE I
DISTRIBUTION OF FIXES OVER THE DIFFERENT CATEGORIES

Category	Release 1		Release 2	
	Absolute	Relative	Absolute	Relative
changed-tested	5	22%	3	30%
changed-untested	10	43%	4	40%
unchanged-tested	0	0%	0	0%
unchanged-untested	8	35%	3	30%

43% respectively 40% of the changed methods were not tested in the analyzed system. These high numbers also result from features that are newly developed during the development phase. For these new features, there was only a very limited number of test cases.

RQ 2: With a probability of bugs in untested-changed methods of 0.53% respectively 0.21%, this group of methods contains most of the bugs. This means that the system itself contains few bugs at the current stage of development and bugs are brought into the system by changes.

Furthermore, the probability of bugs in untested code is, in both releases, less than half of the probability in changed-untested code. Hence, we conclude that only considering test coverage is not as efficient as considering change coverage.

The probability of bugs in changed code regions is also considerably higher than in untested regions. But the combination of both metrics, test coverage and changed methods points to code regions that are more likely to contain bugs than others.

Is Change Coverage Helpful in Practice? We employed the proposed approach also in the context of Munich Re in currently running development phases. We showed the results to developers and testers by presenting code units, like types or assemblies ordered by change coverage. During the discussion of the results, we conducted open interviews with developers to gain knowledge about how helpful information about change coverage is during maintenance and testing.

Developers identified meaningful methods in changed but untested regions by using the static call graph to find methods they know. With these methods, the developers were able to identify features that remained untested. For example the processing of excel sheets in a particular calculation was changed, but remained untested afterwards. In this case, among some others, the (re-)execution of particular test cases and the creation of new test cases were issued. This increased the change coverage considerably for the code regions where the features are located. This shows that change coverage is helpful for practitioners.

VI. CONCLUSION AND FUTURE WORK

We presented an automated approach to assess the alignment of test suites and changes in a simple and understandable way. Instead of using rather complex mechanisms to derive code units that may be subject to changes, we are focusing on changed but untested methods and calculate an expressive metric from these methods. The results show that the use of change coverage is suitable for the assessment of the alignment of testing and development activities.

We also showed that change coverage is suitable for guiding testers during the testing process. With information about change coverage, testing efforts can be assessed and redirected if necessary, because the probability of bugs is increased in changed-untested methods. Furthermore, we presented our tool support that allows us to utilize our technique in practice.

However, the number of bugs we found is too small to derive generalizable results. Therefore, we plan to extend our studies to other systems to increase external validity. But the first results that we presented in this work point out that the consideration of code regions that are modified, but not very well tested is important. This motivates future work on the topic and the inference of improvement goals.

One challenge is the identification of suitable test cases from code regions to give hints to testers and developers which test case to execute to cover more changed, but untested methods. Therefore, we plan to evaluate techniques related to trace link recovery to bridge the gap to test cases.

REFERENCES

[1] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE*, 2005.

[2] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality," in *ICSE*, 2008.

[3] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, 2000.

[4] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA*, 2004.

[5] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, 2012.

[6] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *ISSRE*, 2010.

[7] V. Channakeshava, V. K. Shanbhag, A. Panigrahi, R. Sisodia, and S. Lakshmanan, "Safe subset-regression test selection for managed code," in *ISEC*, 2008.

[8] Y.-F. Chen, D. Rosenblum, and K.-P. Vo, "Testtube: a system for selective regression testing," in *ICSE*, 1994.

[9] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," in *ICSE*, 1998.

[10] M. Harrold and A. Orso, "Retesting software during development and maintenance," in *FoSM*, 2008.

[11] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ISSTA*, 2002.

[12] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, 1997.

[13] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *AST*, 2006.

[14] Y. Malaiya, M. Li, J. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Trans. Rel.*, vol. 51, no. 4, 2002.

[15] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, 2001.

[16] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," New York, USA, 1990.

[17] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Prommer, "How much does unused code matter for maintenance?" in *ICSE*, 2012.

[18] O. Traub, S. Schechter, and M. D. Smith, "Ephemeral instrumentation for lightweight program profiling," School of engineering and Applied Sciences, Harvard University, Tech. Rep., 2000.

[19] E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. Prommer, "Feature profiling for evolving systems," in *ICPC*, 2011.

APPENDIX C

Publication C [3]

Venue: AST@ICSE 2014
Acceptance rate: 43%
Length: 7 pages
Type: Full paper
Reviewed: Peer reviewed

Selecting Manual Regression Test Cases Automatically using Trace Link Recovery and Change Coverage *

Sebastian Eder,
Benedikt Hauptmann, Maximilian Junker
Technische Universität München, Germany
{eders,hauptmab,junkerm}@in.tum.de

Rudolf Vaas, Karl-Heinz Prommer
Munich Re, Germany
{rvaas,hprommer}@munichre.com

ABSTRACT

Regression tests ensure that existing functionality is not impaired by changes to an existing software system. However, executing complete test suites often takes much time. Therefore, a subset of tests has to be found that is sufficient to validate whether the system under test is still valid after it has been changed. This test case selection is especially important if regression tests are executed manually, since manual execution is time intensive and costly.

To select manual test cases, many regression testing techniques exist that aim on achieving coverage of changed or even new code. Many of these techniques base on coverage data from prior test runs or logical properties such as annotated pre and post conditions in the source code. However, coverage information becomes outdated if a system is changed extensively. Also annotated logical properties are often not available in industrial software systems.

We present an approach for test selection that is based on static analyses of the test suite and the system's source code. We combine *trace link recovery* using latent semantic indexing with the metric *change coverage*, which assesses the coverage of source code changes. The proposed approach works automatically without the need to execute tests beforehand or annotate source code. Furthermore, we present a first evaluation of the approach.

Categories and Subject Descriptors

K.6.3 [**Management of Computing and Information Systems**]: Software Management; D.2.4 [**Software Engineering**]: Testing and Debugging

*This work was performed within the Software Campus project *ANSE*; it was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01IS12057. The authors assume responsibility for the content.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

AST'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2858-6/14/05...$15.00
http://dx.doi.org/10.1145/2593501.2593506

General Terms

Software Maintenance, Coverage Testing, Tracing

Keywords

Regression tests, test selection, manual system tests, test coverage, trace link recovery, software maintenance

1. INTRODUCTION

Business information systems often run and are maintained for up to two or three decades. For these long living systems, existing functionality must be preserved in spite of modifications. Therefore, regression testing detects changes that break existing functionality. In the field of business information systems, regression testing is often done manually by using test cases written in natural language.

An example of a manual test case is illustrated in Figure 1: A test case is separated into different steps and every step contains an action the tester has to perform and the expected result the tester has to check.

Regression testing ensures that existing functionality of the system under maintenance is not impaired. However, due to the size of test suites and limited resources, just subsets of test suites can be executed. Thus, the selection strategy is crucial for the effectiveness of regression testing. During maintenance, when a system is changed and tested afterwards, code changes are possibly missed by tests, which leads to field bugs. As we already showed in previous work [8], the rate of field bugs is higher in untested and changed code than in other code regions and therefore, it is more important to test changed methods. Based on these results, we now present an approach to regression test case selection, which targets changed code, for manual system tests which are written in natural language.

	Action	Expected result
1	Start the system	System is ready
2	Login using a valid username and password	Systems displays a message that you are logged in successfully
...		
6	Enter data for calculation	System shows dialog for exporting the results
7	Export results to Excel	Excel file is created
8	Open files	Data is correct

Figure 1: Example: Manual test case.

Problem: Existing test case selection approaches often rely on code coverage data of tests, or on logical properties such as pre and post conditions annotated in source code [10] used to predict which code regions might be executed by a test. However, these approaches are not generally applicable because coverage data from prior test runs becomes outdated if the system under test underlies extensive changes. Furthermore, logical properties are usually not contained in industrial software systems, because they are costly and difficult to create and maintain. This renders existing regression test selection techniques difficult to apply in real world scenarios.

Contribution: We propose an approach based on static analysis to identify regression tests that cover changed code. We do not rely on coverage data nor on annotations in source code, but present a fully automated approach based on *Latent Semantic Analysis* [7] to relate manual tests to source code, recovering trace links between both. With these trace links, we select test cases that have to be executed to test a certain piece of code. We use the metric *change coverage* to identify what parts of a system have been changed and are therefore subject to regression testing. Combining trace link recovery and change coverage enables identifying test cases as regression tests to test the changed code. Goal of the approach is, given a set of methods, that were changed, but not tested, to select test cases that execute methods. Additionally, we provide initial results of an evaluation of our technique, showing that the approach performs well in selecting relevant tests.

2. RELATED WORK

Selective regression testing or regression test selection techniques identify test cases based on changes of the system under test. There are several approaches to this topic [3, 4, 12, 22, 5]. However, these approaches mainly use code coverage information from prior test runs to identify which test case executes which code regions. Furthermore, most approaches focus on being *safe*, in terms of being capable of uncovering the same bugs the complete test suite could also reveal. We focus on selecting at least one test case that executes changed but untested methods. Additionally, these approaches tend to recommend many test cases which might be too much for an application in practice, especially if the tests have to be executed manually [15]. In contrast, we only select test cases that target changed code that has not been covered by tests since their modification and use only fully automated static analyses and therefore do not require the test suite to be executed to identify relevant test cases nor do we rely on logical properties of test cases or the system.

Test case prioritization orders test cases on their likeliness to detect bugs based on coverage metrics such as coverage of changes [20, 23]. Our work is related to the field of test case prioritization, since our approach can be used for this task too. As noticed in [8], changed methods indeed cause more field bugs than unchanged methods. Therefore, we suggest testing untested methods and prioritizing test cases covering these methods high.

Trace link recovery and feature location focus on uncovering relations between different artifacts such as locating higher level features in source code. Many approaches borrow from the fields of information retrieval and natural language processing. Cleland-Huang et al. [6] give an overview on the applicability of trace link recovery and proposes best practices. To the best of our knowledge, there is no approach using traceability link recovery for test case selection, which is the main contribution of this paper.

There are some approaches that target the recovery of trace links between natural language documents in software engineering (e.g. [16, 13]). Besides other algorithms [18], these approaches often use the Vector Space Model or LSI for retrieving links between documents. Falessiet al. [11] report on using and calibrating retrieval algorithms. We use this knowledge to calibrate the presented approach uses and rely on best practices presented by other researchers.

Much work has been done on recovering links between code and design artifacts. Especially Zhao et al. [24] present an approach very similar to our approach: Technically, they also use the call graph of a system to find representative code regions and propagate the tracing results through the call graph (we illustrate our approach in detail in Section 3). They use this technique to relate requirements to code regions and validate their approach based on two open source systems. As they achieve very good results (ca. 100% precision and ca. 95% recall) in linking code regions to requirements, we follow their work. However, they note that requirements have to be well designed to match to code. We use a very similar approach, but evaluate it on real world test cases from industry, rather than requirements documents.

Similarly to Zhao et al. [24], an approach using LSI and the static call graph of the software system under maintenance is presented by Charrada et al. [2]. They use traceability link recovery to detect outdated requirements based on source code changes, by extracting concepts from the source code that are relevant for requirements. We conform to their approach by also using the call graph, but do not consider requirements, but link source code methods to regression test for selection.

The selection properties to consider when tracing from source code to test cases or other artifacts is crucial. In [1], a study about which properties of the source code to take into account is given. They suggest using class names for tracing, but also show that using method names produces good results. Encouraged by this, we use method signatures, since tracing on class level is too coarse in our context.

Lucia et al. [17] introduce an approach facilitating tracing between code and test cases. They also use LSI as technique for comparing documents, but do not give details about how code is preprocessed prior to comparing it to test cases. Furthermore, it remains unclear, whether the test cases are written in natural language. Our approach is dedicated to trace from source code to test cases written in natural language for manual testing.

3. APPROACH

In this section, we introduce our approach used for regression test case selection. After giving a short overview of the approach, we explain the input to our approach and afterwards illustrate the main parts: Change Coverage and Trace Link Recovery using latent semantic indexing and the static call graph of the system under consideration.

Overview: To select test cases, we first seek code areas with methods that were changed but not tested as presented in [8]. We then run the trace link recovery algorithm (based on LSI) to find test cases that might cover these gaps. But as this approach is based on similarities, it *suggests* test cases.

3.1 Starting Point

The algorithm uses test case descriptions in natural language. We consider a test case as a plain text document written in natural language.

Furthermore, the approach expects a software system as input. Currently, we only support the programming language C#. The software system is given to the algorithm as compiled Intermediate Language (IL) code[1]. Additionally, the approach uses different versions of the same software system under maintenance to detect changes to the source code. Changes are considered only at method level (a method was changed or not). We consider a method as changed if its IL code differs from the previous version.

For calculating change coverage (see the following section), we use profiling data from an ephemeral profiler [21] as in our earlier work [8, 9]. The profiler does not record how often a method was called, just whether it was called during a fixed time interval. This data suffices to gain valuable insights into the test coverage of a system under maintenance.

With the data about method changes and the coverage information on method level, change coverage is calculated.

3.2 Change Coverage

To select code regions for which we select test cases, we use *change coverage*, since it quantifies the amount of changes covered by tests. Change coverage is calculated by the fraction number of methods that were changed and tested afterwards and the number of all changed methods (as shown in Equation 1). A detailed description can be found in [8]. The same study shows that collecting coverage information at method level is fine grained enough to gain insights into code coverage in practice, but does not produce too much performance overhead for practical application.

$$\text{change coverage} = \frac{\#\text{methods changed-and-tested}}{\#\text{methods changed}} \quad (1)$$

A change coverage of 1 means that all methods which have been changed have been tested after their last change. On the contrary, a coverage of 0 indicates that none of the changed methods has been covered by a test.

We perceive change coverage as a simple metric that is easy to understand. As reported earlier [8], this helped transferring it into practice.

3.3 Latent Semantic Indexing

As our approach to trace link recovery is based on Latent Semantic Indexing (LSI) [7], we explain it in this section. LSI is a technique to compute for each pair from a corpus of documents how similar they are to each other. The measure for the similarity is a value between zero and one, where one means identical [2] .Compared to other techniques LSI identifies groups of words that belong to a common concept (e.g. car and automobile belong to the same concept), which makes the computation of similarities more precise. Schematically LSI works as follows (and as illustrated in Figure 2): First, every document is represented by a vector in the space of words. Each entry in this vector denotes the weight that this word has with respect to this document (e.g. how often it occurs). Thus the whole set of documents is represented by a term-document matrix. Now LSI applies a procedure called Singular Value Decomposition (SVD). The result of SVD is a smaller matrix where terms are replaced by concepts, thus each document is now represented by a vector in the space of concepts. The similarity between two documents can then be calculated as the distance of the correspondent vectors.

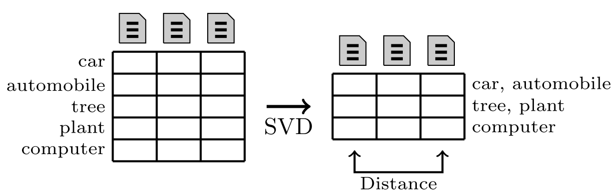


Figure 2: Schematic working of LSI.

3.4 Trace Link Recovery

The procedure for trace link recovery is divided into several steps, as illustrated in Figure 3. These steps are explained in detail below.

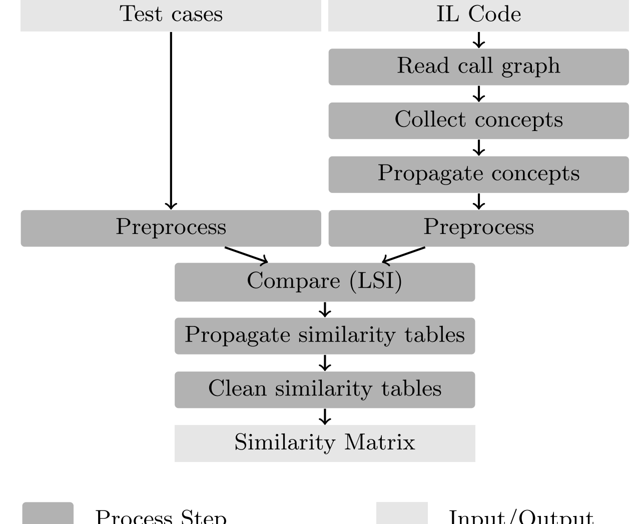


Figure 3: Overview of the approach.

Preprocessing of test cases consists of stemming and stop word removal[3]. Although stemming is theoretically not necessary when using LSI since the model recognizes synonyms given a dataset of sufficient size [14]. However, the size of the test suites we analyzed did not suffice to correctly classify synonyms (in the resulting vector space, two words with the same meaning are too far away from each other affecting the results in a negative way). Therefore, we use stemming to ensure every word is reduced to its normal form, expecting that stemming improves the results [14].

Transferring source code methods into documents: We divide this task in four steps:

Read call graph: The static call graph is retrieved directly from the intermediate language code (IL code).

[1]Call graph dependencies are easier to analyze in IL code.

[2]In the remainder of the paper, documents do not have to be identical, as we use the results from LSI as a measure of similarity.

[3]We used the stemming approach introduced by Porter [19].

Collect concepts: We rely on the common convention that identifiers are denoted in camelCase or different concepts are separated by underscores or dashes. With this assumption, we can split methods signatures into the concepts they contain. For example the method `getRatingAgency()` is transferred into the list [get, rating, agency].

Propagate concepts: We extract the concepts from the signatures of all methods that are reachable in the static call graph from the method under consideration. If, for example, the method `getNameOfAgency()` is reachable from `getRatingAgency()`, the resulting list of concepts is [get, rating, agency, get, name, of, agency]. With this technique, we also get information for methods that are entry points to the system.

Preprocess: On this list, we perform stemming and stop word removal resulting in the list of concepts [get, rate, agenc, get, name, agenc][4]. Note that the same stemming is performed on test cases resulting in the same stemmed terms occurring in the test cases.

We represent the result of the preprocessing of methods as a set of text files, one per method, containing all the concepts of the method itself, and the concepts of all methods that are reachable from this method. These files do not contain correct natural language, but suffice to use it with LSI.

Compare: After preprocessing is finished, we compare the documents resulting from methods and test cases based on LSI. The outcome is a matrix containing the values that indicate how similar a test case is to a method. The LSI comparison is the most time consuming task in our approach.

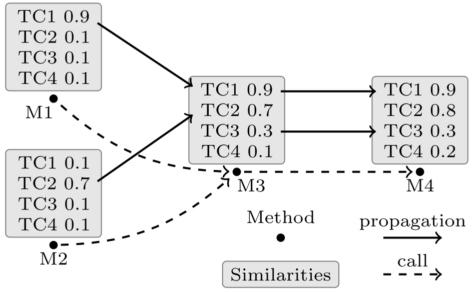


Figure 4: Example: Propagation of similarities.

Propagate similarity tables: Each system contains methods which are not similar to any test case, and therefore cannot be linked[5]. However, using the call graph, we know how to reach these methods. Therefore, we traverse along the call graph reversely to find methods which we can relate to tests with more confidence and propagate them back along the call graph. The result is that every method is annotated with the maximum similarity values for every test case from all methods it can be reached from in the call graph. Figure 4 shows an example of this propagation. With this procedure, we find tests cases for methods that are not mentioned in test cases directly.

Clean similarity tables: To this end, we calculated the similarities between every method and every test case. Since we only want to select relevant test cases, we keep only the elements in the list of similarities of a method where we expect that test case to execute this method. This is done as follows: Given a list of similarity values, sorted in descending order, we cut the list at the point where the biggest distance between two consecutive similarity values occurs. This is the same algorithm as used in [24]. Table 1 shows an example of the cutting algorithm: Between the test cases TC1 and TC2, the distance of the similarity values is 0.5, whereas the distance between TC3 and TC1, and TC2 and TC4 is only 0.1. Therefore, the list is cut between TC1 and TC2, since the distance of similarity values there is the biggest.

Table 1: Example for cutting similarity lists.

Test Case	Similarity
TC3	0.9
TC1	0.8
TC2	0.3
TC4	0.2

(a) Method M4 (before)

Test Case	Similarity
TC3	0.9
TC1	0.8

(b) Method M4 (after)

The result of the algorithm for trace link recovery is a matrix containing similarity values denoting the similarity of all methods to all test cases. An example of a similarity matrix is illustrated in Table 2, where TC1 to TC4 are test cases and M1 to M5 are source code methods.

Table 2: Example similarity matrix.

	M1	M2	M3	M4	M5
TC1	0.9	0.0	0.9	0.9	0.9
TC2	0.0	0.7	0.7	0.8	0.7
TC3	0.0	0.0	0.0	0.0	0.0
TC4	0.0	0.0	0.0	0.0	0.8

4. INITIAL EVALUATION

For evaluation, we conducted a case study in an industrial environment with which we answer the following questions:

RQ1 How accurate is the approach to trace link recovery? This question assesses for how many methods we find test cases that execute the methods.

RQ2 What is the influence of characteristics of test cases on the approach? With this question we measure the influence of characteristics of test cases such as the verdict of test cases and the number of steps that test functionality specific to the system on the accuracy of our approach.

4.1 Study Object

System under test: We perform the study on a business information system at Munich Re, which is one of the world's leading reinsurance companies with more than 47,000 employees in reinsurance and primary insurance worldwide. For their insurance business, they develop a variety of custom software systems. The business information system analyzed in this study implements damage prediction functionality and supports ca. 150 expert users in over ten countries. Table 3 illustrates the study object's main characteristics.

We chose this system as study object for several reasons: The system has been successfully in use for ten years and is still actively used and maintained. Moreover, it is a web

[4]"agenc" and "rate" result from stemming, and "of" is removed as it is a stop word.

[5]For example, logging facilities that are not mentioned in the test cases.

Table 3: Study object: System under test.

Language	C#
Age (years)	10
Size (kLOC)	340
Size (#methods)	39398

application, thus offering a single point for coverage data collection and accessible for researchers outside Munich Re.

Test suite: For an initial evaluation, we selected four manual test cases written in natural language[6]. The test cases we selected cover between 1297 and 2300 methods, and 2711 methods in total. The execution of the test cases takes between 5 and 15 minutes. The characteristics of the test cases are illustrated in Table 4. Column *Steps* shows the number of steps (consisting of action and expected result) a test case has and the column *Methods* denotes the number of methods a test case executes.

Table 4: Study object: Test cases.

Test Case	Steps	Methods
TC1	5	2300
TC2	7	1763
TC3	5	1297
TC4	11	1602

4.2 Study Design and Data Collection

Setup phase: To have reference values for the evaluation, we execute all test cases manually on the system under test and measure the code coverage for each test case. This data will deal as an oracle for our evaluation. However, some parts of the code have not been executed at all since there were no test cases covering this code. Since we have no data for these parts, we focus on executed methods in this study.

This oracle enables us to compare the results from our approach with the actual coverage data.

Evaluation phase: We define a link suggested by our tool from a source code method to a test case as correct, if the method has been executed during the actual test run of the suggested test. We calculate the accuracy of our approach by counting how many suggested links of the methods we executed in the setup phase are correct.

For answering RQ1 (How accurate is the approach to trace link recovery?), we calculate the accuracy of links from source code methods to test cases as the ratio of correctly linked methods and all considered methods as shown in Equation 2.

$$\text{accuracy} = \frac{\#\text{correctly-linked}}{\#\text{considered-methods}} \quad (2)$$

We compare our approach to randomly guessing a test case for every method as a baseline. Furthermore, we apply Pearson's Chi-Squared Test to test the null hypothesis H_0: The presented approach produces no significantly different results as the baseline. We chose a p value of 0.01.

To answer RQ2 (What is the influence of characteristics of test cases on the approach?), following the links from test cases to source code methods, we investigate the influence of the test cases on the accuracy. We calculate in how many cases the links suggested by our tool from test cases to source code methods are correct. To measure the performance of our approach, we calculate its precision using Equation 3, where true positives are links between methods that are really executed by the linked test case and false positives are links that connect methods with test cases that do not execute the method.[7]

$$\text{precision} = \frac{\#\text{true-positives}}{\#\text{true-positives} + \#\text{false-positives}} \quad (3)$$

4.3 Results

Results for RQ1 (How accurate is the approach to trace link recovery?) The proposed approach suggests at least one test case for every method. As we have four test cases for evaluation, assuming that every method is executed by exactly one test case, guessing would yield a chance of $\frac{1}{4} = 25\%$ to hit the right test case. Since methods might be executed by more than one test case, we define the baseline for RQ1 higher: From our analyses we know that each method is executed by 2.56 test cases in average. Therefore randomly guessing yields a chance for guessing a correct test case of $\frac{2.56}{4} = 64\%$ as a baseline for our evaluation.

For the first research question, the results are presented in Table 5: We considered 2711 source code methods in total and linked 2444 methods with a test case that executes them. With 90% of all considered methods classified to a correct test case, we gain considerably better results than the baseline.

Table 5: Results: Methods.

Total	Correctly linked	Accuracy
2711	2444	90%

We link 1.75 test cases to a method in average with a variance of 0.60; 1237 methods are linked to one test case, 909 methods are linked to two test cases, 565 methods are linked to three test cases, and no method is linked to four test cases (this results from the design of the algorithm for cutting similarity lists).

For a statistical interpretation of the results, we observed 2444 methods as linked to a correct test case and 267 methods not linked to a correct test case. With the baseline of 64% of correctly linked test cases by randomly guessing (and 36% incorrectly linked methods), we expect 1740.5 methods to be linked to the correct test case and 970.5 methods to be incorrectly linked. The result of the Chi-Squared-Test ($\chi^2 = 764.3$) shows, that the null hypothesis H_0 can be

[6]For the study, execution traces of test cases are necessary. Since capturing execution traces of manual test cases in an industrial environment is a difficult task, because usually there is no profiling environment and test cases require deeper knowledge of the system under test, we had to face the trade-off between test cases from industry (but less) or artificial test cases (more). To gain more realistic results, we chose test cases from industry.

[7]Recall would measure whether our approach finds all methods executed by a test case or reversed, whether the approach finds all test cases for a method. Since the goal of the presented approach is to close gaps in change coverage and therefore executing methods that were changed, but not tested, it is enough to find one test case for every method, what we do *by design* and thus recall is not appropriate.

rejected and the approach outperforms randomly guessing with statistical significance ($p = 0.01$).

Results for RQ2 (What is the influence of characteristics of test cases on the approach?) For the second research question, we present the results for every test case in Table 6 focusing on links between test cases and methods. Column *Total links* contains the total number of all links suggested by our approach for the corresponding test case. Column *Correct links* contains the number of correct links from methods to the test case. The precision for every test case is contained in the second column indicating how many links of the suggested links for every test case are correct. Overall, we suggested 4750 links and 3619 (76%) of these links are correct.

Table 6: Results: Test cases.

Test Case	Precision	Total links	Correct links
TC1	95%	1558	1480
TC2	73%	852	618
TC3	68%	523	355
TC4	64%	1817	1166
Overall	76%	4750	3619

4.4 Discussion

RQ1: The use case of our approach is finding test cases that execute methods that were changed, but remained untested. Our approach suggests a correct test case in 90% of all considered methods while suggesting 1.75 test cases in average. These results indicate that the approach is suited for the use case, because first, the approach has a high hit rate and second, the approach limits the number of test cases to select.

RQ2: To enable a more detailed discussion, characteristics of the test cases we chose for the initial evaluation are presented in Table 7. Column *Interactions* shows the number of actions and checks that target the system under test (and not an external tool, like step 8 in the example in Figure 1) and therefore contain more words that are characteristic to the system. The value in brackets is the total number of steps contained in the test case (see Table 4). This influences our approach since the more characteristic words are contained in a test case the more terms of the test case are found in source code methods. Furthermore, we chose three test cases with a successful verdict and one failing test case. Column *Verdict* contains the outcome of the test case: Success means the test case was performed without revealing errors, and a failing test case produced errors. Table 7 also shows that we chose heterogeneous test cases in terms of verdict and system specific interactions for our evaluation.

We deliberately chose test cases containing not only interactions with the system under test but also with external tools, because we observed many test cases in the test suite we perform our analysis on exhibiting this characteristic. To gain more representative results for the system under consideration, it is thus necessary to also choose test cases with parts that are considering external systems.

Table 7: Study object: Test cases (extension).

Name	Interactions	Verdict
TC1	5 (5)	Success
TC2	7 (7)	Fail
TC3	2 (5)	Success
TC4	4 (11)	Success

We observe that the precision for TC1, which was executed without errors and only contains interactions regarding the system directly produces the best result considering precision (94%). This is not surprising, since this test case contains mainly words regarding the system and furthermore did not trigger error handling due to errors and thus did not execute program parts that are not mentioned in the test case. However, also TC2, which produced an error and could not be executed completely, can be related to methods with a precision of 73% (we successfully executed four steps in TC2). The remaining test cases T3 and T4 focus heavily on checks regarding data in an exported file. This leads to slightly worse results than we observe with TC1 and TC2.

These results indicate that the precision even rises when we consider test cases that contain more interactions specific to the system and are not failing.

Conjecture: The results show that out approach is able to suggest correct test cases for methods. Therefore, we consider it suitable for finding test cases for methods that are untested, but can be executed by an existing test case. Based on this, we formulate the following hypothesis which is subject to further investigation: Using trace link recovery based on LSI facilitates regression test selection for changed, but yet untested source code methods.

4.5 Threats to Validity

As we performed our trace recovery approach on only one system and four test cases, the results are hardly generalizable. However, we chose a system with a common size and complexity in business information systems. We chose heterogeneous test cases, which means successful and failing test cases and test cases with less or more system interaction and system specific text, to reduce the bias introduced by the small number of test cases.

It is possible that our collection of usage data for test runs did not collect all necessary data for the oracle. This can happen, for example, when caching mechanisms become active after one run of the system. We expect this effect to be small, since developers told us there are no such mechanisms. The collection procedure itself has shown to be working with sufficient accuracy in earlier research [9, 8].

Furthermore, the tracing algorithm might not be implemented correctly. We mitigated this threat by testing our implementation on other artefact types.

5. CONCLUSIONS AND FUTURE WORK

We presented an approach based on trace link recovery (using latent semantic indexing [7]) and change coverage [8] for regression test selection for test cases written in natural language and an initial evaluation of the approach. The approach yields the advantage that a changed program does not need to be executed again to gain enough information for selecting test cases or needs to be annotated again with logical properties to enable test case selection. The evaluation showed that the tracing mechanism works well for manual system tests: Our approach selects a correct test case for 90% of all considered source code methods and outperforms guessing randomly with statistical significance.

The repeated execution of a system under test to gain coverage data of test cases is not desirable, since this again is a task which needs resources that are usually missing, and this is why test case selection is performed in the first place. The annotation of logical properties to industrial software systems is difficult due to the size and complexity of these systems. Furthermore, modern programming languages, for example C# lack clear formal semantics, which is a prerequisite for gaining formal logical information about a system.

Our approach suggests test cases for every method. One effect of this is that if there is no test case covering the code region under consideration, our approach still might suggest a test case based on the concepts contained in the methods of the code region not covering the code. It is subject to future work to investigate effects of this behavior.

In our future work, we want to increase external validity by studying more manual test cases on different systems. Furthermore, we plan to use the trace link recovery approach for not only tracing to test cases, but also to other artifacts as requirements documents. Furthermore, we evaluate how different calibrations of the approach affect the results. Additionally, we plan to investigate the influence of missing test cases on our approach.

Another promising direction is the combination of the presented approach to trace link recovery with coverage based test case selection techniques (for example [5]). We expect to get good results using more complex techniques than change coverage for finding code that has to be tested. Furthermore, we are confident to gain better applicability of existing approaches, which rely on coverage data by applying our trace link recovery algorithm instead of using test coverage data.

6. ACKNOWLEDGMENTS

The authors thank Daniela Steidl, Jonas Eckhardt, Henning Femmer, and Elmar Jürgens for their helpful comments on this work.

7. REFERENCES

[1] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability recovery: selecting the basic linkage properties. *Sci. Comput. Program.*, 40(2–3):213 – 234, 2001.

[2] E. Ben Charrada, A. Koziolek, and M. Glinz. Identifying outdated requirements based on source code changes. In *RE '2012*, 2012.

[3] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2), 2001.

[4] V. Channakeshava, V. K. Shanbhag, A. Panigrahi, R. Sisodia, and S. Lakshmanan. Safe subset-regression test selection for managed code. In *ISEC '08*, 2008.

[5] Y.-F. Chen, D. Rosenblum, and K.-P. Vo. Testtube: a system for selective regression testing. In *ICSE '94*, 1994.

[6] J. Cleland-Huang, R. Settimi, E. Romanova, B. Berenbach, and S. Clark. Best practices for automated traceability. *Computer*, 40(6):27–35, 2007.

[7] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *J. Am. Soc. Inf. Sci. Technol.*, 41(6):391–407, 1990.

[8] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer. Did we test our changes? assessing alignment between tests and development in practice. In *AST '2013*, 2013.

[9] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Prommer. How much does unused code matter for maintenance? In *ICSE '2012*, 2012.

[10] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Inf. Softw. Technol.*, 52(1):14–30, 2010.

[11] D. Falessi, G. Cantone, and G. Canfora. Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. *IEEE Trans. Softw. Eng.*, 39(1):18–44, 2013.

[12] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. *SIGPLAN Not.*, 36(11), 2001.

[13] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Trans. Softw. Eng.*, 32(1):4–19, 2006.

[14] D. Jiménez, E. Ferretti, V. Vidal, P. Rosso, and C. F. Enguix. The influence of semantics in ir using lsi and k-means clustering techniques. In *ISICT '03*, 2003.

[15] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke. Regression test selection of manual system tests in practice. In *CSMR '11*, 2011.

[16] M. Lormans and A. van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *CSMR '06*, 2006.

[17] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.

[18] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *ICPC '2010*, 2010.

[19] M. F. Porter. An algorithm for suffix stripping. *Program: Electronic Library & Information Systems*, 40(3):211–218, 1980.

[20] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10), 2001.

[21] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, School of engineering and Applied Sciences, Harvard University, 2000.

[22] D. Willmor and S. M. Embury. A safe regression test selection technique for database-driven applications. In *ICSM '05*, 2005.

[23] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *ISSRE '97*, 1997.

[24] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.

APPENDIX D

Publication D [4]

Venue: ICSME 2014
Acceptance rate: 36%
Length: 5 pages
Type: Full paper
Reviewed: Peer reviewed

2014 IEEE International Conference on Software Maintenance and Evolution

Which Features Do My Users (Not) Use?

Sebastian Eder, Henning Femmer, Benedikt Hauptmann, Maximilian Junker
Technische Universität München, Germany

***Abstract*—Maintenance of unused features leads to unnecessary costs. Therefore, identifying unused features can help product owners to prioritize maintenance efforts. We present a tool that employs dynamic analyses and text mining techniques to identify use case documents describing unused features to approximate unnecessary features. We report on a preliminary study of an industrial business information system over the course of one year quantifying unused features and measuring the performance of the approach. It indicates the relevance of the problem and the capability of the presented approach to detect unused features.**

I. Introduction

Software systems contain unused features. Studies report that in practice 28% [1] to 45% [2] of a system's features are unused. These features lead to code areas that are not used in a productive environment. Maintaining unused code leads to unnecessary costs, since unused code is often unnecessary [3]. However, product owners directing maintenance efforts often do not know the actual usage of their system in its productive environment. Hence, from the perspective of a product owner, the question arises: "How can we identify the unused features to prevent unnecessary maintenance cost?".

To answer their question, we would like to collect usage data on feature level. Due to technical reasons, this is not well established in practice: code profilers record usage data on source code level, whereas profilers on feature level require instrumentation or annotation of source code that explicitly maps source code to features (see, e.g., [2]), which leads to additional maintenance efforts.

Problem: Usage data on code level reveals only which parts of the code, instead of features, are used or unused. As such, this information is helpful for developers who know the source code of the system and can decide whether to keep or remove source code based on this information. But this information does not help product owners, who are on project management or requirements level and, more important, who do not know the system's source code. This is just one reason for product owners being unable to relate unused code to features. Since product owners do not want to direct maintenance to unused features, they require usage information on the level of features, which is not given by profilers employed in industry. Hence, to help product owners planning maintenance efforts, we need to transfer runtime usage information to higher level usage data and bridge the gap between code level usage data and features.

Contribution: We propose an approach using text mining techniques to bridge the gap between code level usage information and features. In this approach, we approximate to features by analyzing use case documents (see, e.g., [4]). By extracting concepts from source code methods and matching the content of use case documents to these concepts, we can suggest those use case documents that describe most likely unused features.

We analyzed a real-world business information system from the reinsurance domain where features are documented in 46 use cases. We present a preliminary study, based on usage data collected for more than one business year, which lead to the discovery of two use case documents that describe unused features and two use case documents containing large parts that were not performed by the actual users of the system. The results show that the precision and recall of the approach are high, and therefore, indicate the validity of our approach.

II. Related Work

Feature Profiling: Besides [2], we are not aware of research work eliciting feature level usage information on the level of requirements. However, one approach to detect unused features is linking code to features manually. This is a tedious task, if even feasible in practice, since possibly thousands of locations in the source code have to be linked to requirements documents. Additionally, these links possibly become outdated in a changing system and this leads to even more efforts.

Software Usage Mining: Several approaches gather information about the usage of software: techniques for web usage mining [5] gained much attention by researchers, however, there are also techniques for other types of systems [2], [6], [7]. The difference to our work is that those approaches do not establish connections to requirements documents, such as use cases. Furthermore, these approaches often rely on a certain structure or instrumentation of programs, e.g. [7] is specific to software built on top of the Eclipse Framework. Our approach, in contrast, is generic as it inspects the source code text.

Feature Location: Feature Location refers to the task of identifying code areas that implement a certain feature. An extensive survey on feature location is given by Dit et al. [8]. Among the techniques proposed for this task are static and dynamic analyses [9] as well as text mining techniques [10], which is what we applied here. Text mining has been used for feature location, e.g., in [11] where features are located based on natural language documents. The main goal of feature location techniques is to answer a query about a feature by providing a list of matching source code areas. This differs from our goal, as we want to decide which features are not executed on the system.

Trace Link Recovery: Trace Link Recovery focuses on uncovering relations between different software-related artifacts. A taxonomy of trace link recovery techniques has been published by Cleland-Huang [12]. In terms of this taxonomy we use a technique based on term matching. Uncovering traces between requirements documents and source code to ease maintenance has been investigated by Charrada [13],

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.71

[14]. Lucia et al. [15] present an approach and a tool to automatically uncover trace links between a wider range of artifacts, among them use cases. Lormans [16] uses trace link recovery to identify requirements that have not been implemented. Compared to these works, which often use text mining techniques [17], we share the underlying techniques such as LSI, but establish relations between use cases and source code with the goal to find unexecuted features.

III. Approach

We explain our approach to identifying unused features expressed by use case documents, which is based on usage data analysis and Latent Semantic Indexing (LSI) for linking unused code to use case documents.

A. Background: Latent Semantic Indexing

LSI [18] measures the similarity between documents contained in a document corpus. Similarity is expressed by a value between -1 and 1, where a higher value means the compared documents are more similar[1]. LSI identifies words belonging to a common concept (e.g., 'car' and 'automobile'), enabling it to deal with synonyms.

LSI starts with creating a term document matrix (*terms* × *documents*) with entries for each word in each document. The entries are calculated by a global and local weighting function for each word. On this matrix, given the number of desired dimensions (which can be interpreted as the number of concepts), singular value decomposition is performed, resulting in a smaller matrix where words are replaced by their concepts. This matrix represents every document as a vector in the space of concepts. The similarity of documents is calculated by comparing their vectors using cosine similarity.

B. Finding Unused Features

The approach is divided into several steps as illustrated in Figure 1 and uses the variables listed in Table I. It assumes the source code using the same concepts in the same language as the use case documents.

TABLE I. Approach: Variable names and meanings.

Variable	Description
M	Set of all method names
U_i	The ith use case document in all use case documents
M_{used}, M_{unused}	The set of used/unused methods
C_{used}, C_{unused}	The set of concepts in method names in M_{used} / M_{unused}
$s_{U_i,used}$, $s_{U_i,unused}$	The similarity of U_i to C_{used} / C_{used}
Σ_{U_i}	The score of U_i

Input:

- *Use case documents* written in natural language describing the flow of steps users perform.
- *Usage data* collected by an ephemeral profiler[2] [19]. The resulting log files list the executed methods[3].
- *Source code* of the software system[4].

[1] The range of some other implementations of LSI is between 0 and 1.

[2] This profiling technique imposes less performance impact than classical profiling techniques. This improves the applicability in productive systems.

[3] But no information about how often a method was called.

[4] The approach also works on Java-Bytecode and IL Code for C#, since we rely only on method signatures that are extractable from these artifacts.

① Extract and Filter Methods:
Input: Source code
Procedure: Filter out generated code and test code[5] and extract method names M from the remaining methods' signatures.
Output: M.

② Partition Methods:
Input: M, usage data log files
Procedure: Partition method names by usage (used or unused).
Output: M_{used}, M_{unused}.

③ Extract Concepts for Method Sets:
Input: M_{used}, M_{unused}
Procedure: Extract concepts[6] from method names. We rely on the coding convention that method names are written in camelCase or single words are split by underscores. For example, the method `getRatingAgency()` is transferred into the list of concepts `[get, rating, agency]`.
Output: C_{used}, C_{unused}.

④ Compare (LSI):
Input: Corpus of use case documents C_{used} and C_{unused}.
Procedure: Extract cleaned text[7] from use case documents. Compute the similarities of every use case document U_i to the word sets C_{used}, C_{unused} using LSI[8].
Output: For each use case document U_i: $s_{U_i,used}$, $s_{U_i,unused}$.

⑤ Calculate Scores:
Input: For every use case document U_i: $s_{U_i,used}$ and $s_{U_i,unused}$
Procedure: Calculate the score Σ_{U_i} for every use case document U_i as: $\Sigma_{U_i} = s_{U_i,unused} - s_{U_i,used}$. In this step, we want to score a use case document higher, the more similar it is to the concepts of unused methods, and lower, the more similar it is to the concepts of used methods, since there might be use case documents that match well to both sets of methods.
Output: For every use case document U_i: Σ_{U_i}.

⑥ Sort Use Case Documents:
Input: Σ_{U_i} for every use case document U_i
Procedure: Sort the use case documents by their score in descending order.
Output: Ranking of use case documents. Higher ranked use cases are more likely to describe unused features.

Parameter Estimation: The parameters used for configuring LSI highly impact the results. Therefore, we suggest to estimate parameters based on a sample use case document that contains unused features and is identified manually by a system expert. We iterate through possible parameters and choose the parameters with which the proposed approach ranks this use case document highest. The rationale behind this approach is that identifying just one use case document that expresses an unused feature can be done with less effort than identifying all – which then is facilitated by our approach.

[5] Test methods are never executed in the productive environment. Generated code does not contain words relevant for our approach and has not been manually maintained.

[6] Single words contained in method names.

[7] We omit information like authors or version history. Additionally, we remove stop words and stem the text.

[8] To compare word sets to documents, we generate one document for each concept set by writing the contained words to a text document.

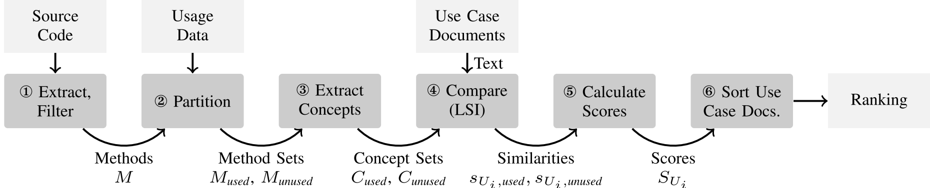


Fig. 1. Schematic illustration of the approach. Light boxes are input or output artifacts, while dark boxes are steps in the approach. Arrows indicate data flow.

IV. Preliminary Study

The goal of this preliminary study is to show the relevance of the research problem and to validate the applicability of the approach in a real world case study. Consequently, we formulate the following two research questions:

RQ1: How many use case documents that express unused features are in the system? This question targets the existence and number of unused features to show that the problem of unused features exists in practice.

RQ2: Is the approach capable of detecting use case documents describing unused features? This question focuses on the validity and performance of our approach in terms of precision and recall, and the applicability in real world examples.

A. Study Object and Subject

We perform the study on a business information system at Munich Re, which is one of the world's leading reinsurance companies with more than 47,000 employees in reinsurance and primary insurance worldwide. For their insurance business, they develop a variety of custom software systems. The business information system analyzed in this study implements damage prediction functionality and supports ca. 150 expert users in over ten countries. Table II illustrates the study object's main characteristics.

TABLE II. Study object: Characteristics.

Language	C#
Age (years)	10
Size (kLOC)	313
Size (#methods)	31,991
Use case documents	46

The system's usage was monitored for 414 days in its productive environment. We filtered out methods and types that only have a testing purpose and did not take external libraries into account. This leads to 11,034 unused methods and 20,957 used methods.

For validating our results, we interviewed the leading architect of the system (*system expert* in the remainder of the paper). Since he has been developing the system for 10 years, he has good knowledge about the system itself but also about the domain. Therefore, we argue that he is capable of estimating whether a feature contained in an use case document is unused based on method usage data.

B. Study Execution

First, we collect usage data, the software system's source code, and its use case documents. From the use case documents, we automatically extract the text.

Second, we present unused methods to the system expert and identify one use case expressing an unused feature manually for parameter estimation[9] (see Section III).

Third, we generate the ranking following our approach as described in Section III.

Fourth, we present the generated ranking to the system expert. He classifies the use case documents as *used* (features contained in the use case document are used completely or only a small part is not used), *partly unused* (large portions of the contained features are not used), and *unused* (the use case was never performed) according to usage data and his knowledge about the system.

C. Results

Our approach produced the ranking shown in Table III. This ranking was produced by using use case document UC2 for the calibration of LSI and we use it for answering our research questions.

TABLE III. Ranking, similarity score, and expert classification of use case documents.

Rank	Name	Score	Expert Classification	
1.	UC1	0.18	unused	
2.	UC2*	0.18	unused	*Used for calibration
3.	UC3	0.16	used	
4.	UC4	0.16	partly unused	
5.	UC5	0.12	partly unused	
6.-46.	...	$\leqslant 0.10$	used	

RQ1: We found two use case documents expressing unused features which were ranked highest by our approach (UC1 and UC2), and two use case documents, which expressed partly unused features (UC4 and UC5). However, UC3 was used according to the system expert. The use case documents ranked below the fifth rank sometimes contained small unused portions, but therein, the features contained in use case documents are described in a too coarse grained fashion (at the level of use case steps or whole flows) to make statements about usage.

RQ2: The top two use case documents in our ranking express unused features. Especially, UC1 was ranked higher than UC2, which was used for calibration and known to contain only unused features. Hence, we found one use case document expressing a completely unused feature. Furthermore, with detecting UC4 and UC5 and ranking them also high, the approach is capable of detecting use case documents expressing unused features. However, we ranked UC3, which describes used features, high, leading to a drop in precision.

[9] For LSI, we used 17 factors for singular value decomposition. The local weighting functions returns 1 if a term occurs in the document and 0 else. The global weighting function is inverse term document frequency.

Since our approach produces a ranking of use case documents rather than classifying use case documents by their feature usage, we calculate precision and recall depending on the number of use case documents (or the number of top ranks) we consider. Precision and recall dependent on the ranks are shown in Table IV. For our calculations of precision and recall, we classify use case documents describing features with large unused parts also as relevant (*true*), because they contain large regions that never have been performed and should be considered by the product owner when planning maintenance. Considering use case documents expressing partly unused features as relevant, our approach achieves an average precision (*AP*)[10] of 0.89, and taking only use case documents into account describing completely unused features, *AP* is 1, since our approach ranks both relevant use case documents highest.

TABLE IV. RESULTS: PRECISION AND RECALL OF THE APPROACH FOR GIVEN RANKS (USE CASES THAT EXPRESS PARTLY UNUSED FEATURES CONSIDERED RELEVANT).

Ranks	Precision	Recall
1	1.00	0.25
2	1.00	0.50
3	0.67	0.50
4	0.75	0.75
5	0.80	1.00
6	0.67	1.00

D. Discussion

RQ1: Unused features exist. Since these features may lead to unnecessary maintenance, we consider the problem addressed by our approach as relevant. When presenting the results to the system expert, he was not aware of the existence of unused feature, which was proven by usage data. Therefore, the reasons for the identified features being unused are unclear and demand further investigation.

However, we gained anecdotal insights into the reasons for the existence of unused features in other systems (reported by the system expert):

- Requirements were demanded and formulated, but never used, because workarounds that were easier to use than the actual system also fulfilled the task.
- Features were not completely implemented at the time of the inspection and were therefore not possible to perform.

RQ2: With good precision considering the highest use case documents in the produced ranking, and also good recall, the presented approach helps finding unused features. The system expert found the information helpful, since based on this he can direct maintenance actions aligned more along the users' needs.

[10]We calculate the average precision *AP* according to [20] by

$$AP = \frac{1}{R}\sum_{i=i}^{n} r_i \left(\frac{\sum_{j=1}^{i} r_j}{i}\right) = \frac{1}{R}\sum_{i=1}^{n}\left(P(i)\cdot r_i\right), \text{ where}$$

$$r_i = \begin{cases} 1 & \text{if document at rank } i \text{ is relevant} \\ 0 & \text{else} \end{cases}, \text{ and}$$

$P(i)$ is the precision at rank i, R is the total number of relevant documents, and $n = 46$, since we considered 46 use case documents. *AP* measures how many irrelevant documents are amongst the relevant documents in a ranking.

General: With the presented approach, we narrowed down the search space for use case documents expressing unused features from 46 (all) use case documents to 6 use case documents. According to the system expert, the remaining use case documents contained only features that *had* to be used by the actual users to do their daily business. Therefore, our approach reduces the effort that has to be spent to find unused features by giving hints to product owners.

The study object already was cleaned from known unused code and use case documents containing unused features in a refactoring phase before we monitored its usage for the study. Hence, our results only point out to use cases that have been overseen by experts in this previous phase. Thus, we expect more unused features in other systems that were not cleaned.

E. Threats to validity

The results presented might be flawed due to technical errors in the usage data collection. Due to the nature of our profiler that records method calls by registering to the just in time compile event, but also to the inline event of methods, it might record methods as used that were not used[11]. This could produce less unused methods, which leads to less unused features. However, this profiler was used for several years in the environment under consideration (see [3], [21], [22]) without significant or visible errors. Therefore, we are confident that the errors introduced are small, if existent.

In this study, we applied the usage data of one year. Even though different time spans might produce other results, we considered this appropriate. This assumption was confirmed by the system expert.

Furthermore, we might not have collected all relevant use case documents for the system, since these were scattered through the company's storage system. This would lead to possibly more unused features than we presented. However, this leads only to an underestimation of unused features. Additionally, we might also not have found all use case documents describing unused features, since the system expert might not be aware of all unused features. We did not find any methods that belonged to a feature that was not used except for the features expressed by the use case documents we identified.

As we conducted the preliminary study only on one system, generalizability of our results is threatened. Especially, choosing different parameters for the tracing algorithm might be necessary for replicating the study on other systems. To mitigate this threat, we presented our approach to parameter estimation and also estimated parameters using UC1. The configuration resulting from using UC1 rather than UC2 was the same as using UC2, and therefore we consider this threat as minor.

V. FUTURE WORK

Based on the insights we gained in our preliminary study, we are motivated to take further steps in the area of the presented research work. We divide our future work in short term tasks, next steps, and long term items, the future research questions.

[11]Such as inlined methods that were not executed.

Next steps: The validation we conducted in this study indicates the abilities of our approach. However, many questions regarding the correctness and limitations are still open. In future work, we plan to perform broader evaluations focusing on the applicability and benefit of our approach.

So far, our approach is uncovering links between source code and use case documents. However, other software engineering artifacts contain valuable information too and can help supporting development and maintenance of software. In future work, we plan to extend our approach to include design artifacts such as user stories or business rules and process artifacts such as change requests or bug tracking issues.

Being aware of unused feature implementation can help to direct maintenance tasks efficiently. In future work, we plan to enrich our methods to create a closer feedback loop to uncover unused features early and thereby reduce maintenance costs effectively from the beginning of the maintenance phase.

Future research questions: One question arising from our study is why there are features which have been specified once but are actually never used in the implemented software systems. This information can be valuable feedback for software engineering research to correct requirements elicitation methods and techniques to specify, design and implement just those functionalities which are actually needed.

Furthermore, once unused implementations have been detected it is still unclear how to proceed: the spectrum of options reaches from simply removing unused functionality up to adapting user training so users can optimize their work by using abandoned features. We need structured approaches to cope with unused functionality in software systems.

VI. Summary and Conclusions

Unused features often are unnecessary. Therefore, maintenance of unused features possibly leads to unnecessary costs. Unfortunately, information about the usage of a software system is collected by profilers on code level. This leads to the problem that product owners, that do not know the source code of their systems, are not able to establish a mapping between usage information and features. Therefore, we provide an approach to the question "How can we identify the unused features to prevent this unnecessary maintenance cost?".

We proposed an approach that bridges the gap between unused code and features using LSI for linking methods to use case documents as an approximation to features and to calculate a ranking that sorts use case documents by their likeliness of containing features that are not used by actual users of a software system. In a preliminary study we showed that our approach yields promising results: with the ranking produced by our approach, we found two use case documents containing completely unused features and two use case documents describing features with large unused parts. Furthermore, we presented results that indicated our approach performs well in terms of precision and recall.

Acknowledgments

The authors thank Christoph Frenzel and Veronika Bauer for their helpful comments on this work, and Karl-Heinz Prommer and Rudolf Vaas for their efforts interpreting the data.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A", and "Software Campus project ANSE, 01IS12057". The authors assume responsibility for the content.

References

[1] J. Johnson, "Roi, it's your job," Keynote at XP '02.

[2] E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. Prommer, "Feature Profiling for Evolving Systems," in *ICPC*, 2011.

[3] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Prommer, "How Much Does Unused Code Matter for Maintenance?" in *ICSE*, 2012.

[4] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process*. Addison-Wesley Reading, 1999.

[5] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, "Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data," *ACM SIGKDD Explor. Newsl.*, vol. 1, no. 2, 2000.

[6] M. El-Ramly and E. Stroulia, "Mining Software Usage Data," in *MSR*, 2004.

[7] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *Softw., IEEE*, vol. 23, no. 4, 2006.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, 2013.

[9] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software," *Empir. Softw. Eng.*, vol. 18, no. 2, 2013.

[10] N. Alhindawi, N. Dragan, M. Collard, and J. Maletic, "Improving Feature Location by Enhancing Source Code with Stereotypes," in *ICSM*, 2013.

[11] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Noninteractive Approach to Feature Location," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, 2006.

[12] J. Cleland-Huang and J. Guo, "Towards More Intelligent Trace Retrieval Algorithms," in *RAISE*, 2014.

[13] E. Ben Charrada, A. Koziolek, and M. Glinz, "Identifying Outdated Requirements Based on Source Code Changes," in *RE*, 2012.

[14] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *ICPC*, 2006.

[15] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering Traceability Links in Software Artifact Management Systems Using Information Retrieval Methods," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, 2007.

[16] M. Lormans and A. van Deursen, "Reconstructing Requirements Coverage Views from Design and Test Using Traceability Recovery via LSI," in *TEFSE*, 2005.

[17] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *ICPC*, 2010.

[18] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *J. Am. Soc. Inf. Sci. Technol.*, vol. 41, no. 6, 1990.

[19] O. Traub, S. Schechter, and M. D. Smith, "Ephemeral Instrumentation for Lightweight Program Profiling," School of engineering and Applied Sciences, Harvard University, Tech. Rep., 2000.

[20] A. Turpin and F. Scholer, "User performance versus precision measures for simple search tasks," in *SIGIR*, 2006.

[21] S. Eder, B. Hauptmann, M. Junker, R. Vaas, and K.-H. Prommer, "Selecting Manual Regression Test Cases Automatically Using Trace Link Recovery and Change Coverage," in *AST*, 2014.

[22] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer, "Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice," in *AST*, 2013.

APPENDIX E

Publication E [5]

Venue: RET@ICSE 2015
Acceptance rate: NA
Length: 7 pages
Type: Full paper
Reviewed: Peer reviewed

2015 IEEE/ACM 2nd International Workshop on Requirements Engineering and Testing

Configuring Latent Semantic Indexing for Requirements Tracing

Sebastian Eder, Henning Femmer, Benedikt Hauptmann, Maximilian Junker
Technische Universität München, Germany

***Abstract*—Latent Semantic Indexing (LSI) is an accepted technique for information retrieval that is used in requirements tracing to recover links between artifacts, e.g., between requirements documents and test cases. However, configuring LSI is difficult, because the number of possible configurations is huge. The configuration of LSI, which depends on the underlying dataset, greatly influences the accuracy of the results. Therefore, one of the key challenges for applying LSI is finding an appropriate configuration. Evaluating results for each configuration is time consuming, and therefore, automatically determining an appropriate configuration for LSI improves the applicability of LSI based methods.**

We propose a fully automated technique to determine appropriate configurations for LSI to recover links between requirements artifacts. We evaluate our technique on six sets of requirements artifacts from industry and academia and show that the configurations selected by our approach yield results that are almost as accurate as results from configurations based on a ground truth like known links or expert knowledge. Our approach improves the applicability of LSI in industry and academia, as researchers and practitioners do not need to determine appropriate configurations manually or provide a ground truth.

I. Introduction

When change requests have to be implemented for a software system, not only the software itself has to be changed, but also its documentation. It is often unclear which parts of a software system have to be changed and which other artifacts are impacted by the change. Furthermore, if one documentation artefact is changed, e.g., a use case, other artifacts also have to be changed, for example test cases. In both cases, we face the problem to decide which artifacts have to be changed in order to keep a consistent state of the software itself and the documentation as a whole.

In the fields of requirements tracing and trace link recovery, there are several methods of finding artifacts that are semantically connected to each other. Many of these approaches rely on Latent Semantic Indexing (LSI) [1] to retrieve links between artifacts that are semantically connected, e.g. [2]–[4]. However, LSI has to be configured differently for every artifact corpus [5].

Problem: There is a huge number of possible configurations for LSI [6] heavily influencing the resulting links and their accuracy [2], [3], [5], [7]–[10]. Existing approaches determine the configuration of LSI by using the configuration that reflects the links best that have to be created by system experts manually first [2]–[4], [11], [12]. Therefore, experts have to invest efforts to gain initial results, and may introduce incorrect links [13], which limits the applicability of LSI in requirements tracing.

Contribution: We propose a fully automated technique that estimates configurations for LSI. Our technique only relies on heuristic metrics that are calculated on the similarities between documents computed by LSI. In contrast to existing techniques, it does *not* rely on system experts' knowledge, existing known links between documents, or a ground truth, to recover semantic links between artifacts written in natural language. This yields the advantage, that experts do not have to invest efforts to gain just first results.

We evaluate our technique on six sets of requirements artifacts from industry and academia. We show that our approach determines configurations for LSI that yield, compared to the best possible configurations and to configurations chosen with a ground truth, accurate results for different kinds of artifacts.

Outline: The remainder of the work is structured as follows: First, we give an overview of related work and then introduce the most important terms and definitions. We then outline the procedure how we derived the proposed approach. We describe our pre-study, the approach and its validation. Afterwards, we describe topics for future work and summarize our work.

II. Related Work

Automated Requirements Tracing: Falessi et al. present seven principles that aim on improving the validity of studies that compare techniques for requirements tracing based on natural language processing [14]. We adhere, as far as possible, to these principles, however, we do not compare different techniques for requiremenents tracing (as we are considering just LSI) but present an approach for finding configurations for LSI that yield accurate results. Furthermore, Falessi et al. present a case study that compares different techniques to requirements tracing. As they state that LSI, not considering certain configurations, yields well results, we are motivated to also use this technique.

Requirements Tracing using LSI: A common approach to requirements tracing is to use LSI for generating links between semantically connected artifacts. Existing approaches use rules of thumb and best practices based on knowledge about correct links between artifacts, as also shown in Section VII, to select initial configurations of LSI, e.g., [2]–[4], [12], [15]. These works aim at the accuracy of recovered trace links, or at improving their accuracy. As optimal parameters for LSI vary

978-1-4673-7073-8/15 $31.00 © 2015 IEEE
DOI 10.1109/RET.2015.13

for every artifact corpus [5], we propose to select parameters more carefully in a systematic way, and to focus on finding an initial configuration for LSI to recover trace links.

There are several approaches for recovering links between semantically similar requirements artifacts. Some approaches use expert feedback on some generated links to adjust the whole set of links [3], [15]. However, expert feedback might worsen the results [13] and forces practitioners to invest efforts, which limits the applicability in practice. Therefore we do not consider expert feedback to adjust the results produced by LSI, but develop a fully automated approach for the estimation of appropriate configurations of LSI for a given artifact corpus without using a ground truth, like expert knowledge or known links.

Parameter Estimation for LSI: Kontostathis gives insights in factors that influence the performance of LSI itself [16]. However, the work focuses on understanding the inner mechanisms of LSI and does not consider the accuracy of the results. In contrast, the accuracy is in the center of our work, since it improves the applicability of LSI in requirements tracing. Ali et al. investigate the impacts of the inputs to tracing approaches when recovering links between source code and requirements artifacts [7]. They concentrate on experts' knowledge and on properties of the given artifact corpus. In contrast, we focus only on the configuration of LSI to produce accurate results independently of the actual artifact corpus or experts' feedback, because often, both are not changeable nor available. Binkley at el. give an approach to estimate parameters for Latent Dirichlet Allocation (LDA) [17], but not for LSI. We focus on LSI.

III. Background and Terms

Latent Semantic Indexing: LSI calculates the similarity between artifacts contained in an artifact corpus based on the content of the contained artifacts. Similarity is usually expressed by a value between -1 and 1, where a greater value means the compared artifacts are more similar[1]. LSI identifies words belonging to a common concept (e.g., 'car' and 'automobile'), enabling it to deal with synonyms to a certain extent.

LSI starts with creating a term-document matrix ($terms \times artifacts$) containing entries for each word in each artifact. The entries are calculated by a global weighting function w_g and a local weighting function w_l for each word, determining a value depending on the occurences of a term in the containing document (local) and on the ocurrences of a term in the whole artifact corpus (global). On this matrix, singular value decomposition is performed and the result is truncated to a smaller matrix, given the number of desired dimensions k (which can be interpreted as the number of concepts). This results in a reduced matrix where words are replaced by their concepts. This matrix represents every artifact as a vector in the space of concepts. The similarity of artifacts is calculated by comparing their vectors, e.g., cosine similarity[2].

We apply LSI to calculate similarity values between source artifacts and target artifacts. For example, if we want to know which use cases are impacted by change requests, the source artifacts represent the change requests and the target artifacts represent the use cases. Therefore, we generate similarity values between all change requests and all use cases. Afterwards, the target artifacts (use cases) are sorted by their similarity value to the source artifact (change requests, in descending order), resulting in a ranking of all target artifacts by similarity to the source artifact.

Configuration options of LSI: A configuration of LSI consists of three items: the local weighting function w_l, the global weighting function w_g, and the number of dimensions k. Table I shows common options for each item, and supplemantary variables. We use these options in the remainder of the work. All combinations of the options for w_l, w_g, and k are applicable, leading to $3 \cdot 3 \cdot (N - 4)$ combinations.

TABLE I
Configuration options of LSI considered

Variables	
df_i	Number of artifacts containing ith term
tf_{ij}	Number of occurences of ith term in jth artifact
gf_i	Number of occurences of ith term in all artifacts
p_{ij}	tf_{ij}/gf_i
N	Total number of artifacts
Options for the Local Weighting Function w_l	
Term Freq.	$tf = tf_{ij}$
Log Term Freq.	$ltf = log(tf_{ij} + 1)$
Binary	$bin = 1$
Options for the Global Weighting Function w_g	
Entropy	$H = 1 - \sum_j \frac{p_{ij}\ log\ p_{ij}}{log\ n}$
Inverse Doc. Freq.	$idf = log \frac{n}{1+df_i}$
Binary	$bin = 1$
Options for the Number of Dimensions k	
Range from 5 to N, considering all possibilities (step width is 1)	

Quality Measurement: To measure the quality of a configuration, we calculate the average precision (AP) [18] on the rankings for every source artifact and then take the mean over all these rankings, which results in the mean average precision (*MAP*) [6]. A *MAP* of 1 means for a ranking, that all relevant artifacts are ranked to the top of the list, without irrelevant artifacts in between[3]. We use *MAP* only for evaluating our approach (see Section VII), and to develop our approach (see Section V). The approach itself does not rely on *MAP*, since for this, it would need a ground truth.

[1] Can be scaled to the range between 0 and 1.

[2] We use only cosine similarity, since our pre-study (see Section V) showed that it outperforms other distance measures by large.

[3] *MAP* implicitely measures recall as well, since a greater number of irrelevant artifacts between relevant artifacts reduces the *MAP*.

Quality of Configurations: The best possible *MAP* among all configurations for one dataset is denoted as MAP_B, and the average over all configurations as MAP_A.The *MAP* produced by a selected configuration C is denoted as MAP_C.

As an acceptence criterion, we consider a configuration C as *appropriate*, if $MAP_C > MAP_B - 0.1$. This means, compared to the best configuration, precision@rank [10] drops by less than 0.1 in average over all possible ranks and over all source artifacts with an appropriate configuration. Therefore, we expect configurations that are *appropriate* to produce rankings that reflect actual semantic links almost as good as the best configuration does.

Heuristic metrics: The proposed approach aims at finding appropriate configurations of LSI by not using a ground truth or expert knowledge. However, it uses heuristic metrics that are calculated solely on the ranking produced by LSI with a certain configuration, and *not* on the ground truth. *MAP* cannot be such a heuristic metric, since it demands a ground truth, like a set of known links between artifacts, to be calculated.

IV. Study Overview

The goal of our study is to develop an approach to select appropriate configurations for LSI without using a ground truth in terms of expert knowledge or known links. We developed the approach in a structured way and in three steps, as illustrated in Figure 1. The next sections are organized along this structure.

In our first step, we conduct a pre study to build hypotheses about which heuristic metrics are suitable to determine appropriate configurations. We do this based on our experience gathered in earlier work [19], [20], and on the dataset MR0. Additionally, we test the hypotheses for validity to determine whether the heuristic metrics are suitable to determine appropriate configurations. This step is detailed in Section V.

The second step is developing a fully automated approach that is based on the hypotheses and heuristic metrics from the first step by operationalizing them, as explained in Section VI.

The last step is validating our approach by applying it to all datasets available. In this step, we determine whether the proposed approach is suitable for finding appropriate configurations for LSI without using a ground truth or expert knowledge. The approach will be described in detail in Section VI, and the validation well be described in Section VII.

We use the ground truth for the datasets only to test our hyptheses from the first step, to develop our approach and to validate the approach. The ground truth is not used for applying the approach.

V. Pre-Study

We summarize the first step of our study in this section, the pre-study: First, we develop our heuristic metrics and build hypotheses based on them. Second we test the hypotheses. The goal of the pre-study is to identify reliable heuristic ranking metrics (that do not depend on a ground truth) we can base our apporach on.

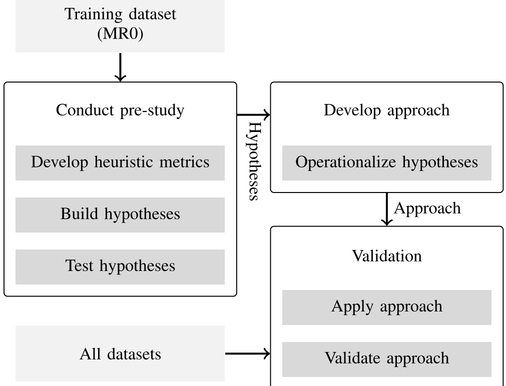


Fig. 1. Schematic overview of the studies. Light boxes are input artifacts, while dark boxes are steps in the approach. Arrows indicate data flow.

A. Heuristic Ranking Metrics

Our approach relies on two heuristic metrics that are calculated based on the rankings a configuration yields. We developed these metrics in a pre-study with the dataset MR0 (see Section VII) for which all correct links were documented by system experts. We use knowledge about correct links only for developing and evaluating our approach, whereas the approach itself does *not* rely on it.

We determined two heuristic metrics to select configurations from our experience: *pos*, and *range*. The following paragraph explains these metrics and the rationale behind them.

Metric *pos*: The position of the largest distance between the similarity values of two consecutive[4] artifacts in the ranking. *Rationale:* Often, only few target artifacts are actually linked to the source artifact. The lower *pos* is, the less artifacts are considered to be linked by LSI following the approach of Zhao et al. [21] that considers artifacts ranked above *pos* to be actually linked to the source artifact. Thus, we expect rankings to be better, if *pos* is lower.

Metric *range*: The difference of the highest and lowest similarity value in a ranking. *Rationale:* With a higher *range*, the ranking gets more expressive, since artifacts with higher similarity are ranked higher with more confidence, and low ranked artifacts have a more distinctive similarity.

To calculate *pos* and *range* for a configuration C, we take the mean of these metrics over the rankings C produced for all source artifacts.

B. Hypotheses

With our two heuristic metrics, we constitue two hypotheses. Since we measure the accuracy of the ranking produced by one configuration by its *MAP*, both hypotheses relate our heuristic metrics to *MAP*.

$\mathbf{H_{pos}}$: Rankings with low *pos*, exhibit a high *MAP*. So we expect configurations that produce rankings with a low *pos* to produce more accurate results.

[4]Rankings are sorted by similarity value in descending order (Section III).

$\mathbf{H_{range}}$: Rankings with high *range*, exhibit a high *MAP*. This hypothesis states that we expect configurations that yield rankings with a high *range* to produce more accurate results.

C. Test of Hypotheses

Figure 2 visualizes the correlation between *pos*, *range*, and *MAP* for the dataset MR0, where one dot is one configuration: Configurations with a high *MAP* exhibit a low *pos* and a high *range*. The visual impression is confirmed by the correlation of the two heuristic metrics to *MAP*: The Pearson correlation coefficient between *pos* and *MAP* is -0.84, consituting a strong correlation, and between *range* and *MAP* it is 0.87, also constituting a strong correlation. Both with a p-value that is greater than 95%.

Due to these correlations, we confirm our hypotheses. Therefore, we use these metrics to determine appropriate configurations. However, picking configurations with the highest *pos* or the highest *range* is not sufficient, since there are outliers, which do not yield optimal results, but exhibit a low *MAP*, as illustrated in Figure 2.

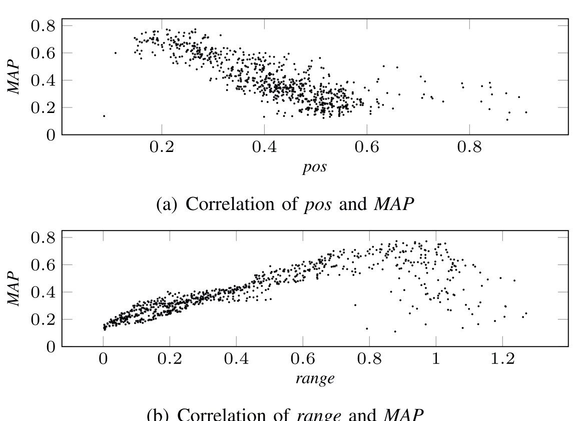


Fig. 2. Correlations of the metrics *pos* and *range* with *MAP* for the study object MR0. Every point represents one configuration.

VI. Approach

Due to the outliers shown in the last section, we develop a more sophisticated approach to determine proper configurations for LSI.

The input to the first step of our approach are the sets of source and target artifacts (the dataset). The output of the approach is one configuration. For all other steps, the inputs are the outputs of the previous step. Note that we select different configurations for every dataset. The approach is illustrated in Figure 3. Note that, as illustrated, the proposed approach does *not* use any kind of ground truth as input, but only the source and target artefacts.

① **Perform LSI** to compare every source artifact to every target artifact for all possible configurations.
Output: For every configuration: Rankings for each source artifact to all target artifacts sorted by similarity.

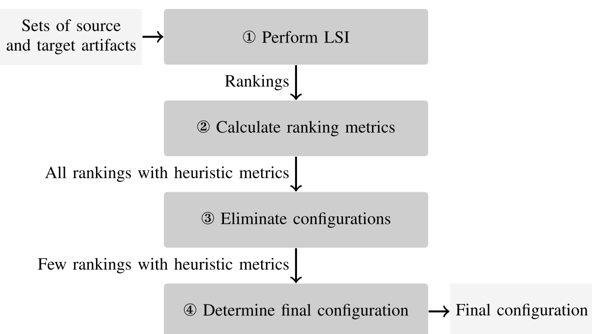


Fig. 3. Schematic illustration of the approach. Light boxes are input/output artifacts, while dark boxes are steps in the approach. Arrows indicate data flow between the steps.

② **Calculate ranking metrics** for the rankings produced by the previous step. For every configuration, we calculate *pos* and *range*.
Output: Ranking metrics for all configurations.

③ **Eliminate configurations** with a mean value of *pos* higher than the median or a mean value of *range* lower than the median, because we want a configuration to satisfy both criteria: low *pos* (only a small number of target artifacts to be linked to the source artifact), and high *range* (a high difference in similarity between highly and lowly ranked target artifacts). We repeat this step until the number of kept configurations is lower than ten[5]. The rationale behind this step is not to select appropriate configurations, but to eliminate the bad ones. We do this incrementally not to loose either configurations with a low *pos* or a high *range*.
Output: Few configurations.

④ **Determine the final configuration** by taking the configuration with the median of *pos* among the remaining configurations, because we do not want to select outliers in terms of *pos* or *range*. The reason for this becomes apparent in Figure 2: The best configurations in terms of *MAP* exhibit a high *range* and a low *pos*, but there are configurations, which are worse, that also fulfil these conditions.

VII. Validating Study

We conduct a case study to evaluate our approach by answering the following research questions. The goal of the validating study is to check whether the proposed approach selects appropriate configurations for LSI. We used knowledge about correct links between artifacts just for designing and evaluating our approach, but not as an input to it.

RQ1: Does the algorithm find appropriate configurations? By answering this question, we validate the ability of the proposed approach to find configurations that produce accurate trace links measured against the best possible configuration.

RQ2: How does the automated approach compare to approaches used in literature? Other approaches in literature

[5]This is an arbitrary choice, but yielded the best results for MR0.

use knowledge about the semantic similarity between the artifacts in their datasets to gain appropriate configurations. We compare our approach that does not take any knowledge about a (partial) set of correct links between artifacts, to the configurations used by other researchers.

A. Study Objects

We use six datasets with varying artifact types to evaluate our approach. Three datasets, MODIS, CM-1, and EasyClinic[6] were already used by other researchers to evaluate their approaches, and can therefore be used for answering RQ1 and RQ2. The other three datasets, MR0, MR1, and MR2, originate from the reinsurance company Munich Re and are confidential. Therefore, they can only be used for answering RQ1, since no other researchers could validate their approaches on them. The characeristics of the datasets are shown in Table II.

B. Study Execution

We execute the study along our approach. We use an own implementation of LSI in Java, which is optimized for running LSI with different configurations and was already used in prior work [19], [20]. We calculate ranking measures in R[7]. Also the elimination and selection of the final configuration is done in R. Running the analyses took 11 minutes on a computing server with 16 processing cores (2.0 GHz) and 64 GB of RAM.

C. Results

RQ1: Figure 4 shows the *MAP* resulting from the configuration C selected by our approach (MAP_C), the *MAP* resulting from the best possible configuration (MAP_B), and the average *MAP* over all configurations (MAP_A), which shows that it is not trivial to find an appropriate configuration. The difference from MAP_B to MAP_C lies below 0.1 in every case, and therefore, we consider all found configurations as *appropriate*, as explained in Section III.

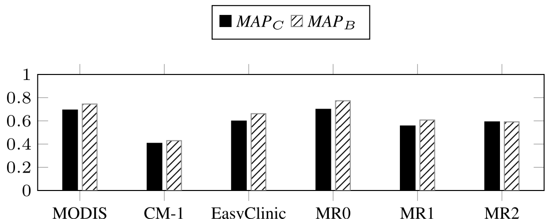


Fig. 4. *MAP* of the selected, best and average configuration for each dataset

Discussion: For all datasets we considered, the approach finds configurations that produce rankings with a *MAP* that is close to MAP_B. In the case of MR2, we find the optimal configuration. The difference of MAP_B to MAP_C lies below 0.1 in every case. This means, compared to the best configuration, precision@rank [10] drops by less than 0.1 in average over all possible ranks and over all source artifacts with the configuration selected by our approach. Therefore, the selected configuration produces rankings that reflect actual semantic links almost as accurate as the best configuration.

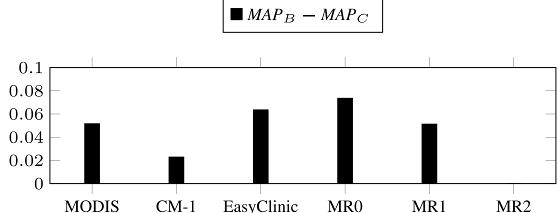


Fig. 5. Difference of *MAP* of the selected, and best configuration for each dataset. The higher the bars, the greater is the distance of the best possible configuration and the selected one.

Figure 6 shows the distribution of the *MAP* for the different study objects. The distributions show that finding a configuration that is that close to the best configuration is not trivial, since there are much more configurations with a *MAP* worse than the *MAP* of the selected configurations. The chance of selecting appropriate configurations according to our definition in Section III by random lies between 1% (MR1: 15 configurations out of 1422 are appropriate) and 23% (CM-1: 932 out of 4050 configurations are appropriate).

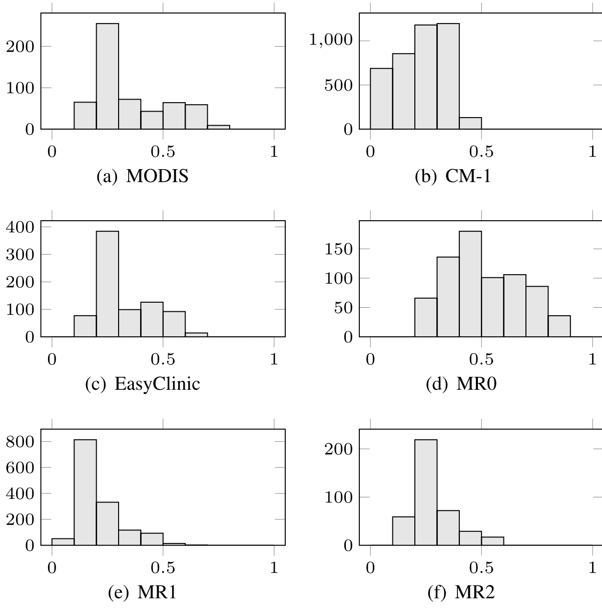


Fig. 6. Distribution of *MAP* over all configurations for the study objects

Thus, we conclude that the proposed approach finds configurations yielding accurate results, given that LSI is able to produce rankings with a desired quality.

RQ2: Table III shows the configurations from previous work [3], [4], [15] that were applied to our datasets and the

[6]These datasets, and others, can be acquired for study reproduction via CoEST at http://www.coest.org/index.php/resources/dat-sets

[7]http://www.r-project.org/

TABLE II
STUDY OBJECTS

Name	Source artifacts	Target artifacts	# possible Configurations	Source
MODIS	19 high-level requirements	49 low-level requirements	567	Industry (NASA)
CM-1	235 high-level requirements	220 low-level requirements	4,050	Industry (NASA)
EasyClinic	30 use cases	63 test cases	792	Academia
MR0	24 use cases	60 test cases	711	Industry (Munich Re)
MR1	135 defect reports	28 use cases	1,422	Industry (Munich Re)
MR2	28 change requests	21 use cases	396	Industry (Munich Re)

TABLE III
CONFIGURATIONS FOR ALL STUDY OBJETCS

Dataset	Configurations of other contributions					Selected		
	Paper	w_l	w_g	k (*C1*)	k (*C2*)	w_l	w_g	k
MODIS	[3], [15]	*tf*	*idf*	10	19	*ltf*	*bin*	7
CM-1	[3], [15]	*tf*	*idf*	100	200	*tf*	*idf*	66
EasyClinic	[4]	*ltf*	*H*	$0.1 \cdot N$	$0.2 \cdot N$	*tf*	*idf*	12

configurations selected by our approach. All three contributions propose two configurations *C1*, and *C2*. However, the configurations per contribution only vary in the number of dimensions k. In Figure 7, we illustrate the comparison of the configuration C selected by our approach, yielding MAP_C to the *MAP* of configurations used in previous work: MAP_{C1} and MAP_{C2}. In the case of CM-1 and EasyClinic, MAP_C is as high as the highest *MAP* achieved by the other configurations, and for MODIS, MAP_C lies between the two configurations proposed by the works we compare to.

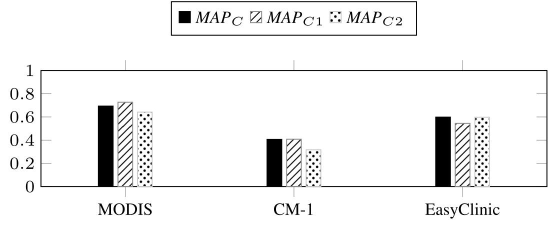


Fig. 7. *MAP* of the configuration selected by our approach (MAP_C) and the configurations proposed by other papers (MAP_{C1} and MAP_{C2})

Discussion: In all cases, our approach selected configurations that yield rankings with a *MAP* close to the configurations proposed by prior work. This means that our approach performs almost as well as approaches using knowledge about existing links between artifacts. We therefore conclude, that our approach is suitable for finding appropriate configurations. And in contrast to prior work, our approach does not rely on knowledge about existing links between artifacts. This shows the improvement of LSI's applicability by our approach.

D. General Discussion

The results show, that the proposed approach finds appropriate configurations among a great number of possible configurations (496 – 4050), where the number of appropriate configurations is small, without necessary knowledge about correct links between artifacts for several datasets. Therefore, it is suitable for estimating parameters for LSI.

The approach is computationally intensive, because LSI has to be performed on the datasets with all possible configurations. Analyzing the largest dataset (CM-1) took 7 minutes on a computing server with 16 processing cores (2.0 GHz) and 64 GB of RAM.

E. Threats to Validity

Even though we conducted the study on six datasets, we can still not generalize from the results. We adressed this threat by selecting study objects from industry containing heterogeneous artifacts from different companies.

We used manually developed lists of the correct links between artifacts that might contain errors. Three of the datasets (MODIS, CM-1, and EasyClinic) were used by other researchers in a plethora of studies and were examined for accuracy. Therefore, we consider this threat as minor for these datasets. For the datasets provided by Munich Re (MR0, MR1, and MR2), we performed manual inspections to mitigate this threat. As we could not find false or missing links, we consider this threat as minor for these datasets.

Measuring the *MAP* of the rankings produced by the configurations selected by our approach might not be suitable for all scenarios, since *MAP* captures whole rankings rather than just the first artifacts and thus implicates that the whole rankings are presented. Other approaches cut the rankings by thresholds of similarity values or after a fixed number of artifacts [14]. They measure, consequently, F-Measure to show the validity of the produced rankings. However, we examined the correlation between precision, recall, and *MAP*, and observed that *MAP* correlates strongly (in a statistical sense) with F-Measure.

VIII. FUTURE WORK

Since the approach is computationally intensive, we plan to employ more intelligent search strategies, e.g., genetic algorithms [6], to find configurations that match the given criteria in terms of the proposed metrics.

Furthermore, we only considered links between homogeneous sets of artifacts written in natural language. Therefore, we plan to evaluate our approach between more heterogeneous sets of artifacts like models or diagrams and source code.

As LSI is not the only approach for recovering semantic links between artifacts [22], we plan to apply similar heuristics for configurations of other tracing techniques. Examples are Latent Dirichlet Allocation (to compare to Binkley et al. [17]), the Jensen-Shannon Divergence, or the Vector Space Model.

Another research direction is, incorporating expert feedback as proposed in prior contributions, e.g., [3], [15]. We expect, as the initial rankings are produced automatically by our approach, the applicability of these approaches to improve.

IX. Summary and Conclusion

Changing requirements of a software system lead to changes not only to the software itself, but also to its documentation and tests, e.g., if a use case is adapted, we also have to update test cases. Therefore, we have to decide which artifacts have to be changed to keep a *consistent state* of the software itself and the documentation corpus.

Latent Semantic Indexing (LSI) is a common technique for requirements tracing, to recover links between artifacts, e.g., between requirements documents and test cases. However, configuring LSI is difficult, because the number of possible configurations is huge. The configuration of LSI, which depends on the underlying dataset, greatly influences the accuracy of the results. Therefore, one of the key challenges in applying LSI-based methods is finding an appropriate configuration producing accurate results.

We presented an approach to find configurations for LSI used in requirements tracing automatically. In contrast to other approaches, our approach does not rely on any kind of ground truth like expert knowledge or existing links between artifacts. The approach only uses two heuristic measures, *pos* and *range* that are calculated solely on the results LSI produces. Therefore, the only input to our approach are the documents for which traceability links have to be recovered.

We furthermore showed in a case study considering six objects from industry and academia that our approach finds appropriate configurations among hundreds or thousands of possible configurations. The found configurations produce rankings with similar accuracy (in terms of Mean Average Precision), as configurations chosen manually by other researchers with knowledge about existing links that had to be created manually beforehand.

The results we gained in our study indicate that our approach is suitable for automatically selecting appropriate configurations for LSI: The configurations selected are, in terms of accuracy, close to configurations selected with a ground truth. This improves the applicability of LSI in requirements tracing in research and practice, since no ground truth has to be established for determining an appropriate configuration.

Acknowledgment

The authors would like to thank Daniela Steidl, Veronika Bauer, Jonas Eckhardt, Daniel Méndez Fernández, and Andreas Vogelsang for their helpful comments on this work.

References

[1] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JASIST*, vol. 41, no. 6, 1990.

[2] M. Lormans and A. van Deursen, "Can LSI help reconstructing requirements traceability in design and test?" in *CSMR*, 2006.

[3] S. K. Sundaram, J. H. Hayes, and A. Dekhtyar, "Baselines in requirements tracing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, 2005.

[4] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *TOSEM*, vol. 16, no. 4, 2007.

[5] R. B. Bradford, "An empirical study of required dimensionality for large-scale latent semantic indexing applications," in *CIKM*, 2008.

[6] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang, "Improving trace accuracy through data-driven configuration and composition of tracing features," in *ESEC/FSE*, 2013.

[7] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, "Factors impacting the inputs of traceability recovery approaches," in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer London, 2012.

[8] A. Garron and A. Kontostathis, "Applying latent semantic indexing on the trec 2010 legal dataset," in *TREC*, 2010.

[9] G. Bavota, A. De Lucia, R. Oliveto, A. Panichella, F. Ricci, and G. Tortora, "The role of artefact corpus in LSI-based traceability recovery," in *TEFSE*, 2013.

[10] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," in *ICPC*, 2008.

[11] A. Kontostathis, "Essential dimensions of latent semantic indexing (LSI)," in *HICSS*, 2007.

[12] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer, "Did we test our changes? assessing alignment between tests and development in practice," in *AST*, 2013.

[13] D. Cuddeback, A. Dekhtyar, and J. Hayes, "Automated requirements traceability: The study of human analysts," in *RE*, 2010.

[14] D. Falessi, G. Cantone, and G. Canfora, "Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, 2013.

[15] J. Hayes, A. Dekhtyar, and S. Sundaram, "Advancing candidate link generation for requirements tracing: the study of methods," *IEEE Trans. Softw. Eng.*, vol. 32, no. 1, 2006.

[16] A. Kontostathis and W. M. Pottenger, "A framework for understanding latent semantic indexing (LSI) performance," *Information Processing & Management*, vol. 42, no. 1, 2006, formal Methods for Information Retrieval.

[17] D. Binkley, D. Heinz, D. J. Lawrie, and J. Overfelt, "Understanding LDA in source code analysis," in *ICPC*, 2014.

[18] S. Robertson, "A new interpretation of average precision," in *SIGIR*, 2008.

[19] S. Eder, B. Hauptmann, M. Junker, R. Vaas, and K.-H. Prommer, "Selecting manual regression test cases automatically using trace link recovery and change coverage," in *AST*, 2014.

[20] S. Eder, H. Femmer, B. Hauptmann, and M. Junker, "Which features do my users (not) use?" in *ICSME*, 2014.

[21] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *TOSEM*, vol. 15, no. 2, 2006.

[22] M. Borg and P. Runeson, "IR in software traceability: From a bird's eye view," in *ESEM*, 2013.

APPENDIX F

Reprint Permission for Publication A [1]

Title: How much does unused code matter for maintenance?

Conference Proceedings: 2012 34th International Conference on Software Engineering (ICSE)

Author: Sebastian Eder; Maximilian Junker; Elmar Jürgens; Benedikt Hauptmann; Rudolf Vaas; Karl-Heinz Prommer

Publisher: IEEE

Date: 2-9 June 2012

Copyright © 2012, IEEE

LOGIN

If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already **a RightsLink user** or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

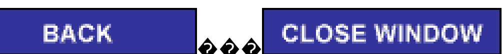
Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line � 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line � [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author�s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: � [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.



Copyright © 2016 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

APPENDIX G

Reprint Permission for Publication B [2]

Title: Did we test our changes? Assessing alignment between tests and development in practice

Conference Proceedings: Automation of Software Test (AST), 2013 8th International Workshop on

Author: Sebastian Eder; Benedikt Hauptmann; Maximilian Junker; Elmar Juergens; Rudolf Vaas; Karl-Heinz Prommer

Publisher: IEEE

Date: 18-19 May 2013

Copyright © 2013, IEEE

LOGIN

If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already **a RightsLink user** or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line � 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line � [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author�s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: � [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2016 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

APPENDIX H

Reprint Permission for Publication C [3]

ASSOCIATION FOR COMPUTING MACHINERY, INC. LICENSE TERMS AND CONDITIONS

Jun 23, 2016

This Agreement between Sebastian Eder ("You") and Association for Computing Machinery, Inc. ("Association for Computing Machinery, Inc.") consists of your license details and the terms and conditions provided by Association for Computing Machinery, Inc. and Copyright Clearance Center.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	3894600372851
License date	Jun 23, 2016
Licensed Content Publisher	Association for Computing Machinery, Inc.
Licensed Content Publication	Proceedings
Licensed Content Title	Selecting manual regression test cases automatically using trace link recovery and change coverage
Licensed Content Author	Sebastian Eder, et al
Licensed Content Date	May 31, 2014
Type of Use	Thesis/Dissertation
Requestor type	Author of this ACM article
Is reuse in the author's own new work?	Yes
Format	Print and electronic
Portion	Full article
Will you be translating?	No
Order reference number	
Title of your thesis/dissertation	Exploiting Execution Profiles in Software Maintenance and Test
Expected completion date	Jul 2016
Estimated size (pages)	155
Requestor Location	Sebastian Eder Boltzmannstr. 3 Garching, 85748 Germany Attn: Sebastian Eder
Billing Type	Credit Card
Credit card info	Visa ending in 2838
Credit card expiration	02/2021
Total	7.05 EUR
Terms and Conditions	

Rightslink Terms and Conditions for ACM Material

1. The publisher of this copyrighted material is Association for Computing Machinery, Inc. (ACM). By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any

time at).
2. ACM reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.
3. ACM hereby grants to licensee a non-exclusive license to use or republish this ACM-copyrighted material* in secondary works (especially for commercial distribution) with the stipulation that consent of the lead author has been obtained independently. Unless otherwise stipulated in a license, grants are for one-time use in a single edition of the work, only with a maximum distribution equal to the number that you identified in the licensing process. Any additional form of republication must be specified according to the terms included at the time of licensing.
*Please note that ACM cannot grant republication or distribution licenses for embedded third-party material. You must confirm the ownership of figures, drawings and artwork prior to use.
4. Any form of republication or redistribution must be used within 180 days from the date stated on the license and any electronic posting is limited to a period of six months unless an extended term is selected during the licensing process. Separate subsidiary and subsequent republication licenses must be purchased to redistribute copyrighted material on an extranet. These licenses may be exercised anywhere in the world.
5. Licensee may not alter or modify the material in any manner (except that you may use, within the scope of the license granted, one or more excerpts from the copyrighted material, provided that the process of excerpting does not alter the meaning of the material or in any way reflect negatively on the publisher or any writer of the material).
6. Licensee must include the following copyright and permission notice in connection with any reproduction of the licensed material: "[Citation] © YEAR Association for Computing Machinery, Inc. Reprinted by permission." Include the article DOI as a link to the definitive version in the ACM Digital Library. Example: Charles, L. "How to Improve Digital Rights Management," Communications of the ACM, Vol. 51:12, © 2008 ACM, Inc. http://doi.acm.org/10.1145/nnnnnn.nnnnnn (where nnnnnn.nnnnnn is replaced by the actual number).
7. Translation of the material in any language requires an explicit license identified during the licensing process. Due to the error-prone nature of language translations, Licensee must include the following copyright and permission notice and disclaimer in connection with any reproduction of the licensed material in translation: "This translation is a derivative of ACM-copyrighted material. ACM did not prepare this translation and does not guarantee that it is an accurate copy of the originally published work. The original intellectual property contained in this work remains the property of ACM."
8. You may exercise the rights licensed immediately upon issuance of the license at the end of the licensing transaction, provided that you have disclosed complete and accurate details of your proposed use. No license is finally effective unless and until full payment is received from you (either by CCC or ACM) as provided in CCC's Billing and Payment terms and conditions.
9. If full payment is not received within 90 days from the grant of license transaction, then any license preliminarily granted shall be deemed automatically revoked and shall be void as if never granted. Further, in the event that you breach any of these terms and conditions or any of CCC's Billing and Payment terms and conditions, the license is automatically revoked and shall be void as if never granted.
10. Use of materials as described in a revoked license, as well as any use of the materials beyond the scope of an unrevoked license, may constitute copyright infringement and publisher reserves the right to take any and all action to protect its copyright in the materials.
11. ACM makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.
12. You hereby indemnify and agree to hold harmless ACM and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.
13. This license is personal to the requestor and may not be sublicensed, assigned, or

transferred by you to any other person without publisher's written permission.
14. This license may not be amended except in a writing signed by both parties (or, in the case of ACM, by CCC on its behalf).
15. ACM hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and ACM (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.
16. This license transaction shall be governed by and construed in accordance with the laws of New York State. You hereby agree to submit to the jurisdiction of the federal and state courts located in New York for purposes of resolving any disputes that may arise in connection with this licensing transaction.
17. There are additional terms and conditions, established by Copyright Clearance Center, Inc. ("CCC") as the administrator of this licensing service that relate to billing and payment for licenses provided through this service. Those terms and conditions apply to each transaction as if they were restated here. As a user of this service, you agreed to those terms and conditions at the time that you established your account, and you may see them again at any time at http://myaccount.copyright.com
18. Thesis/Dissertation: This type of use requires only the minimum administrative fee. It is not a fee for permission. Further reuse of ACM content, by ProQuest/UMI or other document delivery providers, or in republication requires a separate permission license and fee. Commercial resellers of your dissertation containing this article must acquire a separate license.
Special Terms:

Questions? customercare@copyright.com or +1-855-239-3415 (toll free in the US) or +1-978-646-2777.

APPENDIX I

Reprint Permission for Publication D [4]

Title: Which Features Do My Users (Not) Use?

Conference Proceedings: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on

Author: Sebastian Eder; Henning Femmer; Benedikt Hauptmann; Maximilian Junker

Publisher: IEEE

Date: Sept. 29 2014-Oct. 3 2014

Copyright © 2014, IEEE

LOGIN

If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already **a RightsLink user** or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line � 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line � [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author�s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: � [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2016 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

APPENDIX J

Reprint Permission for Publication E [5]

Title: Configuring Latent Semantic Indexing for Requirements Tracing

Conference Proceedings: Requirements Engineering and Testing (RET), 2015 IEEE/ACM 2nd International Workshop on

Author: Sebastian Eder; Henning Femmer; Benedikt Hauptmann; Maximilian Junker

Publisher: IEEE

Date: 18-18 May 2015

Copyright © 2015, IEEE

LOGIN

If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already **a RightsLink user** or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

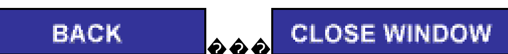
Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line � 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line � [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author�s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: � [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.



Copyright © 2016 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com