

TEST SUITE COMPREHENSION

Understanding Tests of Modular and
Dynamic Systems



DR. MICHAELA GREILER

Test Suite Comprehension

for modular and dynamic systems

Test Suite Comprehension

for modular and dynamic systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 22 april 2013 om 12.30
uur
door

Michaela Simona GREILER

Diplom-Ingenieur - Alpen-Adria University Klagenfurt
geboren te Klagenfurt, Austria.

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. A. van Deursen

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. Arie van Deursen	Technische Universiteit Delft, promotor
Prof. dr. Hausi Mueller	University of Victoria, Canada
Prof. dr. Serge Demeyer	University of Antwerp, Belgium
Prof. dr. Erik Meijer	Technische Universiteit Delft
Prof. dr. Koen Bertels	Technische Universiteit Delft
Prof. dr. Frances Brazier	Technische Universiteit Delft
Dr. Hans-Gerhard Gross	Technische Universiteit Delft



The work in this thesis has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was financially supported by the Netherlands Organization for Scientific Research (NWO)/Jacquard project 638.001.209, *AR-TOSC: Automated Runtime Testability of SOA Composites*.

ISBN 978-90-8891-603-8

Copyright © 2013 by M. Greiler

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the prior permission of the author.

Author email: michaela.greiler@gmail.com

Acknowledgements

“Let us be grateful to the people who make us happy; they are the charming gardeners who make our souls blossom.” (Marcel Proust)

I would like to express my deep gratitude to Arie van Deursen, my research supervisor and promotor, for his professional guidance and valuable critiques throughout my research. He helped me to develop skills needed to be successful in highly competitive environments. I am proud that I have been one of his students. Further, I want to thank Andy Zaidman for our fruitful collaboration. It was a pleasure to work with him. Special thanks should be given to Margaret-Anne Storey, who has not only been a mentor but also became a friend. I am grateful for her guidance, advice and the wonderful time I had while visiting her research lab. I would also like to extend my thanks to all my colleagues that contributed to make this department a great work environment. I would like to thank Alie Mesbah for sharing anecdotes of his PhD experience, Felienne Hermanns for helping me with the Dutch propositions and, especially, Erik Meijer for his advice on choosing my next professional adventure and his effort to support me on my way. I would also like to thank Gerd Gross for involving me in this research project, and the committee for their valuable feedback.

This research would not have been possible without people that always raise a smile on my face and make me happy. On that note, I would like to thank my family and friends for their support during the last four years. A special thank you goes to my dear friends Joolie, Jamie, Nadja and Geisa, who managed to make the PhD experience a good one, even during tough times. Further, I would like to thank my friend Robert for his constant support and his never-ending belief in me. He is my oasis of calm. I thank my mum for always having an open door and an open ear. The times my mum or my sister Katja came to visit me made me feel at home even in a foreign country. Finally, I thank my granny for watching out for me – not only when passing by in an air plane.

Thank you all for your support and love.

Delft, April 2013

Michaela Greiler

Danksagung

“Lasst uns dankbar sein gegenüber Leuten, die uns glücklich machen. Sie sind die liebenswerten Gärtner, die unsere Seele zum Blühen bringen.” (Marcel Proust)

Großen Dank möchte ich meinem Doktorvater Arie van Deursen für seine professionelle Betreuung und seine wertvolle Kritik während meines Doktorats aussprechen. Durch ihn lernte ich mich im Wettbewerb zu behaupten. Ich bin stolz, dass ich eine seiner Studentinnen war und von ihm lernen durfte. Ich möchte mich auch bei Andy Zaidman für unsere gute und erfolgreiche Zusammenarbeit bedanken. Im Speziellen möchte ich mich bei Margaret-Anne Storey bedanken. Sie war nicht nur ein Mentorin für mich sondern wurde auch zu einer Freundin. Ich danke ihr für die Betreuung und die schöne Zeit, die ich in ihrem Forschungsteam hatte. Ferner möchte ich auch all meinen Kollegen danken, die zu der tollen Arbeitsumgebung in dieser Gruppe beigetragen haben. Im Speziellen möchte ich Alie Mesbah, Feliene Hermans und Gerd Gross danken. Ein großer Dank geht an Erik Meijer für seinen Rat bei der Wahl meines nächsten beruflichen Abenteuers und seiner Hilfe bei der Realisierung. Ich danke auch dem Komitee für das wertvolle Feedback.

Ohne die Unterstützung meiner Familie und meiner Freunde wäre diese Arbeit wahrscheinlich nicht möglich gewesen. Sie sind die Menschen, die mich immer zum Lachen gebracht haben und die Sonne in meinem Herzen scheinen ließen. Ein großer Dank geht an meine lieben Freunde Joolie, Jamie, Nadja und Geisa. Sie konnten mir auch in harten Zeiten den Tag versüßen. Der größte Dank geht an meinen Freund Robert, der ständig für mich da war und immer an mich geglaubt hat. Er ist mein Fels in der Brandung. Ich danke meiner Mama, dass sie immer eine offene Tür und ein offenes Ohr für mich hat. Wenn sie oder meine Schwester Katja bei mir auf Besuch waren fühlte ich mich zu Hause – auch in einem noch so fremden und fernen Land. Ich danke auch meiner Omi, da ich weiß, dass sie im Gedanken immer bei mir ist und auf mich Acht gibt – nicht nur wenn ich gerade in einem Flugzeug vorbeifliege.

Ich danke euch allen für eure Unterstützung und eure Liebe.

Delft, April 2013

Michaela Greiler



Contents

Acknowledgements	i
1 Introduction	1
1.1 Software Testing	1
1.1.1 Testing dynamic and modular Software Systems	2
1.1.2 Test Suite Comprehension	2
1.2 Research Questions	4
1.3 Research Methods	5
1.4 Research Overview	6
1.5 Related Work	8
1.6 Origin of papers	9
2 Test Confessions: A Study of Testing Practices for Plug-In Systems	11
2.1 Introduction	12
2.2 Plug-in Systems: Capabilities and Challenges	13
2.3 Experimental Design	14
2.3.1 The Eclipse Plug-In Architecture	14
2.3.2 Research Questions	15
2.3.3 Research Method	15
2.3.4 Participant Selection	16
2.3.5 Presentation of Our Findings	17
2.4 Testing Practices	17
2.4.1 Open Versus Closed Development Setting	17
2.4.2 Test Responsibilities	18
2.4.3 Unit Tests	18
2.4.4 Beyond Unit Testing	19

2.5	Plug-In Specific Integration Testing	20
2.5.1	The Role of PDE Tests	20
2.5.2	Plug-In Characteristics	20
2.5.3	Testing Cross-Product Integration	21
2.5.4	Testing Platform and Dependency Versions	22
2.6	Barriers for Adopting Plug-In Specific Integration Testing Practices	24
2.7	Compensation Strategies	25
2.7.1	Self-Hosting of Projects	25
2.7.2	User Involvement	26
2.7.3	Developer Involvement	26
2.7.4	Openness – A Prerequisite for Participation	26
2.8	Discussion	27
2.8.1	Improving Plug-In Testing	28
2.8.2	Open Versus Closed Source	28
2.9	Credibility and Limitations	29
2.9.1	Credibility	29
2.9.2	Beyond Eclipse	29
2.9.3	Beyond the People	30
2.10	Related Work	30
2.11	Concluding Remarks	31
3	What your Plug-in Test Suites Really Test: An Integration Perspective on Test Suite Understanding	33
3.1	Introduction	34
3.2	Background: Modularization in Eclipse	35
3.3	Information Needs	37
3.3.1	Set-up Interviews	38
3.3.2	Summary: Eclipse Testing Practices	39
3.3.3	Test Suite Understanding Needs	40
3.4	Models for Understanding Plug-in Test Suites	44
3.4.1	The Plug-in Modularization View	45
3.4.2	Extension Initialization View	46
3.4.3	Extension Usage View	49
3.4.4	Service Usage View	52
3.4.5	The Test Suite Modularization View	54
3.5	Implementation and Tool Architecture	55
3.6	Evaluation	59
3.6.1	The Subject Systems	59
3.6.2	RQ1: Applicability and Information Needs	60
3.6.3	RQ2: Scalability	68
3.6.4	RQ3: Accuracy	70

3.7	Discussion and Future Work	71
3.7.1	User Study	71
3.7.2	Limitations	72
3.7.3	Recommendations	72
3.7.4	Threats to validity	73
3.8	Related Work	73
3.9	Concluding Remarks	75
4	Measuring Test Case Similarity to Support Test Suite Understanding	77
4.1	Introduction	77
4.2	Tracing and Trace Reduction	79
4.2.1	Tracing Test Executions	79
4.2.2	Handling mocks and stubs	79
4.2.3	Trace reduction	80
4.3	Determining Similarity Measurements	81
4.3.1	Relevancy support based on occurrence	81
4.3.2	Implementation	81
4.4	Set-Up for Case studies	82
4.4.1	Research Questions	82
4.4.2	Technique customization	82
4.5	Case Study I: JPacman	83
4.5.1	Obtaining the Conceptual Mapping	83
4.5.2	RQ1: Comparison to Conceptual Mapping	84
4.5.3	RQ4: Performance Characteristics	87
4.6	Case Study II: Auction Sniper	87
4.6.1	Obtaining an Initial Understanding	88
4.6.2	RQ2: Suitability of Measurements for Understanding Test Relations	89
4.6.3	RQ3: Handling Mocking	90
4.7	Discussion	91
4.8	Related Work	93
4.9	Conclusion	94
5	Automated Detection of Test Fixture Strategies and Smells	95
5.1	Introduction	96
5.2	Test Smells	97
5.3	Analysis of Fixture Usage	99
5.3.1	Fact Extraction	99
5.3.2	Analysis	100
5.3.3	Presentation	103
5.4	Implementation and Tool Architecture	104

5.5	Experimental Design	104
5.5.1	Research Questions	104
5.5.2	Case Studies	105
5.5.3	Interviews and Questionnaire	105
5.6	Evaluation	106
5.6.1	RQ1: What do the structure and organization of test fixture look like in practice?	106
5.6.2	RQ2: Do fixture related test smells occur in practice?	107
5.6.3	RQ3: Do developers recognize these test smells as potential problems?	108
5.6.4	RQ4: Does a fixture analysis technique help developers to understand and adjust fixture management strategies?	110
5.7	Discussion and Threats to Validity	111
5.8	Related Work	112
5.9	Concluding Remarks	113
6	Strategies for Avoiding Text Fixture Smells During Software Evolution	115
6.1	Introduction	115
6.2	Test Smells	117
6.3	Experimental Setting	119
6.3.1	Research Questions	119
6.3.2	Case Studies	119
6.4	Analysis of Fixture Smell Evolution	120
6.4.1	TestEvoHound	120
6.4.2	Measurements to Answer the Research Questions	121
6.5	Investigation of Test Fixture Smell Evolution	122
6.5.1	Evolution of Test Fixtures	122
6.5.2	Discovery of Test Fixture Smell Trends	124
6.5.3	Dispersion of Test Fixture Smells	124
6.5.4	Development of Test Fixture Smells	127
6.5.5	Fluctuations in Test Fixture Smells	130
6.5.6	Test Fixture Smell Resolution	133
6.6	Discussion	134
6.6.1	Findings	134
6.6.2	Implications for Automated Test Fixture Smell Detection	134
6.6.3	Strategies and Recommendations	135
6.6.4	Threats to Validity	135
6.7	Related Work	136
6.8	Conclusion	136

7 Conclusion	139
7.1 Contributions	139
7.2 Research Questions Revisited	140
7.3 Open Issues and Future Work	144
Bibliography	147
A Appendix: Grounded Theory Study	155
A.1 Resulting Collection of Codes	155
A.1.1 Category 1. <i>Practices</i>	155
A.1.2 Category 2. <i>Plug-in Specific Integration Testing</i>	157
A.1.3 Category 3. <i>Test Barriers</i>	158
A.1.4 Category 4. <i>Compensation Strategies</i>	159
A.2 Key Quotations	161
Index of Codes to Participants	182
Zusammenfassung	185
Samenvatting	187
Curriculum Vitae	189

Introduction

Test suite comprehension became more difficult over the last decade as test suites have grown substantially. Especially for modular and dynamic systems, where the system functionality can change at *runtime* testing is aggravated. In addition to enabling dynamic reconfigurations, modular systems are conglomerates of several sub-systems, with different owners. Those two characteristics aggravate testing and test suite comprehension (e.g., understanding which configurations and combinations of the system have been tested) and therefore such systems require further investigation.

1.1 Software Testing

At its heart, software testing can be defined as the execution of code using combinations of input and state selected to reveal bugs (Binder, 1999). This quite plain sounding activity constitutes an important part of the software development process. It sheds light on the quality and reliability of the software product regarding functional and non-functional properties, and directs improvement activities. Manual testing of software systems can be a tedious and time consuming task. Therefore, efforts have been made to automate the testing process by using or creating another software system to control the execution of tests, including the comparison of actual to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions. Fully automated tests run without human interaction, which allows frequent and continuous test execution throughout the software process (i.e., continuous testing). Continuous testing brings new advantages, as it can be seen as a “safety net” during refactoring and maintenance activities, increasing the confidence in the proper functionality of the system, or alerting in case new bugs have been introduced.

Already Beizer (1990) stated that “the act of designing tests is one of the most

effective bug preventers known”.

This mind set, manifested in the test-driven development movement, directed our research interest towards human-written automated test activities. As testing related activities may consume a large part of the effort required during software development, the goal of this dissertation is to support developers during these testing activities. Therefore, we investigate the current testing practices, reveal testing challenges and design and implement several tools supporting developers to overcome some of these challenges.

1.1.1 Testing dynamic and modular Software Systems

Modular and dynamic software systems have been of interest during this dissertation, as they exhibit the ability to be changed and enhanced at *runtime*, by changing, adding, or removing parts of the system. This ability to change at *runtime* is of interest as it aggravates testing activities because the final state of the system and its functionality may be unknown or uncertain during development as the system can change after deployment.

In this dissertation, plug-in-based systems have been investigated as they are one type of a modular and dynamic system. Plug-in-based systems rely on plug-in components to extend a base system (Marquardt, 1999; Voelter, 2001; Shavor et al., 2005). As argued by Marquardt (1999), the base system can be delivered almost “nakedly”, while most user value is added by plug-ins that are developed separately, which can extend the existing applications without need for change. In more sophisticated plug-in architectures, plug-ins can build upon each other, allowing new products to be assembled in many different ways. The ability to add and remove plug-ins and alter the system at runtime raises concerns about the compatibility of plug-ins (Pohl and Metzger, 2006; Rehmand et al., 2007; Weyuker, 1998). Incompatibility, be it because of combinations of plug-ins or versions, can be hard to strive against, and may restrict the benefits plug-in systems offer.

In particular, integration testing, where the scope of testing is a complete system or subsystem of software components (e.g., plug-ins), is crucial to detect problems between components and to reveal compatibility issues. One problem of integration testing is that testing every possible combination of plug-ins, versions, operating systems and third party libraries is often sheerly impossible, and already considering certain combinations can lead to a combinatorial explosion of possible tests. The number of combinations that are possible to test, considering limited time and resources, can be increased by increasing the degree of test automation in contrast to performing manual testing. The main disadvantage of test automation is the increase in source code, i.e., test code.

1.1.2 Test Suite Comprehension

Automated test suites of modular and dynamic software systems, such as plug-in systems, can comprise a substantial amount of test code (Zaidman et al., 2011).

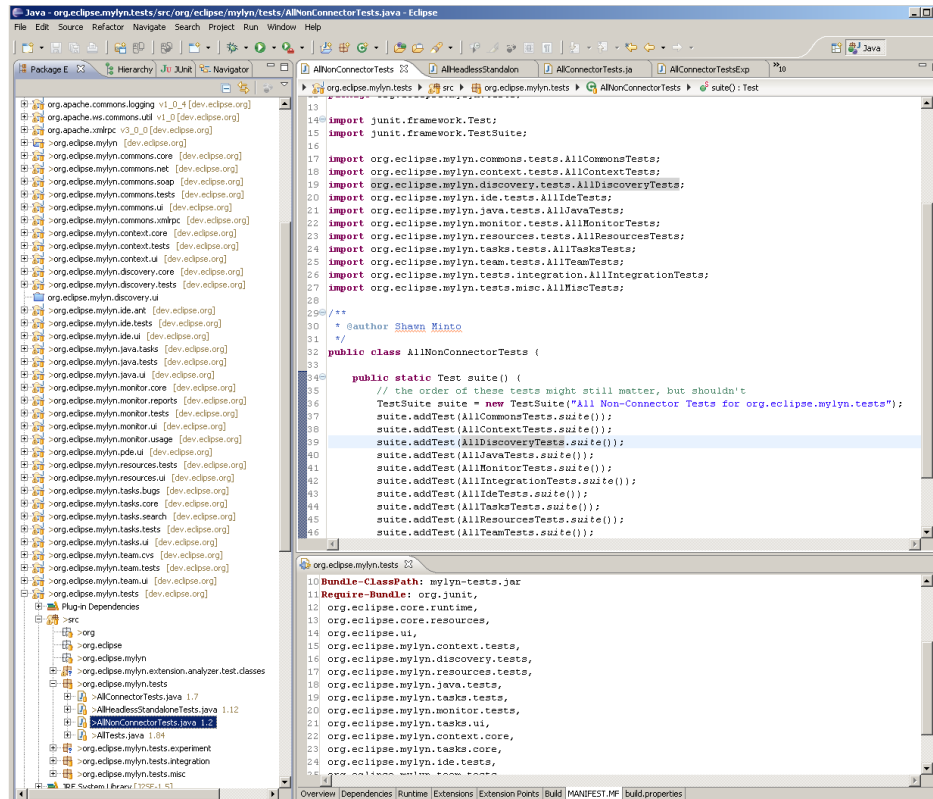


Figure 1.1: Screenshot of the Eclipse IDE showing the test code of the well-known plug-in system Mylyn

Like production code, test code needs to be maintained and adjusted upon changes to production code or requirements, which can become very costly (Greiler et al., 2010; Meszaros, 2007; Van Rompaey et al., 2007). Figure 1.1 shows a screenshot of the Eclipse IDE showing the code and test code of the well-known plug-in system Mylyn,¹ which contains tens of thousands of lines of code, comprised in several dozens of plug-in projects.

In order to be able to adjust and maintain software test code, the maintainer has to sufficiently understand the code. This process is known as software comprehension, or when applied to test code, test suite comprehension. Corbi (1989) defines software comprehension as “the task of building mental models of an underlying software system at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, evolution, and re-engineering purposes”. And Binder (1999) states that “*Effective testing cannot be achieved without using abstraction to conquer the astronomical complexity of typical software systems*”.

¹<http://www.eclipse.org/mylyn/>

The process of constructing mental models at various levels of abstraction requires an investigation of the test artifacts, such as test code and test documentation, but also includes a study of the code and the documentation of the software system under test. This process can be difficult and tedious, and with the increases in size and complexity of the test suites and the system under test, also the process gets more challenging. For plug-in-based systems, investigating static artifacts may not even be enough to understand the test suite, and which plug-ins are tested by it, as only during runtime the concrete bindings between plug-ins are made. This may require the developer to investigate the software executions to reveal the concrete configuration of the system under test and of the test environment.

The process of obtaining a mental model of a software system can be supported by tools that are based on software reverse engineering techniques, which provide the user with a representation of the system under investigation (e.g., the test system) at various levels of abstraction. The term *reverse engineering* refers to the process of deciphering designs from finished products, and was originally used to analyze hardware components (Chikofsky and Cross, 1990). Reverse engineering is a process of examination, whereby it generally involves extracting design artifacts and building or synthesizing abstractions of the system that are less implementation-dependent. The objective of reverse engineering is often to gain a sufficient understanding of the software system to support maintenance, enhancement or replacement activities (Müller et al., 2000).

The two main topics of this dissertation are plug-in-based systems and test suite comprehension. In particular, challenges during testing plug-in-based systems are revealed, and reverse engineering based techniques to support developers during test suite comprehension are presented.

The following sections detail the overarching research questions (Section 1.2) and outline the research methods in use (Section 1.3). Section 1.4 presents an overview of the individual chapters of this dissertation. The individual chapters correlate with the studies performed, and are kept in a way that allows to read them independently at the reader’s wish, as outlined in Section 1.6.

1.2 Research Questions

As outlined in Section 1.1.1, testing modular systems in general (Pohl and Metzger, 2006; Rehmand et al., 2007; Weyuker, 1998), and plug-in-based products in particular, is a daunting task. Especially integration testing is aggravated in such dynamic and modular environments as the myriad of plug-in combinations, versions, interactions, and configurations gives rise to a combinatorial explosion of possibilities. Yet in practice, the systems assembled from plug-ins are widely used, achieving levels of reliability that permit successful adoption. So, which test techniques are used and which challenges must be actually faced by developers to test plug-in-based systems?

In this dissertation, we set out to increase our understanding of how systems assembled from plug-ins are to be tested, which challenges developers face during testing of plug-in systems and, in particular, during integration testing. One of these challenges is the substantial amount of test code, which causes developers to have difficulties understanding and maintaining these large test suites. Therefore, we further investigate how developers can be supported during test suite comprehension and maintenance tasks.

In particular, we investigate the following four research questions in this dissertation:

- [RQ1] What makes testing of modular and dynamic systems challenging?
- [RQ2] What makes integration testing more difficult than unit testing?
- [RQ3] How can we support developers during understanding high level tests?
- [RQ4] How can we support developers during test maintenance?

1.3 Research Methods

During the studies included in this thesis, we use a wide variety of research methods, such as grounded theory (Bryant and Charmaz, 2007; Corbin and Strauss, 1990), interviews (Gubrium et al., 2012), surveys (Fowler, 2002), case study research (Yin, 2003) and software repository mining (Kagdi et al., 2007). Often, we use a mixed method approach, combining several methods in one study to triangulate the findings. We adopt such a mixed method approach as it provides a better understanding of research problems than one approach alone (Creswell and Vicki, 2006). The two main pillars of this research are involving practitioners and open as well as closed source software systems.

We assume that empirical evaluations and the involvement of the industry in software engineering research are crucial to address challenges faced by practitioners and develop techniques and tools that have a chance to be useful and applicable for real world problems. In our studies, we are of the opinion that only by involving people, in particular knowledgeable practitioners, we are able to reveal testing practices and problems during plug-in testing experienced in practice (Chapter 2), reveal challenges during test code comprehension (Chapter 3) and develop tools and techniques that are useful for practitioners (Chapter 3 and 5). Especially, grounded theory, a method originated from the social sciences, is appropriate for our studies as it is suitable for explorative, human-centered research areas.

Case study research mainly applied to open source software systems is the second pillar of this research. To date, many business models and revenues of large software companies and foundations rely on open source software development. Some of the most successful and widely used software products, such

RQ	Research question	Chapters
RQ1	What makes testing of modular and dynamic systems challenging?	Chapter 2 and 3
RQ2	What makes integration testing more difficult than unit testing?	Chapter 2 and 3
RQ3	How can we support developers during understanding high level tests?	Chapter 3 and 4
RQ4	How can we support developers during test maintenance?	Chapter 5 and 6

Table 1.1: *Mapping of research questions to chapters*

as Eclipse IDE, Mozilla Firefox, Apache web server and many more, are solely created through open source software development. The impact of open source software on software engineering research is tremendous. Open source systems allow us to evaluate our techniques and tools in realistic settings which also grant reproducibility and openness for other researchers to verify and challenge the findings. Case study research, often applied to open source systems, helps us to evaluate the scalability, applicability and accuracy of our techniques and tools. Within this research, we greatly benefit from the open source community, but we also think, we can contribute back with our involvement, as we take the effort to identify challenges in practices, develop techniques to address those and present our findings at several industry conferences. Furthermore, our tools are publicly available for download.²

1.4 Research Overview

This section gives a short overview of the chapters of this dissertation. Each research question presented in Section 1.2 is addressed by two chapters, as illustrated in Table 1.1.

In Chapter 2, we present a grounded theory study in which we interviewed 25 professional software engineers about their testing practices and challenges for plug-in-based systems. In this study, we reveal that because testing plug-in-based software is complex and challenging, many developers focus on unit testing and rather circumvent higher level testing activities, such as integration or system

²At <http://swel1.tudelft.nl/bin/view/MichaelaGreiler/Software>, the following tools are available: *ETSE*, *ConnectionMiner*, *TestHound* and *TestEvoHound*.

level testing. As discussed more elaborately in Chapter 2, developers indicate that their main focus for test automation lies on unit testing. The degree of test automation decreases for higher test levels, such as integration, GUI or system testing. In this study, we also detail the barriers that hinder integration and system test adoption, and finally show how limited testing is compensated by the involvement of the community in the test activities.

In Chapter 3, we distill several information needs developers have during test suite comprehension for plug-in systems from interviews. Based on these information needs, we developed a static and dynamic analysis technique that provides the developer with an abstraction of the system under test and its test suites, by recovering five architectural views from the system which highlight the integration points with other plug-ins and how this integration is tested. We implemented this technique in a tool called the Eclipse Test Suite Exploration (ETSE) tool. The evaluation is based on case study research, involving three well-known open source systems, to show the applicability, precision and scalability, as well as an initial user study addressing the usefulness of the tool.

In Chapter 4, we present a dynamic analysis technique that supports developers to understand relationships between different types of test suites (i.e., high level tests and unit tests). This technique links tests together based on the similarity of their execution traces, whereby several trace reduction techniques and similarity metrics come into play. We implemented this technique in a framework called the *Test Similarity Correlator*. We used case study research and mined the test suites of two open source systems with our tool in order to evaluate the applicability and scalability of our technique. Furthermore, we compared the automatically derived similarity relations with the similarity understanding of a human expert. In this study, we show that the automatic derived relations reflect well the understanding of similarity of a human expert and are helpful during software comprehension.

In Chapter 5, we present a static analysis technique that automatically detects *smells*, i.e., inadequate solutions to recurring implementation and design problems, in test code. In particular, we analyzed the code that initializes and configures the system under test (i.e., the *test fixture*) in order to detect *smells* related to the test fixture (i.e., test fixture smells). This detection technique is implemented in a tool called *TestHound*. We evaluated the applicability, scalability and usefulness of this technique by applying it to three software systems, and by presenting the detected smells to a group of 13 software developers. In this study, we show that fixture-related smells exist in practice and that software developers experience TestHound as helpful during understanding test fixture smells.

In Chapter 6, we further investigated fixture-related test smells, by studying their evolution, in order to improve integration of tool support in the continuous integration environment. Therefore, we studied the trends of test fixture smells over time, by performing software repository mining on five well-known open source systems. We implemented our technique to mine test fixture smells for several revisions of software systems and to calculate smell trends in a tool called

TestEvoHound. The findings of this analysis allow to reveal several strategies for test smell avoidance, and show which software changes cause severe increases in test fixture smells.

1.5 Related Work

Test Culture. A few surveys have been conducted in order to reveal software testing practices (Ng et al., 2004; Garousi and Varma, 2010). While these surveys focus on reporting testing practices, we had the additional aim of understanding why certain practices are used or are not used. In a survey, researchers can only address previously defined hypotheses. Our grounded theory study in Chapter 2 on testing practices allowed first to emerge a theory about the testing practices, and to let the structure and the content of the survey follow from the theory.

As an implication, while other surveys concentrate on pre-conceived barriers to testing, such as costs, time and lack of expertise, we could address a much wider range of factors of an organizational and technical nature, as expressed by the participants themselves. Further, the grounded theory findings drove the selection of test practices included in the survey. This allowed us to concentrate on facts specially relevant for plug-in systems (reflected in a separate section of the survey), and in turn to omit questions such as generation of test cases or defect prevention techniques used in previous studies.

Testing component-based systems is known to be a daunting task (Pohl and Metzger, 2006; Rehmand et al., 2007; Weyuker, 1998). Therefore, test practices addressing specific test problems or characteristics for such systems have been introduced. For example, Binder (1999) presents the *popular framework test* pattern that focuses on validating the delta of a further developed framework, and Mariani et al. (2007) investigate regression and compatibility testing for component replacements. Research on configuration-aware software testing of highly-configurable systems focuses often on the combinatorial problem, for example by detecting valid and invalid combinations of configuration parameters (e.g., by means of a greedy algorithm) (Cabral et al., 2010), or by using test prioritization techniques for compatibility testing (Yoon et al., 2013). Especially, testing of software product lines has been investigated during the last decade (Muccini and Hoek, 2003), whereby Lee et al. (2012) report that many research efforts have “focused on solving narrow research challenges”.

Our work, on the other hand, reveals broader testing practices and problems during plug-in testing experienced in practice.

Test comprehension. A recent survey on the use of dynamic analysis for program understanding purposes is provided by Cornelissen et al. (2009). One of the findings of this survey is that very few studies exist addressing dynamically reconfigurable systems – a gap that we try to bridge with our study presented in Chapter 3.

Cornelissen et al. (2007) also worked on supporting the understanding of test

suites, and investigate the automated extraction of sequence diagrams from test executions. Zaidman et al. (2008) investigate implicit connections between production code and test code, by analyzing their co-evolution in version repositories. Koochakzadeh and Garousi (2010) present a graph-based test coverage visualization tool, whose usefulness is evaluated by Garousi and Varma (2010). The tool allows developers to view the test coverage between two artifacts on different scopes (i.e., test package, class and method). While these studies provide important starting points, none of them approaches test suite understanding from an integration or extensibility point of view, nor do they address the plug-in characteristics of systems under test.

Few researchers also worked on techniques to establish relations between test cases to support test prioritization (Yoo et al., 2009), or test case selection (Rothermel and Harrold, 1998; Hurdugaci and Zaidman, 2012). Galli et al. (2004b) have developed a tool to order broken unit tests to steer and optimize the debugging process. Our technique in Chapter 4 complements this work as we establish links between test cases to assist developers in their (test suite) comprehension and maintenance activities.

In the area of test suite analysis and understanding, van Deursen et al. (2002) proposed a series of *test smells* (pointing to hard to understand test cases) as well as a number of refactorings to remedy them. Later, this work was substantially elaborated by Meszaros (2007) into an extensive book on xUnit patterns.

Few research focuses on automatic detection of test smells. Among them, Van Rompaey et al. (2007) tried to detect the test smells *General fixture* and *Eager test* by means of metrics. In a subsequent paper, they describe a tool which uses well known software metrics to predict a broader variety of potential problems and test smells (Breugelmans and Van Rompaey, 2008). Our studies, in Chapters 5 and 6, differ in several aspects. First of all, we focus on test fixture management and analysis of the test code for specific fixture problems relevant in practice, and provide concrete refactoring suggestions. In contrast to our work, Borg and Kropp (2011) describe automated refactoring for acceptance tests based on the FIT framework. To the best of our knowledge, fixture-related test smells and refactoring have not been studied in detail so far.

1.6 Origin of papers

This section elucidates the origin of the different chapters, as they are based on peer-reviewed publications created for this dissertation.

Each chapter can be read in separation. The author of this dissertation is the main author of all publications.

- Chapter 2 appeared in the proceedings of the 2012 34th International Conference on Software Engineering (ICSE 2012). This paper is referenced as (Greiler et al., 2012a). The authors of this publication are Greiler, van Deursen, and Storey.

- Chapter 3 is published in the Empirical Software Engineering journal, November 2012, and is an extended version of the paper “Understanding Plug-in Test Suites from an Extensibility Perspective”, which appeared in the proceedings of the 2010 IEEE 17th Working Conference on Reverse Engineering (WCRE 2010). This article is referenced as (Greiler and van Deursen, 2012). The authors of this publication are Greiler and van Deursen.
- Chapter 4 is published in the proceedings of the 2012 50th International Conference on Objects, Models, Components, Patterns (TOOLS 2012). This paper is referenced as (Greiler et al., 2012b). The authors of this publication are Greiler, van Deursen, and Zaidman.
- Chapter 5 appeared in the proceedings of the 2013 6th International Conference on Software Testing, Verification and Validation (ICST 2013). This article is referenced as (Greiler et al., 2013a). The authors of this publication are Greiler, van Deursen, and Storey.
- Chapter 6 will appear in the proceeding of the 2013 10th Working Conference on Mining Software Repositories (MSR 2013). This paper is referenced as (Greiler et al., 2013b). The authors of this publication are Greiler, Zaidman, van Deursen and Storey.

Apart from these publications the author has been first author of the following publications which have been created during the PhD trajectory but are not directly included in this thesis:

- “Evaluation of Online Testing for Services - A Case Study”, which appeared in the proceedings of the 2010 2nd International Workshop on Principles of Engineering Service Oriented Systems (PESOS 2010). The authors of this publication are Greiler, Gross and van Deursen.
- “Runtime Integration and Testing for Highly Dynamic Service Oriented ICT Solutions”, which has been published in the proceedings of the 2009 4th Testing: Academic and Industrial Conference (TAIC PART 2009). The authors of this publication are Greiler, Gross and Nasr.

Test Confessions: A Study of Testing Practices for Plug-In Systems

Abstract

Testing plug-in-based systems is challenging due to complex interactions among many different plug-ins, and variations in version and configuration.¹ The objective of this paper is to increase our understanding of what testers and developers think and do when it comes to testing plug-in-based systems. To that end, we conduct a qualitative (grounded theory) study, in which we interview 25 senior practitioners about how they test plug-in applications based on the Eclipse plug-in architecture. The outcome is an overview of the testing practices currently used, a set of identified barriers limiting test adoption, and an explanation of how limited testing is compensated by self-hosting of projects and by involving the community. These results are supported by a structured survey of more than 150 professionals. The study reveals that unit testing plays a key role, whereas plug-in specific integration problems are identified and resolved by the community. Based on our findings, we propose a series of recommendations and areas for future research.

¹This chapter appeared in the proceedings of the 2012 34th International Conference on Software Engineering (ICSE 2012). The authors of this publication are Greiler, van Deursen, and Storey.

2.1 Introduction

Plug-in architectures permit the composition of a wide variety of tailored products by combining, configuring, and extending a set of plug-ins (Chatley et al., 2004; Marquardt, 1999). Many successful plug-in architectures are emerging, such as Mozilla’s Add-on infrastructure² used in the Firefox browser, Apache’s Maven build manager,³ the WordPress extension mechanism,⁴ and the Eclipse⁵ plug-in platform.

Testing component-based systems in general (Pohl and Metzger, 2006; Rehmand et al., 2007; Weyuker, 1998), and plug-in-based products in particular, is a daunting task; the myriad of plug-in combinations, versions, interactions, and configurations gives rise to a combinatorial explosion of possibilities. Yet in practice, the systems assembled from plug-ins are widely used, achieving levels of reliability that permit successful adoption. So which test techniques are used to ensure plug-in-based products have adequate quality levels? How is the combinatorial explosion tackled? Are plug-in specific integration testing techniques adopted? For what reasons are these approaches used?

Answering questions like these calls for an in-depth study of test practices in a community of people working on plug-in-based applications. In this paper, we present such a study, revealing what Eclipse community practitioners think and do when it comes to testing plug-in based systems.

Eclipse provides a plug-in-based architecture that is widely used to create a variety of extensible products. It offers the “Rich Client Platform” to build plug-in-based applications and a series of well-known development environments (Shavor et al., 2005). Eclipse is supported by a global community of thousands of commercial, open and closed source software professionals. Besides that, the Eclipse case is interesting as it benefits from a rich testing culture (Gamma and Beck, 2003; Greiler et al., 2010).

We set up our investigation as an *explorative* study. Thus, instead of starting out with preset hypotheses on how testing is or should be done, we aimed to discover how testing is actually performed, why testing is performed in a certain way, and what test-related problems the community is facing. Therefore, we used *grounded theory* (Adolph et al., 2011; Corbin and Strauss, 1990) to conduct and analyze open interviews (lasting 1–2 hours) with 25 senior practitioners and thought leaders from the Eclipse community regarding their test practices.

Our results show a strong focus on unit testing, while the plug-in specific testing challenges and practices are tackled in an *ad-hoc* and manual manner. Based on our results, we identified barriers which hinder integration testing practices for plug-in systems. Furthermore, we analyzed how the lack of explicit testing beyond the unit scope is compensated for, for example through self-hosting of projects

²<https://developer.mozilla.org/en-US/addons>

³<http://maven.apache.org>

⁴<http://wordpress.org/extend/plugins>

⁵<http://www.eclipse.org>

and involvement of the community. We challenged our outcomes through a separate structured survey, in which 151 professionals expressed their (dis)agreement with specific outcomes of our study. Furthermore, we used the findings to propose a series of recommendations (at the technical as well as the organizational level) to improve plug-in testing, community involvement, and the transfer of research results in the area of integration testing.

The paper is structured as follows. In Section 2.2, we sketch the challenges involved in plug-in testing. Then, in Section 2.3, we layout the experimental design and the steps we conducted as part of our study. In Sections 2.4–2.7 we present the key findings of our study, including the test practices used, the barriers faced, and the compensation strategies adopted. In Sections 2.8–2.9, we reflect on our findings, addressing implications as well as limitations of our research. We conclude with a survey of related work (Section 2.10), and a summary of our key findings (Section 2.11).

2.2 Plug-in Systems: Capabilities and Challenges

Plug-in-based systems rely on plug-in components to extend a base system (Marquardt, 1999; Shavor et al., 2005; Voelter, 2001). As argued by Marquardt (1999), a base system can be delivered almost “nakedly”, while most user value is added by plug-ins that are developed separately, extending the existing applications without the need for change. In more sophisticated plug-in architectures, plug-ins can build upon each other, allowing new products to be assembled in many different ways. In contrast to static libraries, plug-ins can be loaded at runtime. Further, plug-ins make use of the inversion of control principle to allow customization of a larger software system.

This means that plug-in systems can be complex compositions, integrating multiple plug-ins from different developers into one product, and raising concerns about the compatibility of their components (Pohl and Metzger, 2006; Rehmand et al., 2007; Weyuker, 1998). Incompatibility, be it because of combinations of plug-ins or versions, can be hard to strive against, and may restrict the benefits plug-in systems offer. For example, many users of the popular WordPress blog-software suffer from compatibility issues, and according to their own statement, *“The number one reason people give us for not upgrading to the latest version of WordPress is fear that their plugins won’t be compatible.”*⁶ There are many resources on the Internet stating incompatible plug-in combinations.⁷ Still, incompatibility of plug-in combinations is an open issue.⁸

These same challenges also occur with Eclipse where combinations of plug-ins

⁶<http://wordpress.org/news/2009/10/plugin-compatibility-beta>

⁷For example, plug-ins incompatible with Onswipe <http://wordpress.org/support/topic/plugin-onswipe-list-of-incompatible-plugins-so-far>

⁸<http://www.wpmods.com/wordpress-plugin-compability-procedure>

or versions can be incompatible.⁹ For example, while resolving a Mylyn issue and tackling an integration problem with a specific Bugzilla version, a user states: *“Thanks, but I think we have given up on Eclipse and Bugzilla integration.”*¹⁰ On project pages, phrases such as: *“However we can not guarantee compatibility with a particular plug-in combination as we do not test with all possible connector combinations”*¹¹ commonly appear.

Such problems exist in many plug-in systems, which sparked our interest and led us conduct a thorough investigation.

2.3 Experimental Design

Testing plug-in-based systems raises a number of challenges related to the interactions between plug-ins, different configurations of the plug-ins, and different versions of the plug-ins used. The overall goal of this paper is to increase our understanding of what testers and developers think and do when it comes to testing plug-in-based systems.

2.3.1 The Eclipse Plug-In Architecture

As the subject of our study, we selected the Eclipse plug-in framework¹² along with its community of practitioners. We selected Eclipse for a number of reasons.

First, Eclipse provides a sophisticated plug-in mechanism based on OSGi¹³ and to that is enhanced with the Eclipse-specific extension mechanism. It is used to build a large variety of different applications,¹⁴ ranging from widely used collections of development environments, to dedicated products built using the Rich Client Platform (RCP). Many of these plug-in-based products are large, complex, and industrial strength.

Second, there is a large community of professionals involved in the development of applications based on the Eclipse plug-in framework. As an example, approximately 1,000 developers meet at the annual EclipseCon event alone.

Third, the Eclipse community has a positive attitude towards testing, as exemplified by the presence of substantial test suites (see our analysis of the Mylyn and eGit test suites (Greiler et al., 2010)) and books emphasizing the test-driven development of plug-ins (Gamma and Beck, 2003). Moreover, Eclipse has explicit support for the testing of plug-ins, through dedicated *Plug-in Development Environment* (PDE) tests.

Finally, the Eclipse framework, as well as the many projects built upon it, are open source. This makes it easy to inspect code or documentation, as well as to

⁹To mention only a few bugs on Bugzilla: 355759, 292783, 196164

¹⁰Bug Identifier: 268207

¹¹<http://sourceforge.net/apps/mediawiki/qcmylyn>

¹²<http://www.eclipse.org>

¹³<http://www.osgi.org>

¹⁴http://en.wikipedia.org/wiki/List_of_Eclipse-based_software

share findings with other researchers. Since the Eclipse platform is also used for closed source commercial development, it is possible to compare open and closed source testing practices.

2.3.2 Research Questions

Our investigation of the testing culture for plug-in-based systems revolves around four research questions. The first three we incorporated in the initial interview guidelines. During our interviews, many professionals explained how they compensate for limited testing, which helped to refine the interview guidelines and led to the last research question.

RQ1: Which testing practices are prevalent in the testing of plug-in-based systems? Do these practices differ from non-plug-in-based systems?

RQ2: Does the plug-in architecture lead to specific test approaches? How are plug-in specific integration challenges, such as versioning and configurations, tested?

RQ3: What are the main challenges experienced when testing plug-in-based systems?

RQ4: Are there additional compensation strategies used to support the testing of plug-ins?

2.3.3 Research Method

This section outlines the main steps of our experimental design. The full details of our setup can be found in the corresponding technical report (Greiler et al., 2011, Appendix A).

We started with a survey of existing approaches to plug-in testing. We studied over 200 resources about the testing of plug-in systems in general, and the Eclipse plug-in architecture in particular. Information was drawn both from developer forums and the scientific literature. Most of the articles found were concerned with technical problems, such as the set-up of the test environment. They did not, however, provide an answer to our research questions.

Next, we conducted a series of interviews with Eclipse experts, each taking 1–2 hours. Interviews were in German or English, which we subsequently transcribed. The questions were based on a guideline, which was refined after each interview. We followed a *grounded theory* (GT) approach, an explorative research method originating from the social sciences (Glaser and Strauss, 1967), but increasingly popular in software engineering research (Adolph et al., 2011). GT is an inductive approach, in which interviews are analyzed in order to derive a theory. It aims at discovering new perspectives and insights, rather than confirming existing ones.

Table 2.1: *Domains, projects, and companies involved in the interviews*

Domain	Project and/or Company
IDEs, Eclipse Distribution	Yoxos, EclipseSource
SOA	Mangrove, SOA, Inria
GUI Testing Tool	GUIDancer, Bredex
Version Control Systems	Mercurial, InlandSoftware
Modeling	xtext, Itemis
Modeling	IMP, University of Amsterdam
Persistence layer	CDO
Domain Specific Language	Spoofax, TU Delft
BPM Solutions	GMF, BonitaSoft
GUI Testing Tool	Q7, Xored
Coverage Analysis	EclEmma
Modeling	EMF, Itemis
BPM Solutions	RCP product, AndrenaObjects
Scientific data acquisition	OpenGDA, Kichacoders
Runtime platform	RAP, EclipseSource
Task Management system	Mylyn, Tasktop
Embedded Software	MicroDoc
RCP product	EclipseSource

As part of GT, each interview transcript was analyzed through a process of *coding*: breaking up the interviews into smaller coherent units (sentences or paragraphs), and adding *codes* (representing key characteristics) to these units. We organized codes into *concepts*, which in turn were grouped into more abstract *categories*. To develop codes, we applied *memoing*: the process of writing down narratives explaining the ideas of the evolving theory. When interviewees progressively provided answers similar to earlier ones, a state of *saturation* was reached, and we adjusted the interview guidelines to elaborate other topics.

The final phase of our study aimed at evaluating our outcomes. To that end, we presented our findings at EclipseCon,¹⁵ the annual Eclipse developer conference. We presented our findings to a broad audience of approximately 100 practitioners during a 40-minute extended talk, where we also actively requested and discussed audience feedback.

Furthermore, we set up a survey to challenge our theory, which was completed by 151 practitioners and EclipseCon participants. The survey followed the structure of the resulting theory: the full questionnaire is available in the technical report (Greiler et al., 2011).

2.3.4 Participant Selection

For the interviews, we carefully selected knowledgeable professionals who could provide relevant information on testing practices. We contacted them by participating in Eclipse conferences and workshops, through blogging, and via Twit-

¹⁵<http://www.eclipsecon.org/2011/sessions/?page=sessions&id=2207>

ter. Eventually, this resulted in 25 participants from 18 different companies, each working on a different project (identified as P1–P25 in this paper), whose detailed characteristics are provided in (Greiler et al., 2011, Appendix A). All have substantial experience in developing and/or testing Eclipse plug-ins or RCP products. 12 participants are developers, 11 are project leads, 1 is a tester and 1 is a test manager. The respective projects are summarized in Table 2.1.¹⁶

In the survey phase, we aimed to reach not only the experts, but the full Eclipse community. To that end, we set up an online survey and announced it via mailing lists, Twitter, and our EclipseCon presentation. This resulted in 151 participants filling in the questionnaire. The majority of the respondents were developers (64%), followed by project leads or managers. Only 6% were testers or test managers.

2.3.5 Presentation of Our Findings

In the subsequent sections, we present the results of our study, organized in one section per research question. For each question, we provide relevant “*quotes*” and *codes*, make general observations, and list outcomes of the evaluative survey.

In the Appendix A, we provide additional data supporting our analysis. In particular, we provide the coding system we developed, comprising 4 top-level categories, 12 subordinate concepts, and 1-10 basic codes per concept, giving a total of 94 codes. For each code, we give the name as well as a short one-sentence description. Furthermore, the technical report provides 15 pages of key quotes illustrating the codes. Last but not least, we provide the full text of the survey, as well as response counts and percentages.

2.4 Testing Practices

Our first research question seeks to understand which practices are used for testing plug-in-based systems, and which software components (i.e., test scope) these address.

2.4.1 Open Versus Closed Development Setting

Approximately half of the participant projects are open source, with the other half being closed source projects (often for a single customer). The participant companies that develop open source software typically also work on closed source projects. The purpose of software development is purely commercial for all but two projects. Open source projects count, for example, on selling functional extensions for the open source product in supplementary products.

¹⁶Please note that for reasons of confidentiality not all companies and projects participating at the interviews are listed.

Most of our participants are paid to develop open source software. A few develop open source products in their free time, but profit personally from the marketing effect, e.g., for their own consultancy company.

In the survey, 21% of the respondents indicated that they develop pure open source, 47% pure closed source, and 32% indicate that they work on both types of projects.

2.4.2 Test Responsibilities

The interviews reveal that it is a common practice to have no dedicated test team, but that testing is performed by the developers themselves (P1, P2, P4, P5, P6, P7, P8, P9, P12, P13, P15, P16, P17, P18, P19). P5 explains: *“Tester and developer, that’s one person. From our view, it does not make sense to have a dedicated test team, which has no idea about what the software does and can only write some tests.”*

Only a few projects report to have dedicated testers, either within the development team or in a separate quality assurance team (P3, P10, P11, P14, P21). P21 explains: *“Automated tests are only developed by developers. Manual testing is done partly [...] Regression testing is done by someone from the customer.”*

Both practices are used in open and closed source projects. Respondents to the survey indicate that closed source projects are more likely to have dedicated teams (41%) than open source or hybrid projects (24%).

2.4.3 Unit Tests

Automated unit tests are very popular, probably because in the majority of the projects, developers are responsible for testing. The teams of P1, P4, P7, P13, P16, P20, and P22 use unit testing as the only automated form of testing; all other forms are manual. P20 gives the strongest opinion: *“We think that with a high test coverage through unit tests, integration tests are not necessary.”* And P18 says: *“At our company, testing is quite standard. We have different stages. We have unit testing, and that’s where we put the main effort – at least 70% of the total expenses.”* Also P15 reports: *“The majority of the tests are written with JUnit, and the main test suites comprise tests that do not depend on Eclipse.”*

The majority of the participants share P14’s opinion: *“Try to get to a level that you write unit tests, always, whenever you can. [...] at max. you use one integration or PDE test to probe the code. Ultimately, unit tests are our best friends, and everything else is already difficult.”*

Participants are aware that unit testing is not always applicable. For projects that rely solely on unit testing, this has visible implications. As P20 confirms: *“We try to encapsulate the logic as much as possible to be able to test with unit tests. What cannot be encapsulated is not tested.”*

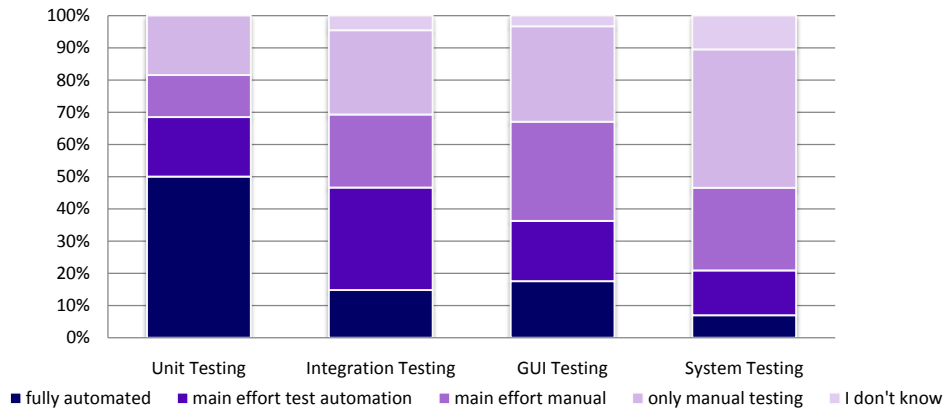


Figure 2.1: Test automation for each test practice

2.4.4 Beyond Unit Testing

There are many other testing practices used, such as integration, GUI, and system testing, but many participants do not describe them as their focus or key practice.

The second most applied techniques are manual and automated integration testing (P3, P5, P6, P8, P10, P11, P12, P14, P15, P17, P18, P19, P21). The PDE test framework is most commonly used for automating integration testing. Participants indicate that they use integration tests for testing server-side logic, embedded systems, and third-party systems connected through the network. Integration tests also include tests indirectly invoking plug-ins throughout the ecosystem. In Section 2.5, we will see that PDE tests are often used in place of unit tests.

Successful adoption and active use of automated GUI testing is limited to four projects. Many participants see alternative solutions to the “expensive” (P15) automated GUI testing approaches by keeping the GUI as small as possible and by decoupling the logic behind a GUI from the GUI code as much as possible (P13, P16, P17, P20, P23). As P13 puts it: “We try to make a point of surfacing as little visible stuff in the UI as possible.” In summary, the degree of adoption, and especially automation, decreases drastically for test practices with a broader scope.

The survey, aimed at the broader Eclipse community, enquires about test effort and the level of automation used for unit, integration, GUI, and system testing. The answers suggest a more or less balanced distribution of total *effort* per test form, but a decrease in automation level. Thus, as illustrated in Figure 2.1, automation drops from 65% for unit, to 42% for integration, to 35% for GUI, and to only 19% for system testing. 37% of the respondents indicate they rely solely on manual testing at the system scope.

What consequences does this have for integration testing? Do practitioners

address plug-in specific characteristics during integration? The findings are described in the following section.

2.5 Plug-In Specific Integration Testing

Our next question (RQ2) relates to the role that the plug-in nature plays during testing, and to what extent it leads to specific testing practices.

2.5.1 The Role of PDE Tests

PDE tests are designed to test plug-in-based Eclipse applications. They are JUnit tests using a special test runner that launches another Eclipse instance in a separate virtual machine. This facilitates calls to the Eclipse Platform API, as well as launching plug-ins during the test. Furthermore, the “headless” execution mode allows tests to start without user-interface components.

Participants often use PDE tests for unit testing purposes. According to P1: *“The problem begins when a JUnit test grows into a PDE test, because of the dependencies on the workbench.”* And P21 states: *“Our PDE tests do not really look at the integration of two components. There are often cases where you actually want to write a unit test, but then it’s hard to write, because the class uses something from the workbench.”* Others also report that they use integration tests for testing legacy code, and P14 reports to *“use integration tests to refactor a code passage, or to fix a bug, when you cannot write a unit test. Then, at least you write an integration test that roughly covers the case, to not destroy something big. That, we use a lot.”*

We next ask, since Eclipse is a plug-in architecture, are there plug-in specific aspects to consider for integration testing?

2.5.2 Plug-In Characteristics

In response to the interview questions regarding the influence plug-in architectures have on testing, participants come up with a variety of answers. Most of the participants consider plug-in testing as different from testing standalone Java applications. Only P8 and P10 report to not see any influence and that testing of plug-in systems is the same as testing monolithic Java applications.

The most often recognized difference is the need to have integration tests (P9, P14, P12, P15, P20). P14 thinks that integration testing becomes more important in a plug-in-based set-up because: *“We have to test the integration of our code and the Eclipse code, [...] And then, you test in a way differently, [...] you have more test requirements, there are more players in the game.”*

Practices differ in the strategies participants use to test plug-in systems and the extension mechanism. P2 says: *“I am not sure if there is a need to test if extensions correctly support the extension point, because it is mostly a registration*

thing.” Also, P13 does not address the plug-in aspect directly, but says: “Our test cases make use of extension points, so we end up testing if extension point processing is working correctly.” P19 presents the most advanced technique to testing by stating: “In some cases, we have extensions just for testing in the test plug-ins. Either the extensions are just loaded or they implement some test behavior.” P19’s team also recommends that developers writing extensions should look at the relevant tests because those tests demonstrate how to use the API.

P12, P16 and P19 report that the extension mechanism makes the system less testable. P16 says: “We tried a lot. We test our functionality by covering the functionality of the extension point in a test case, i.e., testing against an API. The small glue code where the registry gets the extension, that’s not tested, because it is just hard to test that. And for these untested glue code parts we had the most bugs.” And P19 says: “Testing is more difficult, especially because of the separate classloaders. That makes it complicated to access the internals. Therefore some methods which should be protected are public to enable testing.”

Participants associate many different aspects, such as improved modularization capabilities for production and test code, with plug-in architectures and testing. Surprisingly, only a few participants mention the extension mechanisms, and none of the participants mention OSGi services, runtime binding or combinatorial problems for plug-in interactions. This finding leads to our follow-up questions for specific plug-in testing techniques.

2.5.3 Testing Cross-Product Integration

To gain a better understanding of the participants’ integration testing practices, we ask how they test the integration of their own plug-ins with third-party plug-ins (i.e. *cross-product integration testing*), and how they deal with the corresponding combinatorial problem.

To our surprise, none of the projects report to have automated tests to ensure product compatibility. Many participants report that products “*must play nicely with each other*”¹⁷ and that there are no explicit tests for different combinations.

Does this mean that cross-product integration problems do not occur? The answers to this question split the participants in two opposing camps. One group believes that these problems should not happen (P4, P5, P8, P12, P13, P14, P17), but more than half of the participants report to have actually experienced such problems (P2, P6, P7, P9, P10, P11, P15, P16, P18, P19, P20, P24, P25). Some even pointed us directly to corresponding bug reports.¹⁸

Participants report that cross-product integration testing is mainly performed manually, or in a bug-driven way (P15, P16, P18, P19). P18 explains: “We handle problems between several plug-ins in a bug-driven way. If there is a bug we write a test, but we do not think ahead which problems could there be.” And P10

¹⁷<http://eclipse.org/indigo/planning/EclipseSimultaneousRelease.php>

¹⁸Bug Identifier: 280598 and 213988

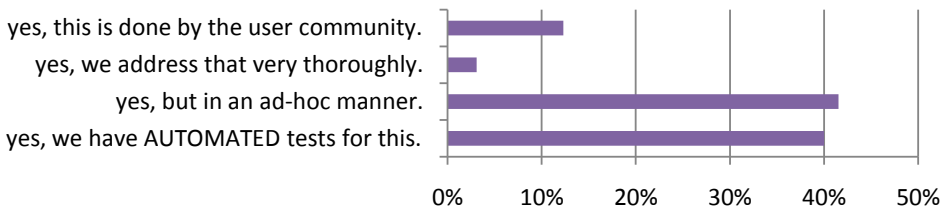


Figure 2.2: *Cross-Product Integration Testing*

reports: “There are no specific types of tests for [integrating multiple plug-ins], but it is covered by the end user tests, and by the GUI tests, which communicate amongst plug-ins, but the internal coverage is more random.”

In the open source domain, participants report that the community reports and tests for problems with plug-in combinations (P6, P9, P13, P16, P19, P20). As P19 says: “we have no automated tests for cross-product problems, but we do manual testing. Then, we install [product 19] with [several other plug-ins] or with other distributions, like MyEclipse, to test for interoperability.” And then he adds: “The user community plays an important role in testing for interoperability.” User involvement emerged as an important strategy for dealing with combinatorial complexity, as we will see in Section 2.7.

In the survey, 43% of the participants indicate that they do not test the integration of different products at all. Out of the 57% who stated that they test cross-product integration, 42% claim to address this in an *ad-hoc* manner, and only 3% claim to address this issue thoroughly (see Figure 2.2).

Thus, testing combinations with third-party plug-ins is not something participants emphasize. This leads us to ask, how are they ensuring compatibility of their plug-ins with the many different versions of the Eclipse platform?

2.5.4 Testing Platform and Dependency Versions

Only a few participants report testing for different versions of the Eclipse platform, typically the most currently supported version. For most of the other participants, P13’s assessment reflects what is done in practice: “A lot of people put version ranges in their bundle dependencies, and they say we can run with 3.3 up to version 4.0 of the platform. But I am willing to bet that 99% of the people do not test that their stuff works, they might assert it, but I do not believe that they test.”

However, in addition to the platform, plug-ins have specific versions and stipulate the versions of dependencies they can work with. How is compatibility for version ranges of plug-in dependencies tested?

In reality, many participants report that they test with one fixed version for each dependency (P8, P9, P11, P13, P14, P15). The minority of practitioners report that they have two streams of their systems. One stream for the latest

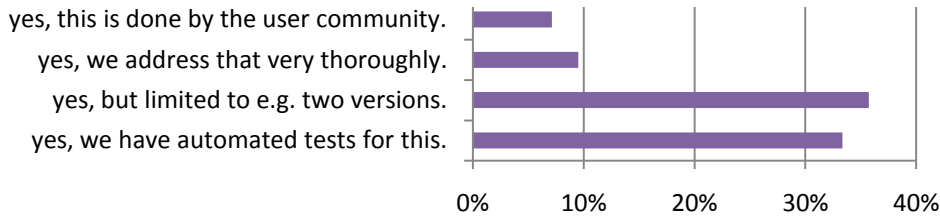


Figure 2.3: *Testing versions of plug-in dependencies.*

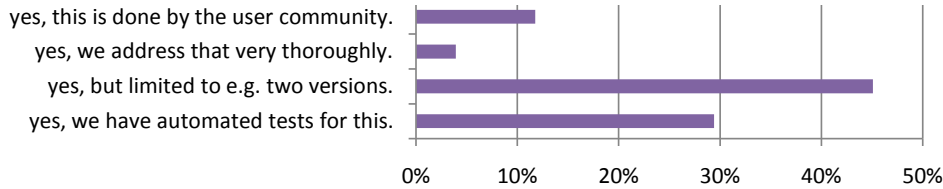


Figure 2.4: *Testing Eclipse platform versions.*

versions of dependencies, and the other one for the dependency versions used in the stable release.

Other projects report that they even ship the product with all dependencies and disable the update mechanisms. Updating dependencies to newer versions is often reported as a challenge. Many try to keep up to date, though some report to update rarely (P9, P11, P14). As P14 puts it: *“We always have one specific version for platform and libraries that we use. If we update that, that’s a major effort. That we do only rarely.”* And P9 says: *“We use a very old version of the main plug-in we depend on. Sometimes we update, but there is always the risk that it will break something and then you have to do extensive [manual] testing.”*

Testing version compatibility, as well as combinations of systems, is more often applied to third-party systems (i.e. outside the Eclipse ecosystem). For example, P10, P17, and P19 report to emphasize testing different versions of Eclipse-external third-party systems during automated testing, but not for Eclipse plug-ins they rely on or build upon.

Also, the majority of survey respondents indicate that they do not test version compatibility of either the platform (55%) or of plug-in dependencies (63%). Out of those testing different dependency versions, only 33% have automated tests, 36% indicate to limit it to a set number of versions, and only 10% test this thoroughly, as illustrated in Figure 2.3. Testing platform versions yields similar results: out of the 45% who indicate they test different versions, 29% have automated tests, 45% limit testing to a set number of versions, and only 4% indicate to address this thoroughly (see Figure 2.4).

2.6 Barriers for Adopting Plug-In Specific Integration Testing Practices

In the preceding sections, we looked at adopted testing practices. In this section, we outline barriers experienced by participants which limit adoption of plug-in specific test practices. The set of barriers reflects what the interviewees considered most important. To integrate the many different barriers and to identify relevant factors, the *constant comparison* approach of GT proved particularly useful (Glaser and Strauss, 1967).

Plug-in systems are conglomerates of several different plug-ins, with different owners. Hence, the *responsibility* for integration or system testing is less clear, especially when system boundaries are crossed. Most projects restrict their official support for compatibility with third-party plug-ins and the Eclipse platform itself. As P8 puts it: *“We only test the latest available versions of our dependencies, those that are together in the release train.”*

In plug-in systems, *end user requirements* are often unclear or even unknown, which makes testing a challenge, as P7 explains: *“[Project 7] is not an end-user plug-in. Other plug-ins build on top of [Project 7], so integration testing would need to include some other components. It is not the final, the whole thing.”* P7 also thinks that integration testing has to be done in strong collaboration with the developers of the end-user plug-in. As an example, he mentions syntax highlighting functionality: *“Only when I know about the language [...] can I test it and see whether it was successful or not. I need some third party component.”*

Also unclear *ownership* of plug-ins hinders testing, as P7 explains: *“You never know, once you write a good test, it will be obsolete with the next version of Eclipse.”*

While there is a rich body of literature on unit testing (Gamma and Beck, 2003), literature on integration and system testing for plug-in-based systems is scarce. This unavailability of *plug-in testing knowledge* makes it hard for beginners and less experienced developers and testers to test Eclipse-based systems. P4 explains: *“Why [testing] is so difficult? For Web projects, you find good templates. For Eclipse, you don’t. [...] Especially for testing plug-ins, we would need some best practices.”*

Setting-up a test environment for unit testing requires minimal effort as standard tooling (e.g., JUnit) exists. For integration, system, and GUI testing, the situation is different. Participants, such as P4, report: *“The difficulty of integration testing Eclipse plug-ins starts with the set-up of the build – that’s difficult.”*

Also, long *test execution time* is often mentioned as a reason for the negative attitudes towards integration, GUI, and system testing (P1, P4, P5, P6, P10, P17, P21). P6 says: *“The long execution time is really bad. A big problem.”* And P17 says: *“It’s a difference between 10 seconds and 1 minute: with 1 minute you switch to Twitter or Facebook.”*

As interviewees report, the limited *testability of Eclipse* can be challenging. P6

outlines: *“The problem is that the Eclipse platform is very hard to test, because components are highly coupled and interfaces are huge, and all is based on a singleton state. This is very hard to decouple.”*

The *PDE tooling* and test infrastructure can also be a hurdle. P21 says: *“We use the PDE JUnit framework to write integration tests, although we are not happy with it. It’s not really suited for that.”*

All of these technical hurdles have the effect that testing beyond unit scope is experienced as *“annoying”* (P6), *“distracting”* (P17), and *“painful”* (P20).

2.7 Compensation Strategies

As we saw in the previous sections, participants report that test automation for system and integration testing is modest. They also mention that integration testing for plug-in specific aspects, like cross-feature integration, versioning and configurations of plug-ins, is often omitted or limited to a manual and *ad-hoc* approach. Does this mean it is not necessary to test those aspects? Addressing this concern is the topic of research question RQ4, in which we seek to understand how developers compensate for limited testing.

During the GT study, we identified three main compensation strategies, namely *self-hosting of projects*, *user involvement*, *developer involvement*, and a prerequisite for participation – *openness*.

2.7.1 Self-Hosting of Projects

Self-hosting refers to the process whereby the software developed in a project is used internally on a regular basis. As P17 describes: *“In our company, we have different set-ups, based on Linux or Windows. This leads already to a high coverage because we use our own products on a daily basis. Then you are aware of problems and report that immediately.”* In the survey, a respondent writes: *“We use ‘self hosting’ as test technique. That is, we use our software regularly. This provides a level of integration testing, since common features are regularly exercised.”*

This practice is also applied at the code level, which means that participants report to use the API and provided extension points in their own projects. This principle, referred to as *“eating your own dog food”*, is well-documented in the Eclipse community (Harrison, 2006), and recognized for helping in managing and testing configurations of plug-ins, including combinations and different versions.¹⁹

¹⁹<http://dev.eclipse.org/newslists/news.eclipse.platform/msg24424.html>

2.7.2 User Involvement

Participants also report that they involve users to “manually test” their systems, as P9 explains: *“The tests that I perform are very simple manual tests, the real tests are coming from the users, who are doing all kind of different things with [project 9].”*

P9 is not alone with this practice. Participants openly state that they rely heavily on the community for test tasks, such as GUI testing, testing of different Eclipse platform versions, and system testing, and to cope with combinatorial testing and testing of plug-in combinations. As P12 says: *“Testing is done by the user-community and they are rigorous about it. We have more than 10,000 installations per month. If there is a bug it gets reported immediately. I do not even have a chance to test [all possible combinations]. There are too many operating systems, there are too many Eclipse versions.”*

2.7.3 Developer Involvement

The Eclipse plug-in architecture enables developers to build plug-ins on top of other plug-ins. Because of this, users of the software are often skilled developers whose projects also depend on and profit from the quality of the projects their work extends. Therefore, projects dedicate part of their time to improve dependent projects. As P11 states: *“Yes, for the GEF part, we find and report bugs, and we provide patches. In fact, perhaps it is not our own product, but our product relies on this other product. So it is normal to improve the other parts that we need.”*

Projects also profit from the automated test suites of the projects they extend. P13 explains: *“That is one of the things I totally rely on, e.g., the Web Tools Platform uses [project 13] heavily, and they have extensive JUnit tests, and so I am quite sure that when I break something that somebody downstream will rapidly notice and report the problem.”*

In Eclipse, the release train²⁰ is a powerful mechanism. Projects elected to be on the release train profit from the packaging phase, in which different bundles of Eclipse, including specific combinations of products, are created. As P13 explains: *“Some testing is performed downstream, when packages of multiple plug-ins are produced. Some packages have plug-ins like Mylyn, [project 13], and a whole ton of other projects. Then, there are people that test whether the packages behave reasonably.”* And he reports that *“if there are problems, people definitely report them, so you do find out about problems.”*

2.7.4 Openness – A Prerequisite for Participation

The question that remains is how to involve users and experts. In this study, we could identify one basic but effective principle, applied consistently by the

²⁰http://wiki.eclipse.org/Indigo/Simultaneous_Release_Plan

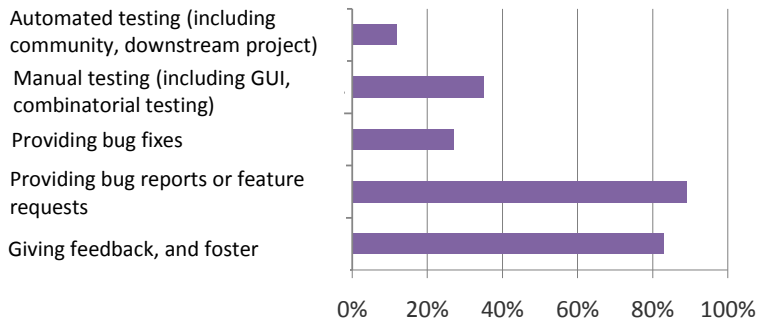


Figure 2.5: *User involvement during testing*

participants – openness. Openness is implemented in communications, release management, and product extensibility.

Open source projects select communication channels that allow the community to influence software development by giving feedback, fostering discussions, submitting feature requests, and even by providing bug fixes. In the closed source domain, participants report that they open up their communication channels to allow community participation. P19 reflects on the impact of user input: *“I would say the majority of the bug reports come from the community. We have accepted more than 800 patches during the life span of this project. 1/7 of all bugs that have been resolved have been resolved through community contributions. That’s quite a high rate. [...] we take the community feedback definitely serious.”*

An important prerequisite to user involvement is access to the software (i.e. open release management). Many open and closed source projects adopted a multi-tier release strategy to benefit from the feedback of the alpha- and beta-testers that use unstable releases and pre-releases.

In the survey, 64% of respondents report to have an open issue tracking system, and 38% report to have a publicly-accessible software repository. 40% of respondents use mailing lists, or newsgroups to inform users, and only 26% report to have a completely closed development process. Respondents also express that users are involved in giving feedback and fostering discussions (82%), in providing bug reports and feature requests (85%), and even providing bug fixes (25%) (see Figure 2.5). 35% of respondents indicate that users are involved in manual testing, including GUI testing and combinatorial testing (e.g., different operating systems, Eclipse versions, or plug-in combinations). 12% report that users are even involved in automated testing.

2.8 Discussion

This section discusses how the new insights on the testing of plug-in-based systems can be used to better support the testing process, and outlines opportunities for

future work.

2.8.1 Improving Plug-In Testing

Since the community turns out to be vital in the testing process, a first recommendation is to make this role more explicit. This can be achieved by organizing dedicated “test days” (in line with Mozilla), or by *rewarding community* members who are the most active testers or issue reporters (e.g. at annual events). Additional possible improvements are a *centralized* place to collect *compatibility information*,²¹ and clear instructions on how downstream testers can contribute to the testing process in an ecosystem.

As an example, although downstream projects frequently execute upstream plug-ins as part of their own testing, at present, it is hard to tell if these executions are correct. Distributing plug-ins with a *test-modus* (e.g. to allow plug-ins to enable assertions), or to offer additional observability or controllability interfaces, would substantially leverage these executions. The test-modus could further report coverage information to a centralized server, informing the upstream plug-in provider about features, combinations, and configurations actually tested.

We think that to leverage plug-in specific testing, and facilitate test automation, *plug-in specific tool support* is needed. As an example, by means of dynamic and static analysis, test executions of plug-in systems can be visualized in order to provide information to the developer about the degree of integration between several plug-ins covered by a specific test suite. In (Greiler et al., 2010), we propose such a technique and introduce ETSE, the Eclipse Test Suite Exploration tool.

In general, we see a need for the research community to revisit current test strategies and techniques with respect to plug-in specific testing needs, in line with Memon et al. (2010) for component-based systems.

2.8.2 Open Versus Closed Source

Our study covers open and closed source development. In the 25 interviews, this did not seem to be a differentiating factor, both reporting similar practices and arguments.

In the survey, we can combine data on the project nature with specific test practices. One finding is that closed source projects have *less* test automation beyond unit scope. A possible explanation is that closed source projects work more with dedicated test teams, which rely on manual testing instead. This is consistent with the fact that closed projects report *more* user involvement for manual GUI testing (30% for closed versus 23% for open source projects).

²¹E.g., WordPress introduced a crowd-sourced “compatibility checker” plug-in for their plug-in directory <http://wordpress.org/news/2009/10/plugin-compatibility-beta>

Another visible difference is that closed source projects adopt plug-in specific integration testing approaches to address version or cross-product integration *less* often. A possible explanation is that closed source projects often aim to create full products (RCP applications) that are not intended for extension by others.

To discuss these differences in detail calls for additional research, which we defer to future work.

2.9 Credibility and Limitations

Assessing the validity of explorative qualitative research is a challenging task (Onwuegbuzie and Leech, 2007; Golafshani, 2003). With that in mind, we discuss the credibility and limitations of our research findings.

2.9.1 Credibility

One of the risks of grounded theory is that the resulting findings do not *fit* with the data or the participants (Bryant and Charmaz, 2007). To mitigate this risk, and to strengthen the credibility of the study, we performed *member checking* and put the resulting theory to the test during a presentation to approximately 100 developers, and during a birds-of-a-feather session at EclipseCon. Further, we *triangulated* our findings in the interviews with an online survey filled in by 151 professionals, which helped us to confirm that the main concepts and codes developed resonate with the majority of the Eclipse community. Although there was a possibility of *bias*, we assume we conducted an open-minded study which led to findings we did not expect. We closely followed grounded theory guidelines, including careful coding and memoing, and revisited both the codes and the analysis iteratively. We provide rich descriptions to give insights into the research findings, supported by a 60-page technical report (Greiler et al., 2011), and also provide the coded framework in Appendix A to increase transparency on the coding process. Threats to *external validity* (i.e. questioning whether the outcomes are valid beyond the specific Eclipse setting) are addressed in the following section.

2.9.2 Beyond Eclipse

Are our findings specific to *open source*? The compensation strategies identified certainly benefit from the open nature of Eclipse. However, the strategies themselves are not restricted to open source and can be applied in other settings (e.g. with beta-users). Furthermore, more than half of the 25 interviewees and the 151 survey respondents are working on closed source projects.

Our findings indicate a trade-off between test effort and tolerance of the community for failures in the field. In an open source setting, the community may be more tolerant and willing to contribute. In a closed source setting, it may take more organizational effort to build up such a community, such as with beta

testing programs. Note that for some application domains, there is zero tolerance for failure, such as with business- or safety-critical systems. Therefore, we do not expect our findings to generalize to such systems.

Another concern might be the *developer-centric* focus of Eclipse. For example, the *developer involvement* discussed in Section 2.7 assumes the ability to report and possibly resolve issues found in the plug-ins used. Note, however, that other findings, such as the test barriers covered in Section 2.6, are independent of whether the applications built are intended for developers. Furthermore, the Eclipse platform is also used to create a large variety of products for non-developers.

Clearly, the *plug-in-based* nature of Eclipse plays an important role, and is the center of our research. We consider a plug-in system as a specific form of a dynamic system with characteristics such as runtime binding, versioning, and combinability. For systems sharing such characteristics, we expect to find similar results. Further, most of the outcomes are independent of the specific plug-in architecture adopted. An investigation to exactly differentiate between various groups of dynamic systems is still an open issue as well as an excellent route for future work.

2.9.3 Beyond the People

A limitation of the current study is that it is based on interviewing and surveying people only. An alternative could have been to examine code, design documents, issue tracking system contents, and other repositories (Hindle et al., 2010; Zaidman et al., 2011). Note, however, that achieving our results with repository mining alone would be very hard as many test-related activities do not leave traces in the repositories. Furthermore, our emphasis is on understanding *why* certain activities are taking place. However, we see repository mining as an opportunity to further evaluate selected findings of our study, which we defer to future work.

2.10 Related Work

A few surveys have been conducted in order to reveal software testing practices (Ng et al., 2004; Garousi and Varma, 2010). Our study is substantially different. While these surveys focus on reporting testing practices, our study had the additional aim of understanding why certain practices are used or are not used. In a survey, researchers can only address a previously defined hypotheses. Our preceding GT study allowed first to emerge a theory about the testing practices, and to let the structure and the content of the survey follow from the theory.

As an implication, while other surveys concentrate on pre-conceived barriers to testing, such as costs, time and lack of expertise, we could address a much wider range of factors of an organizational and technical nature, as expressed by the participants themselves. Further, the GT findings drove the selection of test practices included in the survey. This allowed us to concentrate on facts specially

relevant for plug-in systems (reflected in a separate section of the survey), and in turn to omit questions such as generation of test cases or defect prevention techniques used in previous studies.

There is substantial research on analyzing different aspects of open source software (OSS) development. Mockus et al. (2002) analyze the Apache web server and the Mozilla browser in order to quantify aspects of OSS development (e.g. reported by Raymond (2001)). Raja and Tretter (2009) mine software defects and artifacts to understand several variables used to predict the maintenance model, which also leads them to several hypotheses on the effect of users participation. West et al. report on the important role of openness for community participation, and confirm that a modular software architecture decreases the barrier of getting started and joining an open source project (West and Siobhán, 2008). von Krogh et al. (2003) developed an inductive theory on how and why people join an existing open source software community.

Whereas those studies address open source, our findings apply to open and closed source software development. Furthermore, the focus of our study lies on software testing, a topic not covered in the earlier research.

Whereby research on configuration-aware software testing for highly-configurable systems (i.e. product lines) focuses on the combinatorial problems during interaction testing by detecting valid and invalid combinations of configuration parameters (e.g., by means of a greedy algorithm), our work reveals broader testing practices and problems during plug-in testing experienced in practice (Cabral et al., 2010).

2.11 Concluding Remarks

The main findings of our study are:

1. Unit testing plays a key role in the Eclipse community, with unit test suites comprising thousands of test cases. System, integration, and acceptance testing, on the other hand, are adopted and automated less frequently.
2. The plug-in nature has little impact on the testing approach. The use of extension points, plug-in interactions, plug-in versions, platform versions, and the possibility of plug-in interactions rarely lead to specific test approaches.
3. The main barriers to adopting integration testing practices include unclear accountability and ownership, lack of infrastructure for setting up tests easily, poor testability of integrated products, and long execution time of integration tests.
4. To compensate for the lack of test suites beyond the unit scope, the community at large is involved, by means of downstream testing, self-hosting, explicit test requests, and open communication.

These findings have the following implications:

1. The integration testing approach implicitly assumes community involvement. This involvement can be strengthened by making it more explicit, for example through a reward system or dedicated testing days.
2. Deferring integration testing to deployment calls for an extension of the plug-in architecture with test infrastructure, facilitating (e.g. a dedicated test modus) self-testing upon installation, runtime assertion checking, and tracing to support (upstream) debugging.
3. Innovations in integration testing, typically coming from research, will be ignored unless they address the barriers we identified.

While our findings and recommendations took place in the context of the Eclipse platform, we expect that many of them will generalize to other plug-in architectures. To facilitate replication of our study in contexts such as the Mozilla, Android, or JQuery plug-in architectures, we have provided as much detail as possible on the design and results of our study in the corresponding technical report (Greiler et al., 2011).

With this study, we made a first step to understand the current practices and which barriers exist when testing plug-in-based systems. In addition, this study should encourage the research community to facilitate *technology* and *knowledge transfer* from academia to industry and vice versa.

Acknowledgment

We would like to thank all participants of both the interviews and our surveys for their time and commitment.

What your Plug-in Test Suites Really Test: An Integration Perspective on Test Suite Understanding

Abstract

Software architectures such as plug-in and service-oriented architectures enable developers to build extensible software products, whose functionality can be enriched by adding or configuring components.¹ A well-known example of such an architecture is Eclipse, best known for its use to create a series of extensible IDEs. Although such architectures give users and developers a great deal of flexibility to create new products, the complexity of the built systems increases.

In order to manage this complexity developers use extensive automated test suites. Unfortunately, current testing tools offer little insight in which of the many possible components configurations and combinations of components are actually tested. The goal of this paper is to remedy this problem.

To that end, we interview 25 professional developers on the problems they experience in test suite understanding for plug-in architectures. The findings have been incorporated in five architectural views that provide an extensibility perspective on plug-in-based systems and their test suites. The views combine static and dynamic information on plug-in dependencies, extension initialization, extension and service usage, and the test suites. The views have been implemented in ETSE, the Eclipse Plug-in Test Suite Exploration tool. We evaluate the proposed views by analyzing eGit, Mylyn, and a Mylyn connector.

¹This chapter is published in the Empirical Software Engineering journal, November 2012. The authors of this publication are Greiler and van Deursen.

3.1 Introduction

Plug-in architectures are widely used for complex systems such as browsers, development environments, or embedded systems, since they support modularization, product extensibility, and run time product adaptation and configuration (Chatley et al., 2004; Marquardt, 1999; Mayer et al., 2003). A well-known example of such an architecture is Eclipse² which has been used for building a variety of extensible products, including a range of development environments for different languages (Shavor et al., 2005).

The size and complexity of software products based on plug-ins can be substantial. To deal with this, software developers rely on extensive automated test suites. For example, in their book *Contributing to Eclipse*, Gamma and Beck emphasize test-driven development of Eclipse plug-ins (Gamma and Beck, 2003). Likewise, the Eclipse developer web site³ describes the structure of the unit and user interface tests that come with Eclipse.

A consequence of systematic automated testing is the *test suite understanding problem*: Developers working with such well-tested plug-in-based architectures, face the problem of understanding a sizable code base along with a substantial test suite. As an example, the Mylyn⁴ plug-in for Eclipse comes with approximately 50,000 lines of test code. Developers responsible for modifying Mylyn, must also adjust the Mylyn test suite.

To address the test suite understanding problem, researchers have identified *test smells* pointing to problematic test code, *test refactorings* for improving them, and have proposed visualizations of test execution (Cornelissen et al., 2007; van Deursen et al., 2002; Meszaros, 2007; Van Rompaey et al., 2007). Most of the existing work, however, focuses on the *unit* level. While this is an essential first step, for plug-in-based architectures it will not reveal how plug-ins are loaded, initialized, and executed dynamically. As an example, just starting Eclipse loads close to one hundred plug-ins. Since these plug-ins do have interactions, looking at plug-ins in isolation yields insufficient insight in the way the dynamic plug-in configuration is exercised in test suites.

In this paper, we seek to address the test suite understanding problem beyond the unit level. Our approach includes the following steps.

First of all, in order to get insight in the nature of this problem, we interview 25 senior professionals from the Eclipse community on their testing practices. This study was set up as a *Grounded Theory* study (Adolph et al., 2011; Corbin and Strauss, 1990). The outcomes include a number of challenges professional developers face when confronted with complex test suites for the plug-ins they are working on.

Subsequently, to address these challenges, we propose a series of *architectural views* (van Deursen et al., 2004) that can help engineers understand plug-in inter-

²<http://www.eclipse.org>

³<http://wiki.eclipse.org/Eclipse/Testing>

⁴<http://www.eclipse.org/mylyn>

actions. These views are tailored towards the plug-in architecture of the Eclipse ecosystem. Thus, they support not only regular plug-ins as software composition mechanism, but also dynamic *extension-points*, through which a plug-in can permit other plug-ins to extend its functionality. Furthermore, they address the OSGi module system Eclipse is based on, as well as its service platform,⁵ which offers an additional extensibility mechanism based on *services*.

The five views that we propose to offer insight in these extension mechanisms are the *Plug-in Modularization*, the *Extension Initialization*, the *Extension Usage*, the *Service Usage*, and the *Test Suite Modularization* views. They will be discussed in full detail in Section 3.4. To construct these views, we deploy a mixture of static and dynamic analysis.

To evaluate the applicability of these views, we discuss their application to three open source Eclipse plug-ins (each built from various plug-ins). We analyze the eGit plug-in system⁶ permitting the use of the `git` versioning system within Eclipse, the substantial collection of plug-ins that comprises the Mylyn plug-in for work item management, and the Mylyn connector for the issue tracking system Trac.⁷

The paper is structured as follows. Section 3.2 provides the necessary background material on plug-in architectures. In Section 3.3, we present the findings of the interviews, which reveal the need for support during test suite understanding. Section 3.4 describes our approach, and covers the reconstructed architectural views. Section 3.5 discusses the architecture of our tool suite for reconstructing these views, after which we evaluate the views based on three case studies in Section 3.6. We reflect on the case study findings in Section 3.7, after which we conclude with a summary of related work, contributions, and areas for future research.

This paper is a substantially revised and extended version of an earlier paper (Greiler et al., 2010). The major changes include the addition of the analysis of information needs (Section 3.3), the addition of the service usage and test suite modularization views (Section 3.4), and a new case study based on Trac (Section 3.6).

3.2 Background: Modularization in Eclipse

Plug-in-based dynamic modularization systems are widely used to create adaptive and configurable systems (Chatley et al., 2004; Marquardt, 1999; Mayer et al., 2003). For Java, a well known example is OSGi,⁸ which offers a service registry, life cycle management, and dynamic updating.

⁵<http://www.osgi.org>

⁶<http://www.eclipse.org/egit>

⁷http://wiki.eclipse.org/Mylyn_Trac_Connector

⁸<http://www.osgi.org>

The Eclipse plug-in architecture⁹ is built on top of OSGi, through the Equinox¹⁰ implementation of the OSGi standard. Eclipse groups classes and packages into units, the so called plug-ins. Plug-in applications, like the well known Eclipse development environment, are composed from constituent plug-ins coming from different developers. We call the collection of all plug-ins forming a common application, including the plug-in architecture itself, a software ecosystem. A plug-in consists of code and meta data file, the manifest. The manifest describes, among others, the required and provided dependencies between plug-ins, and the plug-in version and author.

Plug-ins represent the basic extensibility feature of Eclipse, allowing dynamic loading of new functionalities. Plug-in P can invoke functionalities from other plug-ins P_i . At compile time, this requires the availability of the constituent plug-in's Java interfaces, giving rise to a *usage relation* between P and P_i .

A next level of configurability is provided by means of the *extension* mechanism, illustrated in Figure 3.1. Plug-in A offers an extension-point, which is exploited by B to extend A 's functionality. As an example, A could define a user-visible menu, and B would add an entry with an action to this menu.

An extension may be an executable extension contributing executable code to be invoked by the extended plug-in, a data extension, contributing static information such as help files, or a combination of both (Shavor et al., 2005). For executable extensions, a common idiom is to define a Java interface that the actual extension should implement, as shown in Figure 3.1.

A plug-in declares the extensions and extension-point it provides in an XML file. In addition, each extension-point can describe the expected syntactic descriptions of extensions by means of an optional XML schema file. From the extension declarations we can derive an *extension relation* from extensions to extension-points.

Last but not least, the Eclipse platform also uses OSGi *services* to allow loosely coupled interactions. OSGi services are objects of classes that implement one or more interfaces (The OSGi Alliance, 2011). These services are registered in the service registry under their interface names. Other services can discover them by querying the service registry, e.g., for the specific interface name. The registry returns a reference which acts as pointer to the requested service object. The two main mechanisms to provide and acquire services are either programmatically via a call to the service registry, or via a dependency injection mechanism (i.e., declarative services).

Even though at the moment, extension-points and extensions are still the dominant extension mechanism, OSGi services are becoming more and more important in the Eclipse architecture. Especially the next Eclipse platform version, codename e4,¹¹ bets on services to solve the problem of tight coupling within

⁹http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.htm

¹⁰<http://www.eclipse.org/equinox>

¹¹<http://www.eclipse.org/e4>

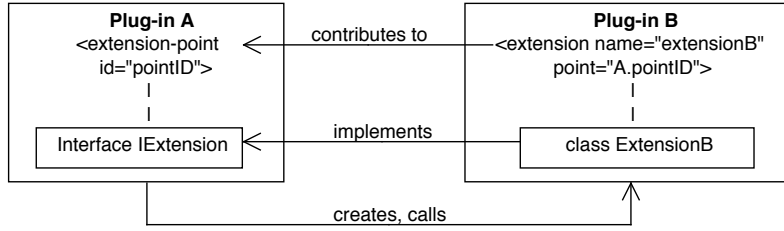


Figure 3.1: *The Eclipse plug-in extension mechanism*

the current Eclipse architecture. The e4 platform introduces a new programming model defining how plug-ins communicate beyond the extension mechanism. The introduced service programming models rely on three distinct parties, namely the service providers, service consumers, and a service broker. Using those, e4 defines a set of core services covering the main platform functionality.

Eclipse has explicit support for the testing of plug-ins, through its Plug-in Development Environment (PDE) and the corresponding PDE tests. PDE tests are written in JUnit, whereby execution of the test cases differs. A special test runner launches another Eclipse instance in a separate virtual machine and executes the test methods within that environment. This means the whole infrastructure (i.e. the Eclipse Platform API) is provided. Further, the developer can, beside the plug-ins under test, include and exclude various other plug-ins to be presented within the test environment.

3.3 Information Needs

In order to identify the information needs that developers have when working with plug-in test suites, we interviewed 25 Eclipse practitioners (henceforth 'Eclipse practitioners'). The information needs that emerged from these interviews are described in Section 3.3.3.

These interviews were conducted in the context of a much larger study, aimed at identifying test practices adopted in plug-in architectures. The general findings of that study are published elsewhere (Greiler et al., 2012a), and are only briefly summarized in the present paper (Section 3.3.2).

The full results we have available from this larger study form a rich empirical data set. In the present paper we report, for the first time, the findings on test suite understanding challenges specifically.

Domain	Project and/or Company
IDEs, Eclipse Distribution	Yoxos, EclipseSource
SOA	Mangrove, SOA, Inria
GUI Testing Tool	GUIDancer, BreDEX
Version Control Systems	Mercurial, InlandSoftware
Modeling	xtext, Itemis
Modeling	IMP, University of Amsterdam
Persistence layer	CDO
Domain Specific Language	Spoofax, Delft University of Technology
BPM Solutions	GMF, BonitaSoft
GUI Testing Tool	Q7, Xored
Coverage Analysis	EclEmma
Modeling	EMF, Itemis
BPM Solutions	RCP product, AndrenaObjects
Scientific data acquisition	OpenGDA, Kichacoders
Runtime platform	RAP, EclipseSource
Task Management system	Mylyn, Tasktop
Embedded Software	MicroDoc
RCP product	EclipseSource

Table 3.1: Domains, projects and/or companies involved in the interviews

3.3.1 Set-up Interviews

We conducted 25 interviews over Skype or telephone (each taking 1–2 hours) with selected professional developers from the Eclipse community. The participants are working on various well known Eclipse projects, as illustrated by Table 3.1. These include Mylyn and eGit, two projects we use as case study to evaluate the views presented in this paper as well. Almost all participants have been developers focusing on plug-in development and testing, except P3 and P10 who are both only involved in testing plug-in-based systems, as detailed in Table 3.2. Approximately half of the projects are open source and the other half closed source projects.

To structure the interviews, we composed a *guideline*, which we adjust after each interview, as our insight in the testing processes increases. The guideline comprises questions on the overall development setting, their general testing practices and then zooms in particular on integration testing techniques, and potential characteristics or challenges of testing plug-in-based system. In addition, we investigated which challenges Eclipsers are facing during understanding test suites.

The study followed a *Grounded Theory* design, a research method from the social sciences aimed at distilling theories from documents, interviews, and other qualitative data (Bryant and Charmaz, 2007; Corbin and Strauss, 1990; Glaser and Strauss, 1967). Grounded theory is increasingly used in software engineering research (Adolph et al., 2011), for example in the area of API documentation (Dagenais and Robillard, 2010), reviewing in open source projects (Rigby and

P	Role	CR	TS	Technology	KLOC
P1	developer	C	4-7	Eclipse plug-in	closed
P2	project lead	O	6	Eclipse plug-in	90
P3	tester	C	7-8	Eclipse plug-in, RCP product	370
P4	developer	O	3-10	Eclipse plug-in	90
P5	developer	C	3-7	OSGi	280
P6	project lead	O	6-9	Eclipse plug-in	1700
P7	project lead	O	2-5	Eclipse plug-ins	50
P8	project lead	O	12	Eclipse plug-in	670
P9	project lead	O	3	Eclipse plug-in	90
P10	test manager	C	20-50	Eclipse plug-in RCP product	closed
P11	developer	O	7-11	Eclipse plug-in	710
P12	project lead	O	1-2	Eclipse plug-in	12 & 56
P13	project lead	O	5-7	Eclipse plug-in	2000
P14	developer	C	5	RCP product	350
P15	project lead	O	20	RCP product	850
P16	developer	O	7-10	Eclipse plug-in	1500
P17	developer	C/O	5-6	Eclipse plug-in	2500
P18	project lead	C	4	RCP product	100
P19	developer	C/O	6-9	Eclipse plug-in	2500
P20	developer	O	7-10	RCP product	1000
P21	developer	C	4-10	RCP product	80-100
P22	developer	C	3-5	Eclipse distribution	140
P23	project lead	C	5-7	RCP product	closed
P24	developer	C	8	RCP product	400
P25	project lead	C	7-12	RCP product	closed

Table 3.2: *Participants involved (P: participants, CR: code repository (closed or open), TS: team size)*

Storey, 2011), and spreadsheet programming (Hermans et al., 2011).

3.3.2 Summary: Eclipse Testing Practices

During the interviews we asked Eclipsers about their testing practices. In summary, Eclipsers invest in testing their systems, and see testing as an essential task of the software engineering process. Nevertheless, unit testing is described as the predominant automated testing practices, whereas integration testing, GUI testing and system testing practices are less adopted.

On the other hand, interviewees express their belief that integration tests are especially essential for plug-in-based systems. They report on technical and organizational barriers for performing integration, GUI, and system testing practices. The main challenges are long test execution times, immature test tooling

or missing test infrastructure, high maintenance effort, as well as limited time for test design and execution (Greiler et al., 2012a).

Our interview results for Eclipse testing practices are also supported by literature. Gamma and Beck (2003) provide best practices for testing Eclipse, and, thus, for plug-in-based architectures, in general. Their book emphasizes test-first development of plug-ins. It does not focus on integration testing of plug-in systems. Guidelines for testing Eclipse¹² emphasize unit testing as well as user interface testing for which capture-and-playback tools are used.

The literature addressing OSGi testing focuses on the provisioning of the infrastructure required during the set-up of integration tests (Rubio, 2009). We have not been able to find test strategies for OSGi targeting integration testing of dynamic modularization systems in general, or plug-in systems in particular. Literature many Eclipse users are aware of and mentioned in the interviews is for example the book “Clean Code” by Martin (2008), which propagates the importance of clean and structured test code.

3.3.3 Test Suite Understanding Needs

During the interviews we asked the participants how easy or difficult the task of understanding test suites is, and which information would facilitate the task. Participants described two main scenarios to interact with the test code (i.e. understanding individual test cases and test suites), each implying different information needs. In the following sections, we will discuss the identified information needs and present excerpts of the interviews. A summary of all nine identified information needs (referred to as $N_1 - N_9$) is presented in Table 3.3. We will use those identifiers in the remaining of the paper to refer to the information needs.

Understanding Test Cases

Participants describe that understanding a particular and limited part of the test suite, i.e., a few test cases, is a common requirement during development. Either a failing test case has to be looked at, e.g., during bug fixing or refactoring, or a new test case has to be developed. This can be due to bug fixing, or feature development. The developer then has to read and understand only a specific test case he or she is pointed to, for example, a failing test, a bug identifier or a code reference. In this case, participants describe they do not need to understand the whole test suite. Some participants also describe they are completely unfamiliar with tests written by other developers, because their tasks only require understanding of their own code and particular test cases, and the ability to run the whole test suite. As participant P17 says: *“You do not have to understand the whole test suite. You just have to understand the part you are currently working on. And there are very many tests I have never seen, because I am not working on this part of the system.”*

¹²<http://wiki.eclipse.org/Eclipse/Testing>

Coding Standards, Naming Conventions. To understand one specific test case the developer needs to understand the source code of the test (N_1). The most essential requirement to understand source code is to have “good code”, as P11 outlines: *“It depends if it is easy [to understand tests]. Tests are also like other parts of code. Sometimes people give bad names to their methods and variables. Then it is hard to understand. For tests it is the same, but if you give good name, good comments, then it is easy to understand.”*

But also the format of a test case must be well structured to facilitate understanding of test code, as P17 reports: *“Tests have to be written similar to a work specification, like click here, execute that and check the result. And it should not be like 300 lines of test code. Then, nobody understands what’s going on. More like a step by step description, and it’s important to have good test names.”* P18 explains: *“We have a standardized form for test cases, and also naming conventions, that helps a lot. We also write tests in form of Given-When-Then, like described in the book ‘Clean Code’¹³ (Martin, 2008)”*.

Explanation and Motivation. Next to readable code, developers mention to need explanations and motivations for tests (i.e. why a test is needed or which requirements are checked by a certain test (N_2)). P7 explains what he thinks would facilitate test code understanding: *“You need a requirements document. [...] That is the starting point. To know what you want from the system. If I want an overview of what the test should be, then I need an overview of what the requirements are. So if you read now some of the unit tests, at the moment there is no motivation. It would say, e.g., ‘I test if this is commutative’, but why we test that is nowhere. So, there is no motivation why we test that, or explanation.”*

One countermeasure some Eclipsers mention is being careful in the way assertions are written. As P11 explains: *“we are trying to put assertions in which we explain well what we are doing.”* Still, also assertions might be hard to interpret and documentation might be needed. According to P12 the reason for a test (N_2) and what is addressed by a test (N_3) should be clear: *“What I think would be very valuable is to describe the scenario, maybe just in form of an in-line document. And describing what you actually test and why that should be tested. And especially with assertions, there you often have just a number, e.g., 15, and then, you should explain why it has to be like that.”* He adds: *“It happens quite often that you look at a test after some time has passed and actually you really cannot understand anymore what’s the scenario, or what is actually tested. That’s a problem, even for small projects, but more severe for larger projects.”*

But understanding single test cases might not be enough - practitioners might also be faced with the need of understanding the whole test suite. Then, different challenges are faced, which we discuss subsequently.

¹³This style is originally from domain-driven design (Evans, 2003).

Understanding Test Suites

Challenges. The second scenario involves comprehending the complete test suite in order to be able to assess quality and coverage of test scenarios. To master this task, developers need to understand which part of the system under test are not covered (N_4), which can be challenging as P14 explains: *“What one specific test does, that’s quite easy to understand. What’s difficult is to see where the blank spots on the map are.”*

Test suites can be quite complex and comprise different types of tests, as P10 describes: *“Even if we leave the unit tests out, we have some thousands of tests, which are GUI tests of different complexity, but also API tests for UDDI or JAXR, or other interfaces.”*

Understanding such test suites requires to get an overview of all test cases (N_5), as P7 explains: *“It is difficult. You have to read it all. Every method tests something. Every test method is sort of a definition of the semantics that it tests. If you skip one test, you do not know one part of the system. These tests are also only testing one plug-in, but [my component] itself has around 6 or 7 plug-ins.”*

Following P7, for plug-in systems it might not be enough to know the test suites of one plug-in or product. Eclipsers face the need of understanding the integration with several plug-ins/products and their test suites (N_6), as also P10 explains: *“If you know the product then keeping an overview of the test suites is not so difficult. But then, we sell several bundles of products, maybe ten different products together, that’s a different dimension. And those have to work together of course. This means you have to know the other products, and then the number of people that know what tests are doing is small. As a single person, to be familiar with all products, that’s a challenge and most people are not.”*

The results of this study show that understanding plug-in test suites is a complex and challenging tasks. P14 says: *“Comprehending the test suite gives us a big headache. And what we realized is that actually we have only two possibilities: either we work on a particular point, and we run only one test case, the one responsible for this piece of code, or we run all of them. We worry very often about not being able to run the test suite in a more fine-grained way.”*

Test Organization and Structure. Understanding the organization and structure of test code is often mentioned as an information need (N_7), and developers express that they are careful during organizing test code. Even though projects might have their own way of organizing test suites, it is common to categorize them according to the plug-ins they belong to, the system features they cover, or use cases they address. But there are also often correlations of tests to code and tests to issues reported.

In the words of P8: *“We have two different folders: one for all the actual test classes which test bigger features, and one folder for test cases sorted according to their Bugzilla number. There, the Bugzilla number is always in the test class name.”*

P19 outlines: *“Our tests have the same modular structure as our components. Normally, we combine 1 to 5 plug-ins in one component. And then we have for each component one test plug-in that comprises all tests for all 5 plug-ins.”*

Participants report that a clear structure of the test code, often following the (package) structure of the production code can facilitate the need to locate (find) test code (N_8).

Plug-ins and Extensions. During test execution of a PDE test, hundreds of plug-ins and their extensions are loaded. Keeping track of which plug-ins and extensions are currently active in a test environment is a challenging task (N_9), as P6 explains: *“The problem with extension-points is that if you start PDE tests with the workbench then you have to be careful that the workspace is in a good state. All kinds of projects, also if they are not on the class path of the test project, contribute to the extension-points and can create strange side effects and lead to surprises.”*

P19 expresses the need to understand how test suites and cases for foreign plug-ins test extensions, as he says: *“We also have dedicated extensions in the test plug-ins, whose only purpose is to be initialized. Or a particular behavior is implemented in those extensions, and then we test against this test behavior. All our tests are available in the CVS, and we recommend to developers who are developing an extension to look at those tests, because they also demonstrate how to use the APIs.”* A requirement to be able to investigate foreign tests for API understanding is to locate tests addressing certain extensions, services or plug-ins (N_8).

Nested Test Suites. It is also common that the test suites themselves are modularized, as P17 outlines: *“We have nested test suites, e.g., one version for version X of [plug-ins of the sub-product], and this test suite is then part of the test suite testing all versions of the [plug-ins of the sub-product], and this test suite is then part of the [product] test suite, and the [product] test suite is part of the test suite testing multiple products.”* Also P8 says: *“Tests are nested. But I have to say that JUnit is the most stupid technology existing. Inside it’s horrible, and the worst thing is that it uses reflection and the name of the class to instantiate the test, and because we use different scenarios, then we can not differentiate anymore. To get that running, we had to hack around a lot. It would be better to instantiate the test classes.”* The problem P8 describes is that when running nested test suites it is not obvious what is actually tested by which sub-test suite, and how the test environment has been set up (N_3 and N_9).

In summary, the interviews showed that test suite understanding is a cumbersome and complex task. Well-developed test code, standardized formats for test cases, and documentation can facilitate this task. Also test organization and structuring support test suite understanding.

Information Need - Test Suites		
ID	Need	P
N1	Understanding test (source) code	$P_{11,17,18}$
N2	Understanding the reason (requirements) for a test	$P_{3,7,11,12,19}$
N3	Identifying what is tested by a test, test plug-in and (assembled) test suites	$P_{8,11,12,14}$
N4	Identifying blank spots	$P_{3,7,10,14}$
N5	Getting an overview of test suites	$P_{7,10,14}$
N6	Understanding integration with other plug-ins	$P_{7,10}$
N7	Understanding test organization	$P_{8,10,12,13,18,19}$
N8	Locating test code	$P_{13,19}$
N9	Identifying what influences the test execution environment	$P_{6,8}$

Table 3.3: Distilled information needs

3.4 Models for Understanding Plug-in Test Suites

The interviews just presented demonstrate that Eclipse developers indeed face a test suite understanding problem. This problem can be partially addressed by regular support for program comprehension (such as dynamic analysis, (Cornelissen et al., 2009) software architecture reconstruction (van Deursen et al., 2004), or reengineering patterns (Demeyer et al., 2003)) as well as methods aimed at understanding unit test suites (Cornelissen et al., 2007; van Deursen et al., 2002; Meszaros, 2007; Van Rompaey et al., 2007). Furthermore, guidelines on how to set up (JUnit) test suites by, e.g., Martin (2008), Feathers (2004) or Freeman and Pryce (2009), will help to avoid and resolve overly complex test suites.

In this paper, we look beyond regular unit testing, and zoom in on the testing challenges imposed by plug-in architectures. To that end, we propose five architectural views.

The goal of the first view, the *Plug-in Modularization View* is to provide such structural and organizational awareness with respect to the code-dependencies of plug-ins. Equipped with this basic structural knowledge, the second step is the analysis of the extension relations between plug-ins and the way they are exercised by the test suite. This is realized through the *Extension Initialization View*. The *Extension Usage* and *Service Usage Views* complete the picture by providing the developer with insight in the way the test suite exercises the actual methods involved in the extensions and services. Finally, the *Test Suite Modularization View* helps to relate this information to the different test suites executed.

In this section we present these views, state their goal, and formulate the information needs they address. In terms of the Symphony software architecture reconstruction process (van Deursen et al., 2004), for each view we distinguish a *source model* corresponding to the raw data we collect, a *target model* reflecting the view that we eventually need to derive, as well as mapping rules between them. In what follows we present a selection of the meta-models for the source

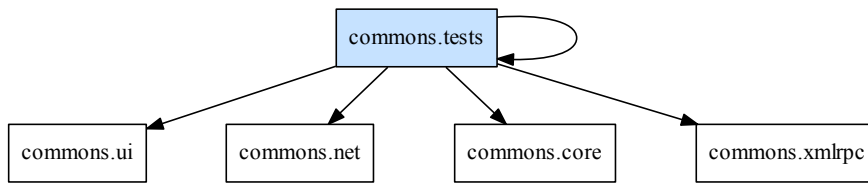


Figure 3.2: Static dependencies of test-component “commons.tests” in Mylyn

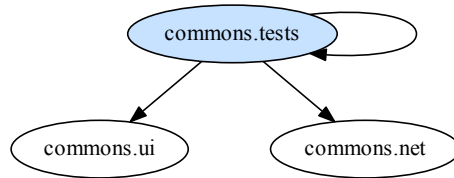


Figure 3.3: Dynamic dependencies of test-component “commons.tests” in Mylyn

and target models involved, as well as the transformation between them.

3.4.1 The Plug-in Modularization View

The *Plug-in Modularization View* that we propose is a simple way to provide insight in the static as well as dynamic dependencies between plug-ins and the test code. The developer can use this view to answer such questions as “which plug-ins are tested by which test-component?”, “where are test harness and test utilities located?”, and “which tests are exercising this plug-in?”. In the interviews, Eclipsers expressed that such information is essential for supporting test suite understanding (N_7). Also modularization capabilities of OSGi are often used to structure and organize test suites, e.g., create one test plug-in for several plug-ins. This view can help to understand how the different plug-ins depend on each other, and exemplify the structure of the system under test and the test plug-ins.

The static part of the view can be obtained through simple static analysis of plug-in source code and meta-data, taking the test suites as starting point. The dynamic dependencies are obtained by running instrumented versions of the code reporting all inter-plug-in method calls.

This view is illustrated for the test-component *commons.tests* of Mylyn showing its static code-dependencies in Figure 3.2 and its dynamic code-dependencies in Figure 3.3. On the left we see that *commons.tests* statically depends on four other plug-ins. The dynamic representation on the right side, reveals that only two out of those four plug-ins are actually exercised in a test run. It does not explain why this is the case (reasons could be that the test suite requires manual

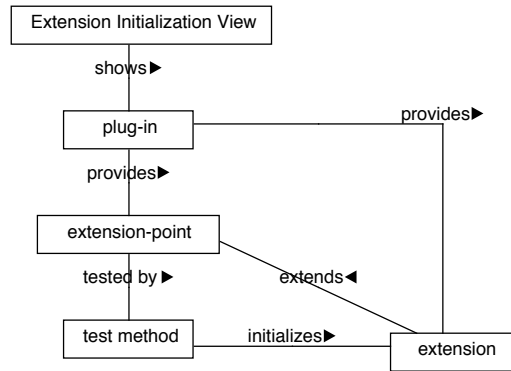


Figure 3.4: *Meta model of the Extension Initialization View*

involvement, or that a different launch configuration should be used), but it steers the investigation towards particular plug-ins.

3.4.2 Extension Initialization View

The *Plug-in Modularization View* just described provides a basic understanding of the test architecture and the code-dependencies between all test artifacts and their plug-ins. This is a prerequisite for the subsequent step of understanding the test suite from the more fine-grained extensibility perspective.

By means of this perspective, we will not only be able to tell which extensions and extension-points are tested in the current test suite (N_3), but we also gain insight in the system under test and its extensibility relations. For example, keeping track of which extensions are initialized during a test run is an information need expressed by P6 (N_9), which can be satisfied by this view. The meta model of this view is illustrated in Figure 3.4, by means of a UML class diagram.¹⁴ The view contains plug-ins, the extensions and extension-points they provide, as well as test methods that initialize the extensions. Extension initialization is the process of activating an extension (i.e. loading its classes). This differs from using an extension which means invoking a method of its classes.

The view helps answering questions on extensions and the way they are tested at system, plug-in, and test method scope. The main focus of the view is revealing which plug-ins in the system under test influence the functionality of each other through the extension mechanism, and which of those influencing relations are activated by a test suite.

System Scope. At system scope, the view gives insight in the extension rela-

¹⁴Drawn using UMLet version 11.3 (see <http://www.umlet.com/>)

tions present in the system under test, i.e., which plug-in contributes to the functionality of another plug-in. This is visualized in one graph, as shown in Figure 3.13 for one of our case studies. The graph presents the overall contributions of the systems, i.e., all extension-points and extensions within the system under test. In case plug-in A declares an extension-point and plug-in B provides an extension for it, the graph shows a link between the two nodes. The labels (fractions) on the links represent the number of statically declared extensions (denominator) one plug-in provides for the other, and the number of extensions that are actually used during a test run (numerator).

Plug-in Scope. Zooming in to the *plug-in level*, the view presents the relations of all extension-points declared by a given plug-in to existing contributions (i.e., extensions) contained in by the system under test.

This can be visualized, e.g., by means of a graph. An example is given in Figure 3.14, again for our Mylyn case study. The graph presents all involved plug-ins as ellipse-shaped nodes. Extension-points are represented as rectangles. Relations between an extension-point and a plug-in providing an extension are presented as edges. Extensions that are actually used during the test run are filled with a color. Thus, Figure 3.14 shows that 5 extensions are actually used, but that extension *tasks.ui* is not used. The view can also be used to show all extensions declared by the system under test for extension-points inside and outside the system under test. This means the view shows how the system under test influences the ecosystem during a test run, as shown in Figure 3.16.

Test Method Scope. At method scope, the developer can observe which test methods have invoked the code of an extension-point responsible for loading extensions, and which extensions have been created for it. For example, from Figure 3.15, the developer knows that test method “testShadowsStructureBridge()” triggers extension-point “mylyn.context.core.bridges” to load all available extensions. In this way, a developer or tester can identify the location of the test-code for a particular extension-point.

Underlying Meta-Models

This view is based on static meta data and dynamic trace information. The meta data comes from the mandatory XML file, and from the optional XML-schema file (see Section 3.2).

The trace used for this view comprises “extension initialization events” during the test run. The underlying trace data follows the meta model shown in Figure 3.5, which is also used to derive dynamic information for the other views. An “extension initialization event” is recorded before a method named “createExecutable()” is called. In the Eclipse Platform, this method is used to create the

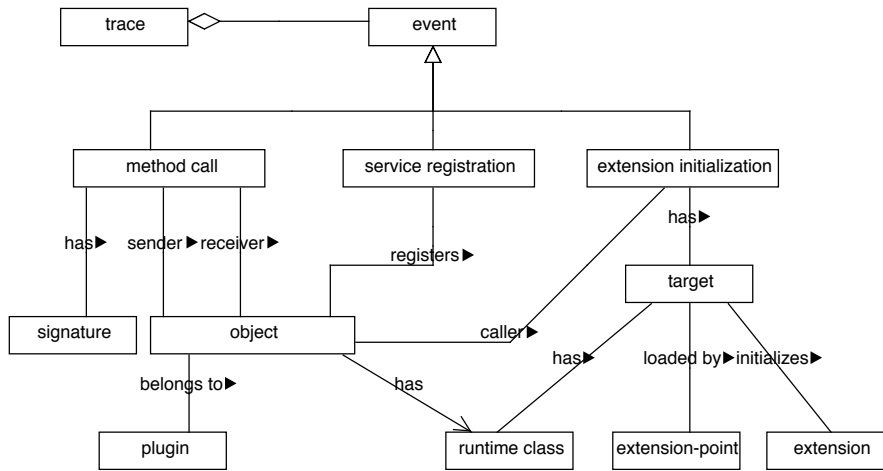


Figure 3.5: Trace meta model

extension from a given class, passed as parameter. This also is the point we intercept to trace the caller of this method and the target-object, by means of an aspect.

This trace data shows only the initialization of an extension. It does not show the usage of this extension, which would be the invocation of a method of the class of the extension.

Reconstructing the View

The data behind this view comprises the static meta data files for extension and extension-point declaration, and the information gained by tracing the creation of extensions during a test run.

The dynamic trace comprises only executable extensions, because only those are created by the method we trace. An alternative to include also data extensions is to intercept not the creation of an extension, but the look-up of extensions from the plug-in registry. We decided against this approach for two reasons: first, the views would become more complex. Second, data extensions, i.e., extensions that enhance the system only with static data, are less interesting from a testing perspective.

Thus, before we can compare the static and dynamic data sources, we have to know which extensions are data extensions, and which extension-points load only data extensions. An executable extension has to state at least one class in its meta data file, used to instantiate the extension. Thus, to determine the type of an extension we analyze the presence or absence of classes in the meta data

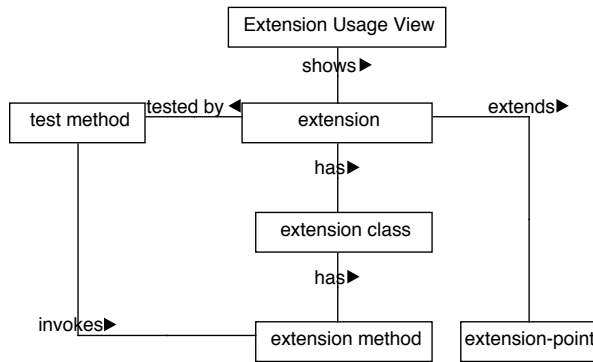


Figure 3.6: Meta Model of the Extension Usage View

file.

An extension-point, on the other hand, states the class an extension has to be based on in the XML-schema file. We analyze these schemes to retrieve the attributes defining the base class. However, an XML schema is not mandatory. If it is missing, we try to find an actual extension for the extension-point. If that extension contains a class, we conclude that the extension-point is executable, otherwise it is a data extension-point. If we cannot find an extension we classify the type of the extension point as unknown.

The remaining data can be filtered and grouped, to show which extensions have been created, by which extension-points, and which test method is involved. The underlying data also exposes information about the usage of an extension. To take advantage of that, the *Extension Usage View* is introduced in the following.

3.4.3 Extension Usage View

The *Extension Usage View* focuses on characterizing the usage of an extension during the test run. The goal of this view is to give the developer or the tester an understanding of how the integration of the extensions has been tested (N_6). The question it addresses is “which extensions have been actually used during the test run, and when and how have they been used?”

The target meta model of the *Extension Usage View* is illustrated in Figure 3.6. In this view, extensions are referenced by their name. Extensions are furthermore related to the extension-points they target, and to the test methods exercising them. Recall from Figure 3.1 that extension-points can declare types (interfaces or classes) that are implemented by actual extension classes.

The *Extension Usage View* can be used at system, extension, and usage level. On *system scope*, we can gain detailed information about which of the declared

extensions have been actually used during a test run, and how many of the test methods are associated with extension usages. Using an extension means to invoke a method of the extension class, overwritten or inherited by the type declared at the extension-point. For example, from Figure 3.17 we can see a list of extensions that have been used during the test run (left side).

Zooming in to the *extension scope*, the developer can see which test methods have used a given extension. For example, on the right side of Figure 3.17, we can see that the extension “mylyn.tasks.ui” has been used during the execution of four test methods. This information is helpful to spot the right piece of code responsible for the extension usage, e.g., to enhance or change it.

A refinement of this view to the *method scope* shows how the extension has been used during the test run, for example illustrated by the pop-up window in Figure 3.17. Here, all methods of an extension that have been called during testing are listed.

With these views, the tester gains knowledge about which integrations of extensions have been tested (N_3), and can locate test code responsible for the usage of an extension (N_8). This helps in understanding the usage of the extension and its API, which P19 has identified as an important task.

Underlying Meta-Models

The execution trace used to construct the *Extension Usage View* is the same as the one used for the initialization view. It comprises detailed method calls of a test run, as shown in Figure 3.5.

We trace all public calls directed to the system under test. For each extension, we calculate all types that the extension is based on and that are declared by the extension-point, as explained in the next subsection. Subsequently we trace all method calls to these types. Since we trace dynamically, extension calls can be resolved to the actual objects and methods executed.

Reconstructing the View

To construct this view, we need in addition to the dynamic data discussed before, all the methods an extension can implement. Those methods can be used by an extension-point to invoke it. We will refer to this set as to the *extension method set*. Therefore, the extension-point has to define a specific base type (e.g. a class or an interface), which can be extended by an extension. To give a simple example, let us look at Listing 3.1. Here, the class *Extension* represents the base type of an extension. This class is defined within the plug-in providing the extension-point. Also within the plug-in defining the extension-point, the code is located which is in charge of invoking all loaded extensions, as illustrated by the method *invokeExtensions()* of class *ExtendsionUsage*. A plug-in which wants to provide an extension for this extension-point has to extend the base class, as done by classes *B* and *C*. Those classes can be part of another plug-in.

Listing 3.1: *Extension Usage Example*

```

abstract class Extension{
    void me();
}

class ExtensionUsage{
    void invokeExtensions(Extension [] extensions){
        for(Extension e : extensions)
            e.me(); }
}

class B extends Extension {
    void me() {}
    void mb() {}
}

class C extends Extension {
    void me() {}
}

```

An extension-point often uses the meta data files (i.e. the plugin.xml) to specify which type it expects. But, Eclipse does not force an extension-point to declare formally the base type, which means we might have to derive our extension method set based on a heuristic. Our heuristic works as follows.

First, in case the extension-point formally declares a base type for an extension, the algorithm uses this to derive recursively all methods defined by it and its super-types, i.e., interfaces and ancestors. This collection represents the extension method set. For our example in Listing 3.1, the method set comprises just method *me()*.

In the case no base type is provided, the algorithm collects all the classes a given *extension* declares from its meta data file. Starting from these types, the algorithm recursively derives all super-types of these classes. Note, however, that not all of them might be visible to the extension-point. For example, consider a class *A*, defined in plug-in *Pa*, that extends class *E*, defined in plug-in *Pe* and implements Interface *I* also defined in *Pa*. Since no declaration of a base class is provided, the algorithm has to decide whether *A* is based on *I* or *E*. This example is illustrated in Figure 3.7.

The algorithm classifies types as visible for the extension-point if they are declared outside of the plug-in providing the extension. Contrary, a type is considered as invisible when declared within the plug-in of the extension. Those are excluded from the type set. Applying this to our example reveals that the base class has to be *E*.

If the extension and the extension-point are declared in the same plug-in all types are considered relevant. This results in an conservative heuristic, i.e., it cannot miss a relevant type, but might include too many. From the resulting set

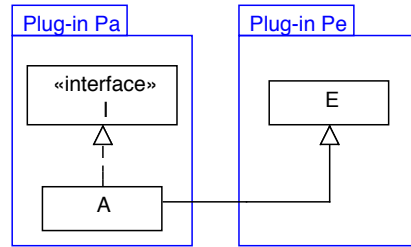


Figure 3.7: *Deriving an Extension Base Type*

of types the extension method set can be derived.

Applying this algorithm to the example of Listing 3.1 reveals that, in case class *B* is defined within another plug-in, method *mb()* will not be visible to the extension-point, and is therefore excluded from the extension method set. In case class *B* is defined within the plug-in defining also the extension-point the algorithm will declare class *B* as a potential extension class and include methods *me()* and *mb()* in the extension method set.

Finally, the trace is inspected for calls made to methods included in the method set. Only when the traced runtime-class corresponds to the class of an extension, the call is considered as an actual usage in a particular test method.

Based on this analysis, the view shows for every extension which test methods have caused their usage, and which methods out of the extension method set have been used.

3.4.4 Service Usage View

The Eclipse plug-in architecture builds on top of OSGi. Especially in the new e4 version of the Eclipse platform, OSGi services are an important extensibility mechanism. Services are a means to decouple plug-in interactions, and allow interchangeability of service providers, which also improves testability of the system. The *Service Usage View* helps in understanding which services are used during test execution (N_3). It helps answering questions like “which plug-ins acquire or register which services?”, and “how are these services used during test execution?” The meta model of this view is illustrated in Figure 3.8. A concrete example of this view for the usage of the service “IProxyService” is given in Figure 3.18. From this view, it is apparent that this service was used during the execution of five test methods.

This view also makes explicit which concrete instantiation of a service is used and invoked during the test run. This is important information in order to determine the configuration of the test environment or to configure the test environment correctly, which is a challenge P6 pointed out (N_9).

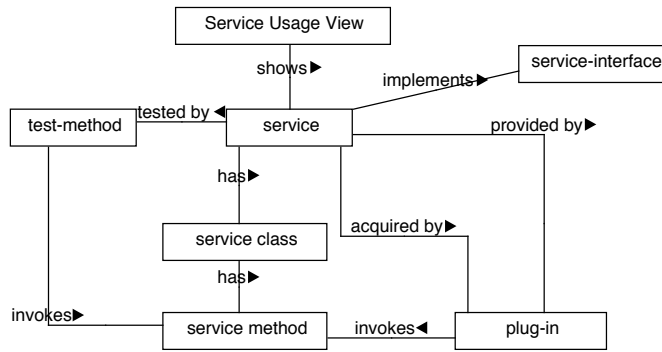


Figure 3.8: *Meta Model of the Service Usage View*

Underlying Meta-Models

OSGi services can be registered and obtained either programmatically (see Listing 3.2 for some examples) or by using dependency injection defining services in an XML-file (i.e., declarative services). To obtain the static data representing service registration and acquisition, we analyze the plug-in's byte code for service registration or service acquisition, as well as the meta data for the use of declarative services.

The dynamic data required is provided by two execution traces. First, one trace covers method call events, as described in Section 3.4.3. Second, service registration events, as illustrated in Listing 3.2, are traced.

Reconstructing the View

To construct this view, we need in addition to the dynamic execution trace data, the method set of a service which can be invoked by a service user. We will refer to this set as to the service method set. Determining this service method set is easier than determining the extension method set, since we always know the base type for a service. From this type on, we recursively derive all super-types of this base type, as discussed before.

All methods of this set of types contribute to the service method set, which is used to analyze the trace for service usage. During the analysis of the byte code for service registration, only the base type, e.g., the interface a service implements, might be known, while the runtime type is unknown. Therefore the runtime type of the service registered is determined by tracing the registration events during runtime. Only when the traced runtime-class corresponds to the class of a registered service, the call is considered as an actual usage in a particular test method.

Listing 3.2: *Excerpt programmatic service registration and acquisition*

```
/* Service Acquisition */
public void getService(BundleContext c){
// 1) Direct Acquisition of a service
ServiceReference serviceReference = c.getServiceReference(IProxyService.class.getName());
IProxyService service = (IProxyService) c.getService(serviceReference);

// 2) Via a ServiceTracker
ProxyServiceTrackerCustomizer custom = new ProxyServiceTrackerCustomizer(c);
ServiceTracker serviceTracker=new ServiceTracker(c, IProxyService.class.getName(), custom);
serviceTracker.open();
...

/*Service Registration */
public void registerService(BundleContext c){
IProxyService service = new ProxyService();
c.registerService(IProxyService.class.getName(), service, null);
...
}
```

Based on this analysis, the view shows for every service, which plug-ins registered or acquired this service, and which test methods have caused a service usage, as well as which methods out of the service method set have been used.

3.4.5 The Test Suite Modularization View

For complex Eclipse plug-ins, it is common practice to assemble individual test cases into different test suites. Examples are suites for specific versions of required plug-ins or external services, fast test cases run during development, more time consuming test cases depending on the user interface (workbench) or network access, and so on. These suites are often assembled programmatically, and sometimes involve the creation of different fixtures in order to run the same test cases under different circumstances.

The *Test Suite Modularization View* aims at clarifying how test cases are grouped into (nested) test suites at run time. It maps assembled test suites to (1) the test plug-ins that contribute test cases; and (2) plug-ins under test. It helps answering questions like “which plug-ins contribute tests to this particular test suite?”, “which plug-ins are tested by this test suite?”, and “which extensions, extension-points and/or services get addressed by this test suite?”. The meta model of this view is illustrated in Figure 3.9. This view helps the developer to choose the right test suite to execute, to understand which entities are tested by a particular test suite, or to assemble a new, customized test suite addressing the right plug-ins of the system, and satisfies information needs N_3 , N_7 and N_8 expressed in Section 3.3.3.

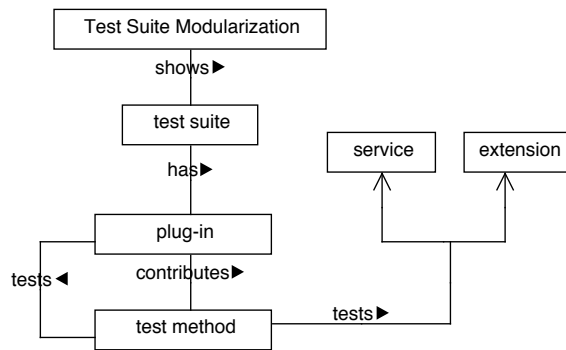


Figure 3.9: *Meta Model of the Test Suite Modularization View*

Underlying Meta-Models

This view is based on static meta data and dynamic trace information. The meta data comes from the plug-in manifest files of the plug-ins, the mandatory XML file for extension and extension-point definition, from the optional XML-schema file (see Section 3.2), the XML-definitions for declarative services, as well as from the analysis of the byte code for service registration or acquisition.

The dynamic data comes from two traces. First, a trace comprising method calls during the test run, and second, the trace comprising “service registration events” as illustrated by the trace meta model in Figure 3.5.

Reconstructing the View

To reconstruct this view the static meta data and trace data is combined, and the algorithms already discussed, e.g., to derive the extension or service method sets, are used. Then grouping of the data takes place to reveal which plug-ins contribute test cases to the test suite, which plug-ins have been executed during the test run of this test suite, and which extensions and services have been used.

3.5 Implementation and Tool Architecture

We implemented the reconstruction and presentation of our views in ETSE,¹⁵ the *Eclipse Test Suite Exploration Tool*. It is implemented in Java, offers an API to construct the views in question, and a graphical user interface which is implemented as Eclipse extension, which integrates the tool in the Eclipse IDE.

ETSE consists of three logical modules: a module dedicated to information gathering, a module responsible for knowledge inference and a module responsible

¹⁵ETSE is available at <http://swerl.tudelft.nl/bin/view/Main/ETSE>

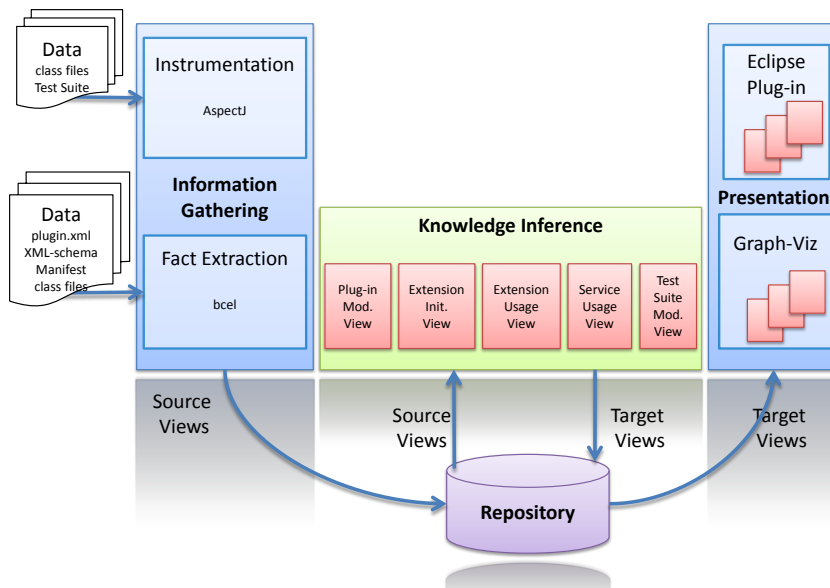


Figure 3.10: *ETSE Architecture*

for the presentation of the views, as shown in Figure 3.10.

Module: Information Gathering. This module is responsible for gathering static meta data as well as for gathering execution traces during test execution. To analyze the static Java code we use the Byte Code Engineering Library,¹⁶ which inspects and manipulates the binary Java class files. Meta data, including the OSGi manifest, the service definitions and the plugin.xml files, is collected and analyzed. The user can instruct ETSE which test suite and which system under test should be examined by using the “Configuration tab” provided by ETSE’s user interface. To trace the execution of the test run, we use aspect-oriented programming, in particular the AspectJ¹⁷ framework. Because we do not want to intercept plain Java applications, but Equinox, the Eclipse implementation of OSGi, we are using the Equinox aspects framework.¹⁸ This provides load time-weaving of advices, i.e., when a class is loaded by OSGi. There are four main groups of aspects that can be differentiated: the aspect used for weaving into the initialization of the extensions, the aspect used to trace method calls, the aspect used to trace plug-in starts and stops, and the aspect used to trace registration and acquisition of OSGi services. All the analyzed data is finally stored as source views, in a format similar to the meta model illustrated in Figure 3.5, in a repository.

¹⁶<http://jakarta.apache.org/bcel>

¹⁷<http://www.eclipse.org/aspectj>

¹⁸<http://www.eclipse.org/equinox/incubator/aspects/equinox-aspects-quick-start.php>

Module: Knowledge Inference This module uses the data gathered during information gathering, and transforms the source views to the different target views, among those the *Plug-in Modularization*, *Extension Initialization*, the *Extension* and *Service Usage* Views, and the *Test Suite Modularization* View. These transformations implement the algorithms presented in this paper.

Module: Presentation The presentation module is used to visually represent the target views to the user of ETSE. Target views can be saved as comma separated value files, which a user can later visualize ad libitum. Also ETSE can visualize those files for the user. First, ETSE allows users to store the target views in the dot-graph format, which then can be visualized by Graphviz,¹⁹ a graph visualization package. Furthermore, ETSE is integrated in the Eclipse IDE, where it provides a graphical user interface allowing the user to interact easier with the tool. Within this paper we show visualizations based on dot-graphs, as well as visualizations rendered by ETSE's user interface within Eclipse. A screenshot of the ETSE Eclipse integration showing the *Extension Initialization View* is provided in Figure 3.11. Users can for example navigate between views, or define different level of detail for each view, as detailed below.

Navigation between scopes ETSE presents each *architectural view* in a separate tab within an Eclipse “view”. The user can easily switch between the *architectural views* by activating the desired tab. Within each tab, the user can navigate between the several scopes discussed in this paper (i.e., system, plug-in, extension, service or method scope). For example, in the *Extension Initialization View*, the user can switch between the plug-in or the system scope by activating a radio button. In the *Extension* and *Service Usage View*, the user will first see a list of all the extensions respectively service that have been used during a particular test run on the left side (i.e. system scope). By selecting an extension/service from the list, all test methods which have triggered a use of that particular extension/service are shown on the right side (i.e. extension resp. service scope). The user can further zoom in on method scope by clicking on a particular test method. This will cause a pop-up window to appear and to show which methods of the selected extension/service have been used during execution of the selected test method. All scopes of this view are illustrated in Figure 3.17. Further, the user can also request to see the source code of the test method by left-clicking on the test method. Then, the Java class comprising the test method is opened and visualized within the editor.

¹⁹<http://www.graphviz.org>



Figure 3.11: Screenshot of the ETSE Eclipse integration showing the Extension Initialization View

3.6 Evaluation

We evaluate the proposed architectural views with respect to *applicability*, *scalability*, and *accuracy*. This leads to the following research questions:

RQ1: *Which information about the test suite and system under test can be obtained by the proposed views and to which extent does the information provided by the tool address the information needs identified?*

RQ2: *To what extent do the views scale to large systems?*

RQ3: *To what extent are the views a correct representation of the system under test?*

Our evaluation is explorative in nature, aimed at generating an understanding about the applicability of the proposed views. Therefore, the evaluation has been set up as a case study involving three (open source) systems, to try to answer our research questions.

3.6.1 The Subject Systems

One experimental subject is eGit, a plug-in system designed to integrate the source control management system Git into Eclipse. The eGit system is a good fit for our evaluation, mainly because it is a relatively new system under active development, which uses also the new Eclipse technologies (e.g., services). In the last year (i.e., 2011), it grew from around 30,000 to nearly 100,000 lines of code, and from 1,700 to 14,000 lines of test code. eGit consists of five main plug-ins, and two test plug-ins. We investigated the four main test suites: The *All-Tests* suite executing 95 test cases and located in the `egit.core.test` plug-in. The *All-JUnit-Tests* suite, executing 23 test cases, the *All-Non-SWT-Tests* suite, executing 62 test cases, and the *All-SWT-Tests* suite executing 129 test cases. The latter ones are all located in the `egit.ui.test` plug-in.

The other study subject is Mylyn, a task management system for Eclipse. Mylyn has been chosen because it is a large-scale plug-in system, and gives valuable insights on the ability of the views to help comprehending such a complex system, as well as to the scalability of the views. We used Mylyn 3.4 for Eclipse 3.5. The Mylyn core comprises 27 plug-ins, which come with 11 test components. Additional contributions, like connectors, apart from the Trac connector discussed below, are excluded from this study. The source code under study comprises 200,000 lines of code, and the test suite has 30,000 lines of code. We investigate the included *AllComponents* test suite which runs 518 test cases, and the *AllHeadlessStandaloneTests* test suite running 92 test cases.

The last subject system is a Mylyn connector for the issue tracking system Trac. We choose the Trac-Mylyn connector for two reasons: First, it is of 8,500 lines of code and 3,400 lines of test code, a quite small, but well-tested plug-in system that permits, in addition to the investigation by means of the views,

View	Information Needs addressed by View	
ID	Need	Questions addressed
PMV	N3	Which plug-ins or packages are tested by this test plug-in?
	N4	Which plug-ins or packages are not tested?
	N7	Which tests address this plug-in? Where are test utilities located?
EIV	N3	Which extensions are loaded?
	N4	Which extensions are not loaded?
	N5	On system level, how is the integration of plug-ins tested?
	N6	How do plug-ins of the system under test interact with each other?
	N8	Which test method causes the extension-point to load extensions?
EUV	N9	Which extensions might influence the test execution environment?
	N3	Which extensions or which extension methods are invoked?
	N4	Which extensions or extension methods are not used?
	N5	How many extensions are used during a test run? How many are missed?
SUV	N8	Which test method triggers a use of this extension?
	N3	Which services or methods of a service are invoked?
	N4	Which services or method of a service have not been tested?
	N5	How many services have been registered or used? Which have not?
	N8	Which test method invokes or registers this service (method)?
TMV	N9	Which concrete services are used?
	N3	Which plug-ins, extensions or services are addressed by this test suite?
	N7	Which plug-ins contribute tests to this particular test suite?
	N8	In which test plug-in is this particular test located?

Table 3.4: *Distilled information needs; Views: Plug-in Modularization View (PMV), Extension Initialization View (EIV), Extension Usage View (EUV), Service Usage View (SUV), Test Suite Modularization View (TMV); What is tested (N3), Blank spots (N4), Overview (N5), Integration (N6), Structure (N7), Location (N8), Environment (N9).*

manual inspection of the complete system. Second, we choose it because it is referred to in the Mylyn Contributor Reference²⁰ as the system to look at, in case a new connector for Mylyn should be developed. The Trac-Mylyn connector consists of three plug-ins and one test plug-in.

We analyzed all three subject systems completely by tracing and generating all views with ETSE, and investigated each view also on all different abstraction levels. Within this evaluation, we outline and discuss mainly the Mylyn system, as it represents the most challenging system (because of the size) for our techniques. Most of the views illustrated for Mylyn are equally good for the other two subject systems. In case the analysis of one of the other two subject systems yields different results, we present these deviations within this section.

3.6.2 RQ1: Applicability and Information Needs

In this section, we investigate which information about the test suite and system under test can be obtained by the proposed views and to which extent does the information provided by the tool address the information needs identified.

²⁰http://wiki.eclipse.org/Mylyn/Integrator_Reference#Creating_connector_projects

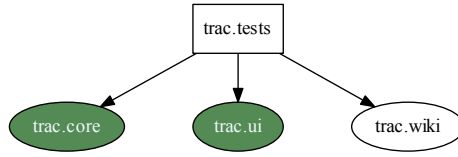


Figure 3.12: Plug-in Modularization View for Trac

Answering RQ1: In summary, the five proposed views satisfy many of the information needs identified concerning test suite understanding. They can help to understand and investigate test code and the system under test from a top-down approach. The views help to understand what (i.e. plug-ins, extension-points, extensions and services, as well as their methods) has been tested (N_3), and what has been left out (N_4). They provide an overview of the test suites (N_5), highlight the integration with other plug-ins (N_6), shed light on the test organization (N_7) and the configuration of the test execution environment (N_9), and help to locate test code (N_8). On the other hand, the views are not suited to investigate the system from a bottom-up approach, i.e. start with a single test case. Information needs such as understanding source code (N_1) or the reasons behind tests (N_2) are not covered by these views. The relations between views and information needs are summarized in Table 3.4.

The following subsections provide a detailed evaluation of each view. We do so by going through the use of the views for Mylyn followed by a reflection on the strengths and weaknesses of the views. Since Mylyn uses only one service, the Service Usage View will be explained by looking at the eGit system.

Plug-in Modularization View

The *Plug-in Modularization View* aims at providing a first high-level overview of dependencies at the top-most level of plug-ins. An example of a plug-in modularization view was shown in Figures 3.2 and 3.3 for the Mylyn test component. It shows the four plug-ins the Mylyn test component statically depends on, as well as the two that are actually executed during testing. A similar view is shown in Figure 3.12, displaying which of the Trac plug-ins are involved in testing. These views highlight which plug-ins a test plug-in depends on statically and also which of those are actually invoked during a test run (N_3 , N_4). This information can be valuable to understand the structure and organization of test as well as production code (N_7). Structure of test and production code plays an role during test suite understanding (see Section 3.3.3).

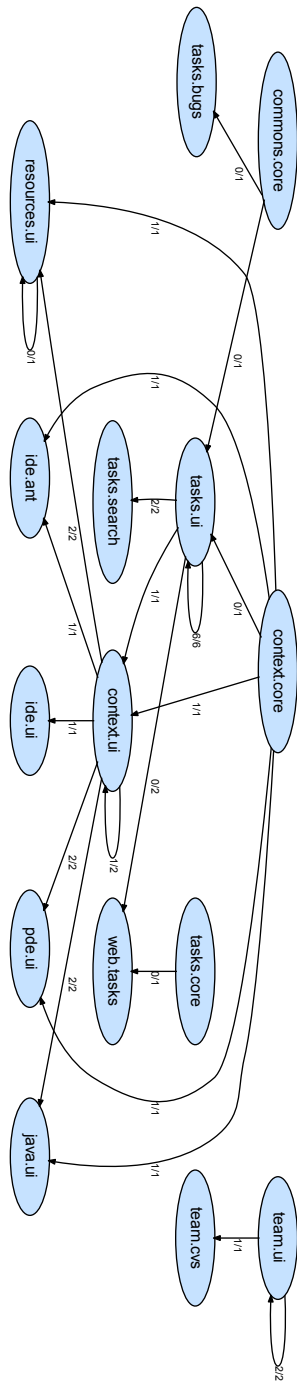


Figure 3.13: *Extension Initialization View on system scope showing static and dynamic dependencies based on extension-points*

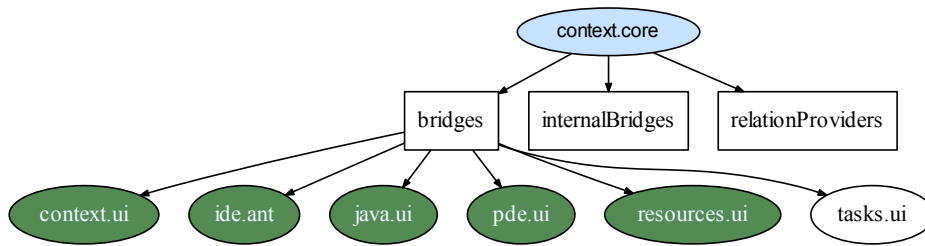


Figure 3.14: *Extension Initialization View on plug-in scope based on extension-points*

Extension Initialization View

By means of the Extension Initialization View, we get an overview of the system under test (N_5) and of how the integration of several plug-ins has been tested (N_6). We see that the 27 plug-ins in Mylyn offer 25 extension-points to contribute functionality, and also that they declare 148 extensions to enhance their functionality and that of Eclipse. Furthermore, we can use this view to understand how the 148 extensions are related to the 25 extension-points within the system under test, and which of those relations have been covered by the test suite.

This view at system scope for Mylyn is illustrated in Figure 3.13. ETSE allows to remove common prefixes from the plug-in names to improve readability, as we did with “org.eclipse.mylyn”. An edge between two plug-ins means that one plug-in declares an extension-point for which the other plug-in provides an extension. Only plug-ins with actual extension relations, which means that a plug-in exists providing an extension-point and another one using it, are shown, reducing the number of nodes to 15. From this representation of the system, it is apparent which plug-ins influence each other, and also which of those relations have been actually addressed by the test suite (N_3), and which have been left out (N_4). The view abstracts from the specific extension-points declared. The fraction on the edge states how many of the static declared extensions (denominator) are activated during a test run (numerator).

At plug-in scope, this view for plug-in *mylyn.context.core* is illustrated by Figure 3.14.²¹ The plug-in provides three extension-points, namely *bridges*, *internalBridges* and *relationProviders*. The view shows that within Mylyn six plug-ins exist that use extension-point *bridges* to influence the plug-in, represented by the six nodes connected to this extension-point. The coloring of five nodes indicates that only five of the relations are activated during the test run. The view does not give explanations, but points to one plug-in the developer might manually inspect and find an empty XML declaration for this extension.

The developer can also zoom at method scope, as illustrated by Figure 3.15. This view reveals which test method causes this extension-point to load its ex-

²¹ETSE can also export graphs as dot-files, which can then be visualized with GraphViz.



Figure 3.15: *Extension-Initialization View on test method scope*

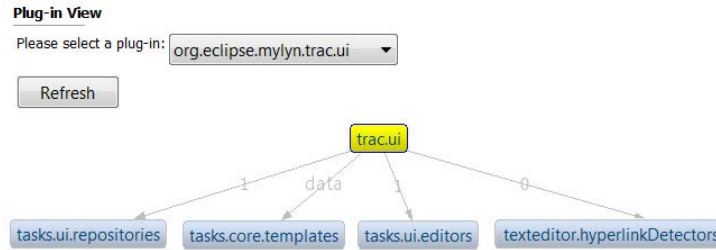


Figure 3.16: *Extension Initialization View for extensions of Trac on plug-in scope*

tensions, and can be used to locate test code (N_8).

The *Extension Initialization View* serves to show how plug-ins affect each other's behavior. The present view does not show how the system under test is influenced by its ecosystem, i.e., the Eclipse platform. Nevertheless, the borders defining the system under test can be chosen by the viewer, thus allowing the developer to include parts of Eclipse he or she is interested in. Also for smaller systems, like the Trac connector, this view is helpful, as it shows how the system enhances (extends) the ecosystem during the test run. For example, Figure 3.16 shows that plug-in “trac.ui” has four extensions enhancing the Eclipse ecosystem, out of which two extensions are initialized during this test run, and one extension is a data extension.

The *Extension Initialization View* visualizes information about the integration of plug-ins within the system under test (N_5 , N_6), the coverage of these integration points during test execution (N_3 , N_4), as well as about the configuration of the test environment (N_9). This covers several information needs about *plug-ins and extensions* raised in Section 3.3.3. For example, this view can answer the question of P6, who wants to know which plug-ins in the system under test can influence the functionality of other plug-ins. On *method scope* this view also allows to locate test code addressing (N_8) a given extensions and thus facilitating understanding of *test organization and structure*.

Extension Usage View

The Extension Usage View helps to understand how the test suite addresses integration with respect to extensions. It gives an overview of how many extensions have been used during the test run (N_5), and whether extensions have been only initialized or whether there has been a method invocation (N_3 , N_4). For example, from Figure 3.17, the developer can see that in total 32 extensions have been used during this test run, whereby 18 extensions have been just initialized, and for 14 methods of these extensions have been invoked. 172 test methods have caused a usage of an extension during this test run. On a detailed level, this view allows to locate the test code related to an extension (N_8). In Figure 3.17, the developer clicks on an extension of interest, for example on extension of plug-in “mylyn.tasks.ui” for the extension-point *exportWizards*, and on the right hand side, the view shows all four test methods that trigger a method invocation of this extensions.

Further, the view sheds light on the structural testing approach followed by this test suite, e.g., how many of the methods of an extension have been used and which have been left out (N_3 , N_4). The developer can see this information by double clicks on an extension of interest. For example in Figure 3.17, the developer can see from the pop-up window that method “performFinish()” and method “getContainer()” of the extension for *exportWizards*, declared in plug-in “mylyn.tasks.ui”, have been invoked by the test method “testExportAllToZip()”. Such structural information about the coverage of extensions and their methods by particular test suites can help the developer to identify “blank spots” (i.e. untested functionality) as expressed as an information need by developer P14 in Section 3.3.3.

Service Usage View

The *Service Usage View* helps to get an overview of which services are used during a test run (N_5). In eGit two services, the “*IJSchService*” and the “*IProxyService*” service, are acquired. eGit also provides one service, namely the “*DebugOptionsListener*” service.

In the Service Usage View, illustrated in Figure 3.18, the developer sees that only the service “*IProxyService*” is used during the test run, in six different test methods. If the developers zooms in further, the view reveals which methods of the service have been invoked during this test run (N_3) and which ones have not (N_4).

The *Service Usage View* helps not only to see which services are registered and used within the system under test, but also reveals which service provider has been actually chosen during test execution (N_9). Furthermore, the view shows which services have been tested by which test methods (N_8). Similar to the Extension Usage View, this view also reveals how many and which methods of a service have been actually used in this test run.

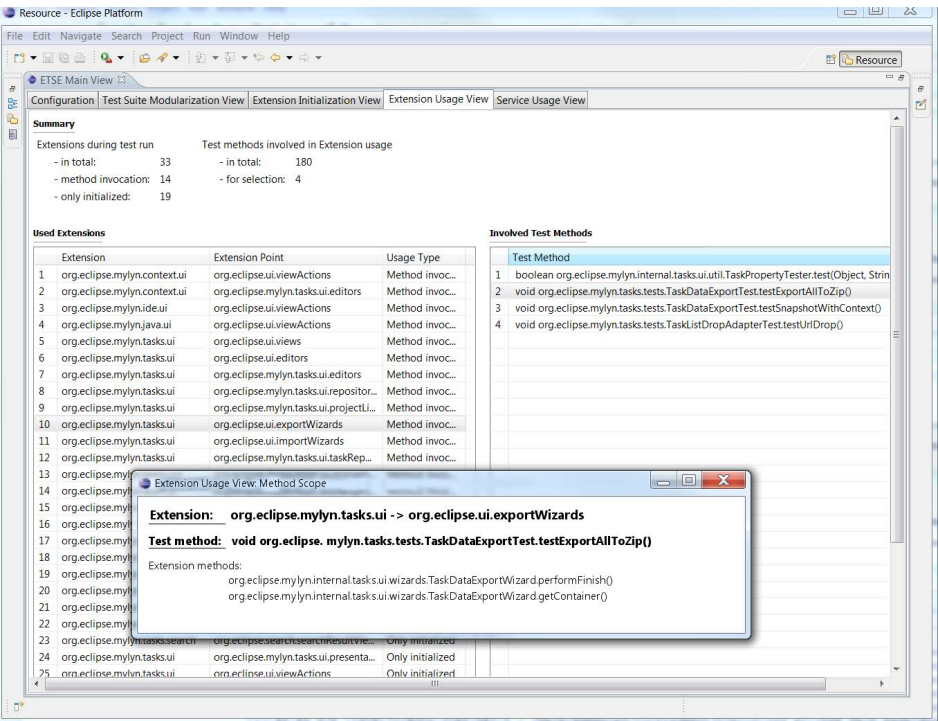


Figure 3.17: Extension Usage View showing the system scope (left side), the extension scope (right side) and the method scope (pop-up) for Mylyn

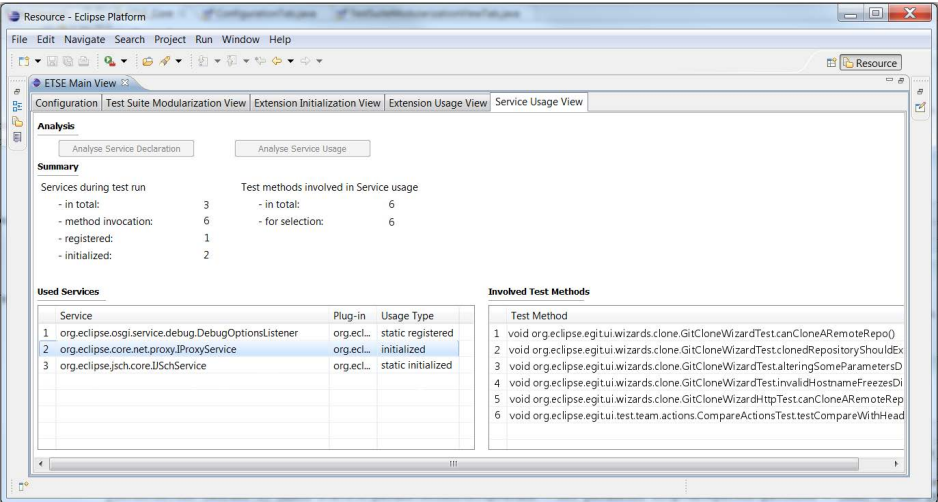


Figure 3.18: Service Usage View on system and service scope for eGit

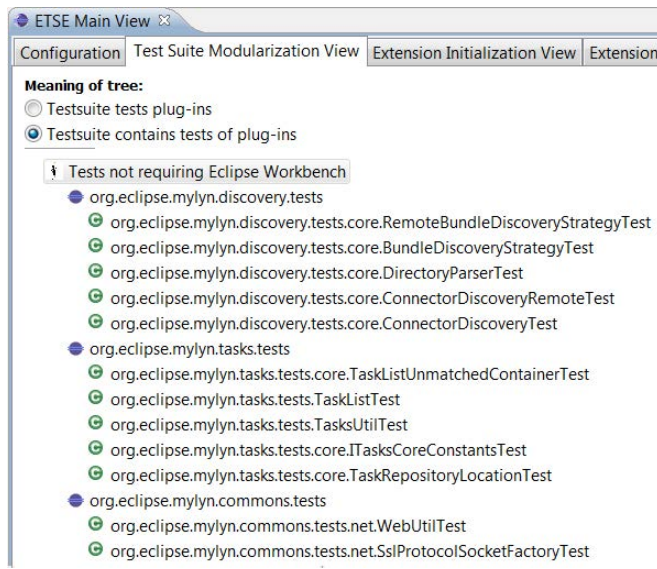


Figure 3.19: *Test Suite Modularization View showing which plug-ins contribute tests*

The plug-in systems investigated in this study still make limited use of services, mainly because they are designed to work with Eclipse 3.5-3.7. In future, we anticipate that the usage of services within Eclipse will increase drastically, especially with introduction of the new version of Eclipse, e4. In e4, the main platform API, known as the “twenty things”²² are provided by services. More and more Eclipse plug-in projects get ready for the transition to the new technologies in e4. At the moment, developers are not concerned with services in association with test suite understanding, as they only use few or none of them. On the other hand, they are well aware of their future importance and the evolution towards adoption. When adoption raises, structural information about the coverage of services and their methods by particular test suites will be as valuable to identify untested functionality, or scenarios, as this information for extensions.

Test Suite Modularization View

The *Test Suite Modularization View* helps to understand what is tested by a certain test suite (N_3), and which plug-ins are contributing tests (N_7). From Figure 3.19, the developer sees that test plug-in “mylyn.discovery.tests” contributes five test cases, and that two other plug-ins namely “mylyn.tasks.tests” and “mylyn.common.tests” contribute test cases to this test suite. On a more detailed level, the view shows which plug-ins are tested by those test plug-ins, and also which test cases are using those plug-ins (N_8).

²²<http://www.eclipse.org/e4/resources/e4-whitepaper.php>

In Figure 3.20, this view shows the three test plug-ins, and reveals (when opening those plug-ins) which plug-ins under test have been addressed by test cases from this test plug-in. For example, test plug-in “mylyn.tasks.tests” addresses ten plug-ins under test. When expanding one of those plug-ins, e.g., “mylyn.monitor.core” we see which test cases, in this case only one called “testRequestCredentials”, addresses this plug-in.

The *Test Suite Modularization View* helps to understand which test components contribute tests to a certain test suite (N_7), and which plug-ins are tested during a test execution (N_3 , N_4). During the interviews, the developers explicitly indicated this kind of information is helpful to them. Without our view, a developer might use the information provided by the standard JUnit framework integration in Eclipse, to understand what a test suite addresses. The JUnit framework allows to inspect a test run, based on a tree representation. While this standard visualization shows which test cases are executed, as well as the outcome for each test case, it does not reflect which plug-ins under test are tested, nor which test plug-ins contribute tests.

The test suite modularization view, on the other hand, addresses the correlation between test suite, plug-ins contributing test cases, and the plug-ins that have been tested in this test run. The developer can tell, e.g., from Figure 3.20, that in this test suite the plug-in “mylyn.tasks.tests” contributed test cases “testRequestCredential”, and that during the execution of this test case, plug-in “mylyn.monitor.core” has been tested.

This view helps to cover information needs related to the inspection of coverage of *plug-ins and extensions* (N_3 , N_4) and to *test organization* (N_7 , N_8), as identified in Section 3.3.3. For example, the test suite modularization view allows to inspect the test run in terms of entities (i.e. plug-ins, extensions or services) covered within each test suite, and also helps to understand test organization as it reveals which (test) plug-ins contribute tests.

3.6.3 RQ2: Scalability

We evaluate the scalability of the views with respect to their understandability by human viewers and in terms of the size of the trace files. *Answering RQ2:* In general, the views provide several abstraction levels (e.g, system, plug-in and method scope) to better cope with scalability issues. The views scale well to the case studies at hand, both in terms of space requirements and in terms of understandability of the presented views for an observer.

In the following, we discuss scalability for the views in more detail at several of the abstraction levels offered by the views.

Understandability Our views are based on visualizations of graphs, trees, and tabular data. We manage the inherent limitations of these representations by allowing users to filter data (restricting the view to, e.g., particular extensions) and

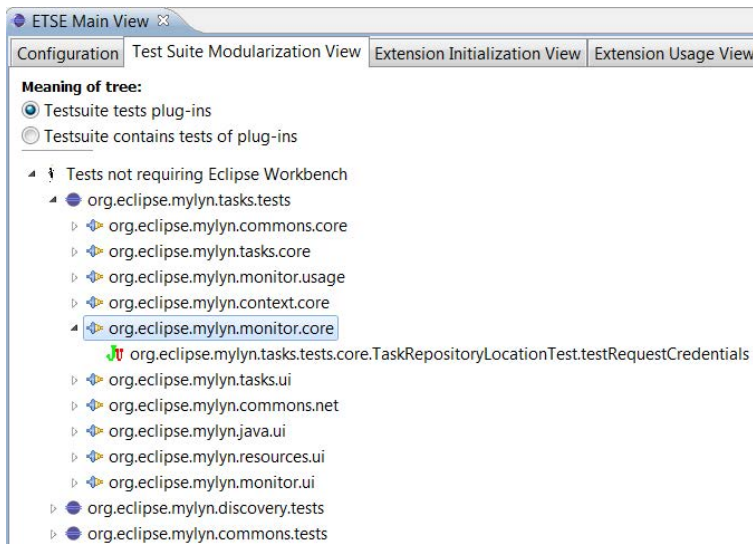


Figure 3.20: *Test Suite Modularization View showing which plug-ins are tested*

by offering views at different levels of abstraction (e.g., system, plug-in, method scope).

For example, the number of top-level entities for the *Test Suite Modularization View* for Mylyn is 11, corresponding to the 11 test plug-ins of Mylyn, while the next level consists of 27 plug-ins. The lowest level in this view is comprised by individual test methods, of which there are 518 for Mylyn, which can be managed by (un)folding parts of the tree representation.

Likewise, the extension and the service usage view present information that can be consumed per item. This means the entities do not have to be put in relation to each other by the viewer. Therefore, we consider this view as scalable, independent of the size of the system under test or the number of extensions. At all scopes, the viewer will be either interested in a summary of the data, like 15 out of 58 created extensions have been used, or the developer is concerned with a particular extension, service or method.

The *Extension Initialization View* for Mylyn displays 15 plug-ins related to extension initialization, a number which still yields comprehensible graphs. Likewise, eGit provides 53 extensions, of which 16 are initialized during the test run. If the view is applied to all extension-points of the full Eclipse workbench, the graph may comprise hundreds of nodes, which is less likely to be understandable. Such a use of this view, however, would not be the typical usage scenario: the views are intended to shed light on the extension relations of particular clusters of plug-ins, such as those of Mylyn. Note that the understandability of the view at plug-in scope depends on the number of extension-points defined per plug-in, and not on the overall size of the system under test. This means that for a small system

like eGit, the view can be as helpful as for a large-scale system. In both subject systems, the views have an average of 2 and a maximum of 10 extension-points defined per plug-in (based only on plug-ins providing extension-points).

Space Requirements Another question is the manageability of the data with respect to its required disk space. The size of the trace file used to create the *Extension Initialization View* is, e.g., 32Mb for Mylyn and 52Kb for eGit and Trac. Also the trace file used for the *Test Suite Modularization View* is manageable for all subject systems, comprising 50Mb for Mylyn, 14Mb for eGit, and 12Mb for Trac.

On the other hand, the trace file required for the identification of the Extension Usage includes trace data from several packages outside the system under test, and can become large. The trace of Mylyn for all 148 extensions has 6Gb. Likewise, the trace for all services of eGit has 300Mb. However, the number of packages included for tracing are affected by the number of extensions or services analyzed. The size of the file depends on this variable. We argue that a usual usage scenario for this view involves the inspection of a small number of up to seven extensions or services. Then the corresponding trace will be substantially smaller. Once this trace is analyzed the remaining information can be stored within a few megabytes file (e.g., 6Mb for Mylyn).

3.6.4 RQ3: Accuracy

The main concern with respect to the accuracy of the information displayed in the views relates to the heuristics used to determine if a class is related to an extension-point. *Answering RQ3:* Our heuristics capture the practices of the developers well, which is why all classifications and derived views have been accurate (i.e. no false-negative or false-positive occurred). In the following the results are discussed in more detail.

The Extension Initialization View tells the developer which test method causes an extension-point to load an extension. For Mylyn, the view shows nine test methods related to an extension-point defined in Mylyn.

We manually inspected all of those nine test methods, to see if it is apparent from the test method how it is involved in testing the extension-point. For all, it was immediately clear that the code tests the extension-point, i.e., no false-positives occurred. Due to the nature of the algorithm, extension initializations are never missed, i.e. false-negative do not appear.

The accuracy of the *Extension Usage View* is mainly influenced by the classification of the classes to be either visible or invisible for the extension-point. A classification error might occur, if the extension-point does not provide a base-class in its XML schema file. When in doubt, the algorithm behaves conservatively and classifies all types, that are extended by the class of the extension as related. This means no extension usages are missed, but it leads to a wrong classification in case the extension class does not only extend the type declared by the

extension-point, but additional classes (e.g. classes only visible to this plug-in). Then, methods of these additional classes are added to the *extension method* set. This can cause the *Extension Usage View* to show more methods of an extension as being used than those that actually occur, and the viewer has to reduce them manually (i.e. identify the right base class of the extension).

This false classification is reduced, by considering that if the extension-point is not declared in the plug-in that provides the extension, and a type extended by the extension is defined within this plug-in, this type cannot be visible to the extension-point, and can be excluded. Until the extension-point is required to indicate a type, we cannot eliminate potential misclassification. In Mylyn, all extension-points provide an XML schema-file. To get an impression for the likelihood of a misclassification we manually inspected all 29 extension classes declared for an extension-point within Mylyn, i.e. representing the system under test. None of those would have led to a misclassification. In addition, we inspected 9 extension classes created for extension-points declared outside the subject system (but in the ecosystem) to see their potential classification error. Of these, only one class would have caused a misclassification. This class is part of the extension for the extension-point *org.eclipse.ui.handlers*. Here plug-in *mylyn.tasks.bugs* provides an extension based on the class “NewTaskFromMarkerHandler”. This class extends class “MarkerViewHandler”²³ which extends “AbstractHandler”²⁴. Our algorithm would identify “MarkerViewHandler” already as a potential extension class, whereas the base type of the extension is defined as “AbstractHandler”. This leads to the inclusion of methods defined in “MarkerViewHandler” in the analysis.

Because the *Service Usage* does not only rely on static data, but also uses dynamic data about service registration events, we can determine the runtime type of each service, and therefore determine which service of a particular service provider is invoked during service usage. For the *Test Suite Modularization View* no heuristics are used.

3.7 Discussion and Future Work

3.7.1 User Study

Our present evaluation is via case studies, aimed at assessing the applicability, scalability, and accuracy of our approach. With the confidence gained from these cases, a logical next step is to involve actual users in the evaluation.

First feedback from Eclipsers was obtained via a presentation of our findings about test suite understanding, as well as the five architectural views, to approximately 70 practitioners during the *Eclipse Testing Day*.²⁵ The overall responses

²³This class is in the package `org.eclipse.ui.views.markers`

²⁴This class is in the package `org.eclipse.core.commands`

²⁵http://wiki.eclipse.org/Eclipse_Testing_Day_2011

were positive, and encouraged us to proceed.

We consider a full user study as beyond the scope of the present paper. However, to get some initial insight we took the following steps already. In particular, we asked three developers, who have been participants in the grounded theory study described in Section 3.3, for their feedback on ETSE. We demonstrated the tool and outlined the meaning of each architectural view. All three participants reacted very positively and expressed that the tool gives them a new perspective on plug-in test suites. The developers found the visualization of the degree of integration testing of system under test, illustrated by the *Extension Initialization View*, very useful.

Two new features emerged during this interviews: First, all developers said that an additional visualization of the views similar to coverage tools would be interesting. Second, one developer explained that he thinks a further abstraction level based on Eclipse *features* (i.e. sets of plug-ins shipped together) could be beneficial for illustration of integration with third party systems.

3.7.2 Limitations

One of our current limitations relates to the boundaries of the system under test. At the moment, we are only partly addressing the integration of the system under test in its ecosystem. The views mainly focus on the relations within the system under test. Contributions to the ecosystem, i.e., extensions from the system under test for extension-points defined outside are addressed. Extensions outside the scope of the system under test for extension-points within the system under test are not automatically covered. For example, that would be any extension defined by a foreign plug-in for a plug-in inside the system under test. In case the tester is interested in such integrations, those foreign parts of Eclipse must be included in the system under test.

3.7.3 Recommendations

Standardization Eclipse extensions can be of two types: data or executable. In Eclipse there is no formal way to distinguish them. Furthermore, an extension-point is not forced to provide an XML schema-file describing the syntactical contract between creator and contributor. Being stricter in the declaration for extension-points would not only help ETSE identify the proper extension relations, but also help Eclipse developers understand the relationships more easily.

Set-Up and Tear-Down While executing a test suite with the Eclipse plug-in test runner, the framework is only started once. Also plug-ins and extensions are created on demand and are not automatically stopped after a test execution of one method. This means that the execution of a test method can change the state of the system, and therefore possibly change the outcome of following tests. For example, a test method that creates an extension, might also need to activate the

plug-in providing this extension. In the case, the extension would be used also by a subsequent test method, this test method would not have to activate the plug-in anymore. We are of the opinion that there is not enough awareness for the implications of such dependencies between tests. The test runner should allow users to configure the set-up and tear-down of the execution environment, in this case Eclipse. We anticipate that such information would be useful to integrate with ETSE views as well.

3.7.4 Threats to validity

With respect to *external* validity, the case studies chosen, Mylyn, eGit, and Trac connector can be considered representative for Eclipse plug-ins. In particular Mylyn is a complex plug-in, and hence we expect the views to be applicable to other complex plug-ins as well.

While the extension mechanism is Eclipse-specific, it is essentially a callback mechanism, which is a common way to achieve extensibility in many systems. We conjecture that the proposed views are applicable for other systems utilizing callback mechanisms as well, in particular if they are, like Eclipse, based on OSGi. Further, the views are also usable for OSGi services, and two of the views can be used independently from the extension mechanisms or the services.

Concerning *repeatability*, the subject systems are open source and accessible by other researchers.

The views have been implemented in ETSE. Since, ETSE is a software system, and also relies on several other frameworks, such as the BCEL framework, the implementation might not be free of bugs, and the quality of the system constitutes a threat to *internal validity*. We took countermeasures against this threat by testing ETSE by means of an automated test suite, and we manually checked many of the results the tool delivered by inspecting the code.

3.8 Related Work

A recent survey on the use of dynamic analysis for program understanding purposes is provided by Cornelissen et al. (2009). One of the findings of this survey is that very few studies exist addressing dynamically reconfigurable systems – a gap that we try to bridge with our paper.

In the area of test suite analysis and understanding, van Deursen et al. (2002) proposed a series of JUnit test *smells* (pointing to hard to understand test cases) as well as a number of refactorings to remedy them. Later, this work was substantially elaborated by Meszaros (2007) into an extensive book on xUnit patterns. Van Rompaey et al. (2007) propose a formalization of a series of test smells, as well as metrics to support their detection. They also propose heuristics to connect a test class to its corresponding class-under-test – which we also use in our approach. Galli et al. (2004a) present a taxonomy of (Smalltalk) unit tests, in

which they distinguish tests based on, for example, the number of test methods per method-under test, and whether or not exceptions are taken into account.

In order to support the understanding of test suites, Cornelissen et al. (2007) investigate the automated extraction of sequence diagrams from test executions. Zaidman et al. (2008) investigate implicit connections between production code and test code, by analyzing their co-evolution in version repositories. While these studies provide important starting points, none of them approaches test suite understanding from an integration or extensibility point of view.

Koochakzadeh and Garousi (2010) present a graph-based test coverage visualization tool, whose usefulness is evaluated by Garousi and Varma (2010). The tool, which is now part of the CodeCover test coverage tool for Eclipse, allows to view the test coverage between two artifacts on different scopes (i.e. test package, class and method in xUnit). The views of this tool differ strongly from our views as their focus is visualization of purely “traditional” coverage information with no connection to the plug-in characteristics of systems under test. Further, our approach also analyzes the static meta data information of the plug-in system to gain information on potential integration possibilities and their coverage within the test execution. Visualization of those connections helps also to facilitate comprehension of the system under test, and its test suite.

A substantial body of research has been conducted in the area of integration testing (Binder, 1999; Jorgensen and Erickson, 1994; Pezzè and Young, 2008). Closest to the Eclipse extension mechanism are test strategies addressing polymorphism, such as the *all-receiver classes* adequacy criterion (Rountev et al., 2004).

Most integration testing approaches are model-based, and explain how models, e.g., UML state machines, can be used to derive test cases systematically (Hartmann et al., 2000; Reis et al., 2007). In the Eclipse setting, it is not common to have models of plug-ins and their extension-points available a priori. As we saw, however, our views can be reverse engineered from static dependency declarations as well as from run time plug-in interactions. As such, they can help developers compare actual plug-in interactions with declared dependencies.

The Eclipse plug-in architecture and the related Eclipse IDE are well studied subject systems in the research community. For example, Wermelinger and Yu (2008) analyze the evolution of Eclipse plug-ins, and Mens et al. (2008) investigate whether software evolution metrics are supported within the Eclipse context. Both studies analyze the evolution of Eclipse, whereas our study performs a static and dynamic analysis to study extensibility relations between plug-ins.

The grounded theory method originates from the social sciences, and has nowadays gained popularity in the software engineering research field (Dagenais and Robillard, 2010; Hermans et al., 2011; Rigby and Storey, 2011). Also for our study, grounded theory was beneficial as it is suitable, in particular, for explorative, human-centered research areas.

3.9 Concluding Remarks

In this paper, we have investigated the task of understanding test suites for plug-in-based architectures, and proposed five architectural views to facilitate comprehension. In particular, the following are our key contributions:

1. An investigation of the task of “understanding plug-in-based test suites” by means of interviews with 25 professional;
2. Five architectural views that can be used to understand test suites for plug-in-based systems from an extensibility perspective for various extension mechanisms;
3. The Eclipse Plug-in Test Suite Exploration (ETSE) tool, which recovers the proposed views from existing systems by means of static and dynamic analysis, and which can be integrated in the Eclipse IDE; and
4. An empirical study of the use of these views in Mylyn, eGit, and a Mylyn connector.

In our future work, we will first of all apply the proposed approach to more plug-in-based architectures in collaboration with Eclipse developers. Within this collaboration, we are planning to conduct a thorough user study, with professionals, to investigate the usefulness of the views during typical test suite comprehension and/or maintenance tasks. Furthermore, we will investigate to what extent the views can be used as a base to derive adequacy criteria used to prevent failures reported in the actual usage of concrete plug-in-based systems such as Eclipse. Finally, we plan to enhance this base with models representing the shared properties of plug-in-based systems. Together, from the models a new test strategy and approach for plug-in-based systems that provide dynamic reconfigurations should emerge.

Measuring Test Case Similarity to Support Test Suite Understanding

Abstract

In order to support test suite understanding, we investigate whether we can automatically derive relations between test cases.¹ In particular, we search for trace-based similarities between (high-level) end-to-end tests on the one hand and fine grained unit tests on the other. Our approach uses the shared word count metric to determine similarity. We evaluate our approach in two case studies and show which relations between end-to-end and unit tests are found by our approach, and how this information can be used to support test suite understanding.

4.1 Introduction

Modern software development practice dictates early and frequent (automated) testing. While automated test suites are helpful from a (continuous) integration and regression testing perspective, they lead to a substantial amount of test code (Zaidman et al., 2011). Like production code, test code needs to be maintained, understood, and adjusted upon changes to production code or requirements (Greiler et al., 2010; Meszaros, 2007; Van Rompaey et al., 2007).

In light of the necessity of understanding and maintaining test suites, which can become very costly due to the large amounts of test code, it is our stance that tool support can reduce the burden put on the software and test engineers.

¹This chapter is published in the proceedings of the 2012 International Conference on Objects, Models, Components, Patterns (TOOLS 2012). The authors of this publication are Greiler, van Deursen, and Zaidman.

The V-model from Figure 4.1 shows that different levels of tests validate different types of software artifacts, with each level contributing to the large amount of test code. Figure 4.1 also shows that, ideally, requirements can be *traced* all the way to source code, making it easier to perform *impact analysis*, i.e., determining what the impact of a changing requirement is on the source code. The right side of the V-model however, the test side, does not have similar tool support.

In this paper we propose to support engineers by helping them to understand relationships between different types of test suites. As an example, an automated test suite can include “end-to-end” tests, exercising an application from the user-interface down to the database, covering functionality that is meaningful to the end user.² The test suite will typically also include dedicated unit tests, aimed at exercising a very specific piece of behavior of a particular class. Suppose now a requirement changes, entailing a modification to the end-to-end test, which unit tests should the software engineer change as well? And vice-versa, if a unit test is changed, should this be reflected in an end-to-end test as well?

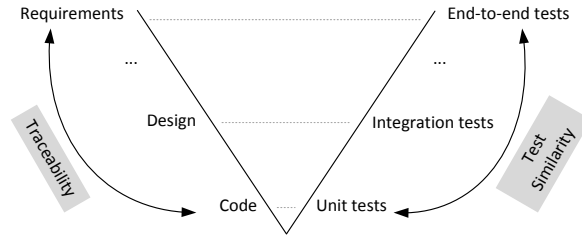


Figure 4.1: *The V-model for testing*

Our goal is to develop an automated technique for establishing relations between test cases, in order to assist developers in their (test suite) maintenance activities. To reach this goal, we employ *dynamic analysis* (Cornelissen et al., 2009). We collect call traces from test executions, and use these to compute a *similarity* value based on the shared word count metric. The resulting technique, which we call *test connection mining*, can be used to establish connections between test cases at different levels. An implementation of our technique is available via a framework called the *Test Similarity Correlator*.

We evaluate the approach in two case studies, by elaborating on the usefulness of the approach to improve the understanding. We analyze how measuring similarity based test relations can help to (1) find relevant tests by showing test relationships, (2) understand the functionality of a test by describing high-level test cases with related unit test cases and (3) reveal blank spots in the investigated unit test suites.

This paper is structured as follows: in Section 4.2, we discuss our test execution tracing approach. In Section 4.3, we describe the similarity metrics we use to

²We deliberately did not use the term acceptance test, as it is commonly associated with tests executed by the customers/users.

compare traces. Subsequently, we describe our approach and its implementation (Section 4.3) as well as the set-up of our case studies (Section 4.4). The two case studies are covered in Sections 4.5 and 4.6. We conclude with discussion, related work, and a summary of our contributions in Sections 4.7–4.9.

4.2 Tracing and Trace Reduction

Test connection mining first of all requires obtaining execution traces with relevant information of manageable size. This section describes the specific execution trace we use and the trace reduction techniques we apply.

4.2.1 Tracing Test Executions

Before the test run, the production and test code are instrumented. Subsequently, during the test run we obtain an execution trace comprised of various types of *events*: (1) *test execution* events represent the execution of a test method, (2) *set-up and tear-down* events mark the execution of a method which is either used for test set-up or tear-down, (3) *method execution* events signalling the execution of a public method within the code under test and (4) *exception thrown* events indicating that an exception has been thrown.

For the similarity measurements it is important to be able to distinguish between production and test code. Otherwise, executions of test helper methods will appear in the trace and render the test cases as less related. Due to the common practice to put the test code in a separate package (e.g., `test.jpacman`), we simply filter executions of methods belonging to test code out during instrumentation. If test and production code are within the same packages, test classes can be annotated and correctly addressed during instrumentation.

4.2.2 Handling mocks and stubs

When mocks or stubs are used, care has to be taken to correctly trace calls to mocked classes and map these calls to the corresponding original class.

A first issue is that using mocking frameworks can have the effect that an automatically created mock-object is not part of the instrumented package. For example, by using the JMock³ library interfaces and classes defined to be mocked are automatically wrapped in a proxy class which is located in the same package as the class directing the mocking operation, which will usually be the test class or a helper class within the test package. Because we do not trace executions of methods defined in the test package, these classes have to be addressed specifically. We do so by keeping track of a list of mocked types.

Mocking also plays a role for tracing the production code, as the mocked and unmocked classes have to be mapped to allow identifying their similarity. There-

³<http://www.jmock.org>

Listing 4.1: *Trace differences with or without mocking*

```
//Execution of method join of the mocked interface Sniper
void TestClass.$Proxy1.join()

//Execution of method join of class AuctionSniper implementing Sniper
void AuctionSniper.join()
```

fore, we have to indicate that a method on a mockable object has been invoked. To that end, we check whether the runtime type of the target is contained in the list of mocked classes. If yes, we further investigate whether the method intercepted is part of the mockable type, since a class implementing a mockable interface can have additional methods. Therefore, we derive recursively, via reflection, the set of methods belonging to this type including all methods defined by it and its (potential) super-types. Only if the method intercepted is an actual method of the mockable type, we discovered a mockery execution. As such, we add it to the trace and mark it with a *mockery mark*.

Finally, we need to neutralize those mock and stub calls. As illustrated in Listing 4.1, an execution of a method of a mocked type can be traced as the execution of an inner class within the (test) class defining the mock operation. As this differs from the trace events left behind by the execution of a method of the actual type, we render them as similar, by using the *mockery marks* set during tracing. Note also the actual type might differ from the mocked type by being the implementation of a type or extending a common type. We inspect the trace and replace all executions of methods of an actual type, as well as the executions of the mocked type by their common (super) type. For example, the traces in Listing 4.1 would be mapped to “void Sniper.join()”.

4.2.3 Trace reduction

Trace reduction techniques are important in order to reduce the size of traces to improve performance and scalability, and to help reveal the key functionality of the test case, e.g., by reducing common functionality or noise (Cornelissen et al., 2008). We adopt the following five reduction techniques before starting our analysis:

Language based Reduction. The traces are reduced to only public method executions and do not comprise any private or protected methods. Furthermore, only the production code is fully traced; for the test code only test method execution events are traced to be able to separate individual test cases from an entire test run.

Set-up and Tear-Down Reduction. As set-up and tear-down methods do not explicitly contribute to the specific focus of a unit test, and are usually shared by each test case within a test class, all method executions taking place during set-up or tear-down are removed.

Shared Word Reduction. This trace reduction focuses on helping identify the core functionality of a test case, by removing trace events that are omnipresent in almost all test traces (defined by a variable threshold).

Complement Reduction. This reduction focuses on reducing the trace size by removing calls within the trace of interest that are not existing in any of the test traces to compare to. Although, after such a reduction target traces will be calculated as more similar to the source trace, the reduction itself does not influence the information perceived useful for ranking the target traces with respect to each other.

Unique Set of Calls. This technique reduces all trace events within a trace to a unique set of events. Because information such as order and position of events are not preserved this reduction is only useful for similarity measurements that do not take such information into account.

4.3 Determining Similarity Measurements

The second step involved in test connection mining consists of computing trace similarities. In particular, we compute the similarity between a source trace te (e.g., from an end-to-end test) and a target trace tu (e.g., from a unit test).

As similarity metrics we compared (1) shared word count (Weiss et al., 2004), (2) Levenshtein distance (Stephen, 1994) and (3) pattern matching based on the Knuth-Morris-Pratt algorithm (Stephen, 1994). From an initial experiment we observed that all three metrics provided similar results, which is why we continue with the shared word count in the remainder of this paper.

The shared word count measurement (Weiss et al., 2004) assesses the number of tracing events that two test execution traces have in common. The similarity between a source trace and a target trace is calculated as the number of tracing events comprised in both test traces.

4.3.1 Relevancy support based on occurrence

Some tests are related to other tests, because they test common functionality. Using this piece of knowledge, we can improve our results, by marking these more general tests as less important. Vice versa, by giving a test appearing less often a high impact, results with more specific functionality are ranked higher. We do so by multiplying the similarity measurement for a specific trace tu with the total number of test cases it has been compared to, and dividing this by the number of times the trace appeared as a result. We also use the average similarity of test case tu_i to rank similar results. For example, if target test cases tu_1 and tu_2 have the same similarity with te , than the test case with the smaller average similarity among all te_j is ranked first.

4.3.2 Implementation

We implemented the various trace reduction techniques and similarity measurements presented in this paper in a Java based framework called *Test Similarity*

Correlator.⁴ Our tool offers an API to steer customized test similarity measurements, varying in trace reduction, thresholds and similarity calculations.

To instrument the test execution we use the AspectJ⁵ framework. We offer three different annotations to facilitate tracing of execution of test-methods, set-up and tear-down methods. *Test Similarity Correlator* comprises several aspects, addressing join points to weave in our tracing advices, including the aspect to address code generated by the mocking library JMock.

4.4 Set-Up for Case studies

4.4.1 Research Questions

To evaluate the usefulness of test connection mining, we conducted an explorative study based on two case studies. In these case studies, we aim at answering the following questions:

- RQ1 How do the associations found by test connection mining relate to associations a human expert would establish?
- RQ2 Are the associations found useful for a typical test suite understanding task, i.e., getting familiar with the test suite of a foreign system?
- RQ3 How does mocking influence the similarity measurements?
- RQ4 What are the performance characteristics, both in time and in space, of the analysis conducted?

To answer these questions, we select a subject system that is shipped with both a functional test suite as well as a unit test suite. We manually compile a *conceptual mapping* between unit and functional test cases, and compare these to the mappings found through test connection mining automatically.

The first case study is used to assess RQ1 and RQ4, whereby in the second case study we focus on RQ2 and RQ3.

4.4.2 Technique customization

The specific trace reduction configuration (see Section 4.2) we use in the case studies consists of the following steps.

Before calculating the trace similarity, traces are reduced by using the *Set-up and Tear-Down* reduction, followed by the *Shared Word*, and the *Complement* reductions. The order of the reduction is important and influences the ranking of the results. For example, if all unit test cases call a method “quit()” as part of their tear down method, but only one unit test actually uses this method during test execution, the application of first the *Shared Word* reduction and then the

⁴<http://swerl.tudelft.nl/bin/view/Main/TestSimilarityCorrelator>

⁵<http://www.eclipse.org/aspectj>

Set-up and Tear-Down reduction would eliminate this call from the trace. The *Shared Word* reduction technique can be customized by a threshold influencing how many traces must comprise a trace event before it is removed.

For similarity measurements based on shared word count, which does not take the order of events into account, the traces are further reduced to their unique set of events.

4.5 Case Study I: JPacman

As first subject system we use JPacman,⁶ a simple game written in Java inspired by Pacman, used for educational purposes at Delft University of Technology since 2003. Key characteristics of JPacman are listed in Figure 4.2.

JPacman follows the model-view-controller architecture. Each class in the model package comes with a (JUnit) unit test class, which together comprise 73 unit test cases. The test suite makes use of several test patterns described by Binder (1999), using state machines, decision tables, and patterns such as *polymorphic server test* (reusing superclass test suites at the subclass level). This results in a line coverage of 90% in the model package, as measured by Cobertura.⁷

The functional test suite is directly derived from a set of JPacman user scenarios written in *behavior-driven development*⁸ style. These scenarios are of the form given in Listing 4.3. There are 14 such scenarios, each of which is translated into a JUnit test case. The resulting functional test cases exercise around 80% of the user interface code and around 90% of the model code.

4.5.1 Obtaining the Conceptual Mapping

JPacman's main developer created a conceptual mapping in advance. The key criterion to identify a relation between an end-to-end test t and a unit test u was the question whether the behavior of u is important in order to understand the behavior of t . The conceptual mapping contains both *positive* incidences

Code size (lines)	4,000
Test code size (lines)	2,000
No of classes	26
No of test classes	16
No of unit tests	73
No of functional tests	14

Figure 4.2: JPacman characteristics

```

Given [context]
And [some more context]...
When [event]
Then [outcome]
And [another outcome]...

```

Figure 4.3: JPacman Test Scenarios

⁶Version 4.4.4, dated October 2011. JPacman can be obtained for research and educational purposes from its author, 2nd author of this paper.

⁷<http://cobertura.sourceforge.net/>

⁸<http://dannorth.net/whats-in-a-story/>

(important connections to be established) and *negative* ones (unlikely connections that would be confusing). In most end-to-end (ETE — numbered from *ETE01* to *ETE14*) test cases, we had at least 5 positive and 9 negative connections.

While the mapping obtained can be considered a useful baseline, it should be noted that it is *incomplete*: it only identifies clearly connected and clearly disconnected test pairs. The remaining tests are categorized as *undecided*. Furthermore, we tried to be as specific as possible: relations to “infrastructure” unit test cases relevant to many end-to-end tests were not included.

4.5.2 RQ1: Comparison to Conceptual Mapping

We used a spreadsheet containing 14×73 matrices to study the differences between the conceptual mapping as well as the ones obtained through our automated analysis. Due to size restrictions, we can not show all results of the measurements⁹. Besides saving space, showing the top 5 results is also realistic from a practical point of view, because in practice a user of the tool would also look primarily at the highest ranked results. In Table 4.1 we show for each end-to-end test the 5 most similar unit tests based on the shared word count metric. A ranking is indicated as *optimal* in case it is marked as highly related in the conceptual mapping and it is ranked high (i.e. top match). Incidences marked as related by the expert which are high ranked are evaluated as *good*. Results of the category undecided are subjected to additional manual analysis: the results are indicated as *ok* only if the relation is strong enough to be justified, and labeled as *nok* otherwise. Unrelated results ranked highly, as well as (highly) related results ranked low, are also evaluated as *nok*.

The overall impression is that the automated analysis provides a useful approximation of the manually obtained mapping. Looking at all the results for each end-to-end test case, we found that:

- For all but one end-to-end test (i.e. *ETE02*), the top match is ranked as the first or second automatically calculated result.
- Within the top 10 results only one unit test case marked unrelated is listed.
- All remaining results ranked within the top 10 (i.e. from the undecided category) are sufficiently related to the end-to-end tests to justify investigation.
- No relations are missing as all test cases marked as relevant by the expert have been identified as related. Thereby, 80% of all test cases marked as related have been ranked within the upper half of the results *showing similarity* and within the top 30% of overall results.

⁹The complete results are available at <http://swerl.tudelft.nl/twiki/pub/Main/TestSimilarityCorrelator/similarityResults.zip>

Test Case (no.)	match	Test Case (no.)	match	Test Case (no.)	match
1 Move to empty cell & undo		2 Move beyond border		3 Move to wall	
MovePlayer(23)	optimal	FoodMove(44)	ok	DxDyImpossMove(15)	optimal
UndoEmptyMove(17)	optimal	FoodMoveUndo(39)	ok	SimpleMove(22)	good
UndoDxDy(18)	optimal	UndoFoodMove(19)	ok	DieAndUndo(26)	optimal
UndoFoodMove(19)	ok	PlayerWins(24)	ok	DieAndRestart(25)	optimal
Apply(38)	optimal	wonSneakPaths(35)	ok	MovePlayer(23)	good
4 Eat food & undo		5 Win and restart		6 Get killed and restart	
FoodMoveUndo(39)	optimal	SetUp(12)	ok	DieAndRestart(25)	optimal
UndoFoodMove(19)	optimal	PlayerWins(24)	optimal	PlayerWins(24)	ok
UndoFood(47)	optimal	FoodMoveUndo(39)	good	wonSneakPaths(35)	ok
FoodMove(44)	optimal	FoodMove(44)	optimal	Updates(37)	ok
MovePlayer(23)	good	DxDyPossibleMove(14)	ok	UndoFoodMove(19)	ok
7 Monster to empty cell		8 Monster beyond border		9 Monster to wall	
UndoMonsterMove(16)	optimal	Wall(70)	optimal	EmptyCell(69)	optimal
MoveMonster(28)	optimal	MonsterPlayer(73)	ok	MonsterFood(72)	ok
Updates(37)	ok	MonsterFood(72)	ok	MonsterPlayer(73)	ok
OutOfBorder(68)	ok	EmptyCell(69)	optimal	Wall(70)	optimal
FoodMove(71)	ok	MonsterKillsPlayer(27)	ok	MonsterKillsPlayer(27)	ok
10 Monster to food		11 Monster to player		12 Suspend	
MoveMonster(28)	optimal	MonsterPlayer(73)	optimal	SuspendRestart(29)	optimal
Updates(37)	ok	Wall(70)	ok	Start(21)	good
Apply(66)	good	MonsterFood(72)	ok	SneakPlaying(33)	ok
FoodMoveUndo(67)	optimal	EmptyCell(69)	good	SuspendUndo(30)	optimal
FoodMove(71)	ok	MonsterKillsPlayer(27)	optimal	SneakHalted(36)	good
13 Die and Undo		14 Smoke			
DieAndUndo(26)	optimal	SetUp(12)	good		
MovePlayer(23)	good	PlayerWins(24)	optimal		
wonSneakPaths(35)	ok	FoodMoveUndo(39)	ok		
SimpleMove(22)	ok	wonSneakPaths(35)	ok		
DieAndRestart(25)	optimal	FoodMove(44)	ok		

Table 4.1: Top 5 ranked unit tests per end-to-end test for JPacman

- 92% of all tests marked as unrelated correctly map to *no similarity* by the measurements. The remaining unrelated tests revealed weak connections and have been ranked in the bottom half of the results, except for one test (14).

Correct Identifications. *Top matches.* The top two results of the measurements in most cases also contain the top match for an end-to-end test case. For example, the end-to-end test involving a keyboard event to move the player to the right and then undoing that move (*ETE01*), is connected to a unit test actually moving the player. As another example, *ETE03* attempts to move through a wall (which is impossible), and is connected to a unit test addressing the correct positioning of the Pacman’s mouth after attempting an impossible move. As dying is a type of an impossible state, connections to dying are also correct.

Moving Monsters vs. Players. Some groups of test cases are moving players (i.e. 44, 45, 46), whereas other tests (72, 73, 74) are moving monsters. In the con-

ceptual mapping, tests moving players are related to ETE tests 1-6, and marked as unrelated for ETE tests 7-11, whereby tests moving players are related the opposite way. These relations respectively non-relations are correctly identified by the measurements, except for test case 74, which we will outline below.

Surprises. *Moving Monsters.* According to the expert, a group of tests (72, 73, 74) all move monsters, and should lead to similar rankings. Surprisingly, one test (74) performs differently from the rest, and also differs from the conceptual mapping. After investigation, it became apparent that this test is not as focused as expected and desired by the expert. The test even concludes with a method which is never followed by an assertion statement. This investigation revealed a clear “test smell” and pointed the expert to a place in need of a refactoring.

Sneak paths. A surprising connection is provided for the “monster to player” test (*ETE11*), which is connected to “wonSneakPaths” (35). This relates to unit tests aimed at testing for *sneak paths* in state machines, following Binder’s test approach for state machines (Binder, 1999). A sneak path is an illegal transition, and the JPacman sneak path test cases verify that illegal (state, event) pairs do not lead to a state change. To do so, the test case brings the application in the desired state (e.g., *Playing*, or *Died*), and then attempts to trigger a state change.

The process of bringing the application in a given state, however, may bear similarity with other test cases. For example in unit test 35, the player first wins. Then multiple steps, such as the player getting caught by a monster or the player running into a monster, are triggered which should not change the state from “won” to “lost” anymore. As this triggers the player to die or being killed, this sneak path test case shows up as being related not only to end-to-end tests triggering winning situations. A better separation of set-up logic (bringing the application in the right state) and executing the method-under-test would help reveal more focused associations.

Deviations. *Moving beyond border.* *ETE02* is the only test which does not map to a top match within the first 5 results. The first top matches are found from rank 7 onwards. Reasons for this behavior are that *ETE02* is one of the smallest end-to-end tests involving a move, and that testing the behavior for “beyond border” covers branches that only lead to different data, not different calls made. All 5 high ranked results correctly involve doing a move. After investigation of the results, the expert reports that the unit test cases indicated as related in the conceptual mapping do a bit more than only a move (e.g. an undo operation), which is why our approach gives these unit tests a lower rank.

Move to Wall. *ETE03* contains the only unrelated connection within the top 10 results: on rank 9 is the “possible move” test. On the other hand, counterpart test “impossible move” is a top match.

Disparate test sizes. The main deviations (tests marked as unrelated being ranked higher than tests marked as related) are due to extreme size differences in unit tests. The expert easily relates narrow focused tests, whereby the automatic approach, by design of the shared word count, gives preference to broader tests

(which share more events). A prime example is the “wonSneakPaths” test, which is related to many end-to-end tests as it triggers a broad range of functionality. The more equal the amount of functionality tested by the unit test cases is, the better the results revealed by the automatic approach.

Additional Lessons Learned. *API violations.* The smoke test (*ETE14*) consists of a series of events applied to JPacman. As such, it is fairly heterogeneous, doing many different things. This makes it hard to come up with a reasonable positive mapping to unit tests. On the other hand, some unit test cases are not relevant to *any* end-to-end test, including the “smoke test”. As an example, tests 57, 58 and 59 deal with using the API in a wrong way, which should generate an appropriate exception. Seeing that these test cases are not related to the smoke test gives confidence that such violations are not possible at a higher level.

Human vs. automated mapping. Fine-grained deviations between tests, like state and specific object instantiations, have been used by the expert to relate tests to each other. For example, for the expert the determining events for relating unit test cases involving moving to end-to-end tests have been the actual actors (objects). The automated approach is able to differentiate similar to an expert between objects. On the other hand, the importance of states for human mappings is not equally reflected by the automated approach as it assigns every event the same importance. Identifying and prioritizing states before the similarity calculation is performed could improve the approximation to the “human” conceptual mapping. As we will see in the second case study, if tests are small and focused, the impact of state changes reflects well in the similarity measurements.

4.5.3 RQ4: Performance Characteristics

Since JPacman is the larger case study, we will answer RQ4 here. The traces obtained for both case studies are relatively small: the smallest one is 1kb and comprises 2 trace events, the largest being 62Kb and 700 trace events (after applying trace reduction). Similarity calculations within this dimension are computed within 10 seconds for the whole test suite. Even the results for the smoke test of JPacman, comprising approximately 60,000 trace events (4Mb) before reduction, are almost instantly ready after applying trace reduction techniques.

4.6 Case Study II: Auction Sniper

The second case study revolves around a system developed in strict test-driven development (TDD) manner called Auction Sniper. Its test suite also makes heavy use of mocking techniques in order to isolate unit tests. In contrast to the first case study, where we compare the test relations with a conceptual mapping of an expert, in this case study we investigate the usefulness of the technique to help an outsider understand test relations (RQ2). In addition, we investigate how our technique can cope with the influence of mocking techniques (RQ3).

Auction Sniper is an application which allows to automatically bid in auctions. Auction Sniper watches different auctions and increases the bid in case a higher bid of another bidder arrived until the auction closes or a certain limit has been reached. This system is used as an example in the book “Growing Object-Oriented Software, Guided by Tests” by Freeman and Pryce (2009) to describe TDD techniques. The software and the related tests are explained in detail in this book and are publicly available¹⁰. The system comprises approximately 2,000 lines of code. The test suite has 1,240 lines of code, which constitute 37 unit tests, 6 end-to-end tests and 2 integration tests.

4.6.1 Obtaining an Initial Understanding

We analyzed the book and derived an initial understanding of the relations between end-to-end tests and unit tests. The authors always start with an end-to-end test, which kick-starts each development circle for a new functionality, whereby the authors explain each end-to-end test “should have just enough new requirements to force a manageable increase in functionality” Freeman and Pryce (2009). Then, the scaffold implementation of the new functionality follows. Prior to implementation of detailed functionality, the authors develop and explain the necessary unit tests.

Based on this iterative development, we map each unit test case developed within the cycle of an end-to-end (ETE) test as related to this ETE test. We refine this first mapping by identifying the differences of the ETE test cases based on their code. We mapped some unit tests not covered in the book based on their name. In the following we summarize the functionality of the six ETE tests. All unit test case names are given in Table 4.4 which can be helpful during comprehension of the presented results.

The end-to-end tests. ETE tests 01 to 06 are actually enhancements of each other, involving a lot of common functionality. The main steps are: 1. An auction sells an item, 2. An auction sniper joins this auction. 3. Then, the auction closes, 4. Finally, the auction sniper shows the outcome of an auction.

In addition, test cases 02 to 05 place actual bids. Only test case 06 deviates from the rest, as it does not close the auction and sends an invalid message. Another main difference between the test cases is the state in which the sniper is when the auction closes. In *ETE01* the sniper is in the state “joining” when the auction closes, which results in a lost auction. In *ETE02* the sniper makes a bid, but loses in the “bidding” state. In *ETE03* the sniper makes a higher bid, and wins in the “winning” state. *ETE04* simply tests that a sniper can bid on two items. The functionality of *ETE03* and *ETE04* is so similar that we will treat them as one test subsequently. In *ETE05* the sniper is already in “losing” state before the auction closes, because of a stop price. *ETE06* tests failure reporting. The test sends an invalid message to the auction and causes the application to

¹⁰<https://github.com/sf105/goos-code>

throw and handle an error, but leaves the auction unclosed.

4.6.2 RQ2: Suitability of Measurements for Understanding Test Relations

After measuring the similarity of the tests, we investigate each unit test via code inspection, and assess the ranking and the mapping, which results in the final conceptual mapping illustrated in Table 4.4. Based on this detailed investigation we finally assess the rankings of the similarity measurements. Below we outline correct identifications, surprises and deviations of the measurements with our initial understanding by sketching groups of unit tests. We will see that the automatic mapping reflects the final mapping derived after in-depth investigation very accurately, and is thus useful for supporting an outsider in understanding the test suite and its relations. The rankings and assessments for the best 5 results are illustrated in Table 4.2. For test case *ETE06* we present the top 10 results to illustrate the effect of the relevancy support (see Table 4.3). A ranking is indicated as *optimal* only in case it is highly related and ranked within the top first results. Otherwise, results highly related, or results related are indicated as okay (i.e., *ok*) in case they are within the first 5 results. On the other hand, in case of a related result, which is not highly related, but is ranked before the highly related ones, it is marked as not okay (i.e., *nok*).

Correct Identifications. *Report Close of Auction.* Unit test cases 02, 08, and 09 revolve around reporting the closing of an auction, and are thus indeed related to all ETE tests except to *ETE06*. Nevertheless, each of them provokes a different state of the sniper when the auction closed event takes place. Therefore, the mapping should be the strongest for *ETE01* with test 02, *ETE02* with test 08, *ETE3/04* with test 11, and *ETE05* with test 09. The measurements for these relations accurately reflect those subtle differences.

Notifies Bid Details. Tests 33 and 34 are related to all of the ETE tests, except for *ETE01*, which does not make a bid. As *ETE02* exclusively focuses on bidding, the relation is correctly identified as the strongest for this test. For other tests they appear on ranks 6 and 7.

Mapping per Focus. Test case 03 which only bids correctly achieves the highest rank for test *ETE02*. Test case 10, related to winning an auction, maps to *ETE03/04*. Tests 05, 06 and 09, which address losing before the auction is closed are also correctly identified as highest-ranking results for *ETE05*. Test cases 35-37, and 12-15 are testing the reporting of a failing auction. They are correctly ranked as highly related to *ETE06*. *ETE06* is a good example to demonstrate the impact of the *relevancy support based on occurrence*, described in Section 4.3.1. Test cases 33 and 34 share more steps with *ETE06* than for example test cases 35 and 37. Both achieve just a similarity ranking of 0.2. Nevertheless, tests 35 and 37 reflect much stronger the focus of *ETE06*. Because 35 and 37 are never indicated as related to any other ETE test, the *relevancy support* pushes them to the top results. The new ranking of, for example test 35, is calculated as its

Listing 4.2: *Test case: isWonWhenAuctionClosesWhileWinning*

```
@Test public void
isWonWhenAuctionClosesWhileWinning() {
    assertEquals(SniperState.LOST, SniperState.JOINING.whenAuctionClosed());
    assertEquals(SniperState.LOST, SniperState.BIDDING.whenAuctionClosed());
    assertEquals(SniperState.WON, SniperState.WINNING.whenAuctionClosed()); }
```

similarity divided by the number of times it has been ranked as a result among all tests (i.e., 0.2 divided by 1/6).

Surprises. *Winning and State Transitions.* A surprise was the ranking of test case 20 “isWonWhenAuctionClosesWhileWinning” within the results of *ETE01*, as the name suggests it is rather related to winning (i.e., *ETE03/04*). Inspecting the code, illustrated in Listing 4.2, reveals that the name is misleading as it tests different auction outcomes. Two times the auction is lost, contrary to the name, and it also triggers the rarely addressed state of *ETE01* (i.e., “joining” when the auction is closed). Test case 18 also triggers the transition between each stage and therefore should have a low relation to each of the test cases.

Not bidding, bidding and losing. Test cases 05 and 06, contrary to their name suggestions, do place bids and lose and are therefore also related to other test cases than *ETE06*. Actually only test case 32 does not make a bid, which is correctly mapped to *ETE01* and gets low ratings for the other tests. Since test case 06 also reaches the winning state before losing, the indicated relation to *ETE03/04* is understandable.

Defects and a Failing Auction. We expected test cases 21, 22 to be related to *ETE06*. But, tests 21 and 22 create a different failure as they put the system in a faulty state and assert a specific exception to be thrown. Such a behavior is not triggered in the end-to-end test, and consequently the non-appearance of those test cases for any ETE is correct.

Deviations. *Reporting winning.* Test case 11, which reports that an auction has been won after closing is ranked as the third result for *ETE02* even though this end-to-end test addresses losing. The common events, such as starting an auction, bidding for an item and closing an auction dominate the ranking.

Additional Lessons Learned. *Common functionality.* Some functionality is common to all tests. For example, tests of the class “*SniperTablesModelTest*” check the rendering of the user interface. Tests 01, 16, and 17 trigger common functionality such as adding a sniper and listeners. Such trace events are reduced and can yield empty test cases. Traces reduced to empty traces are marked as common functionality in the ranking.

4.6.3 RQ3: Handling Mocking

The test suite of Auction Sniper makes heavy use of the mocking library JMock. Without explicitly addressing mocked types during the analysis test cases in-

ETE 01				ETE 02			
test	<i>sim</i>	avg	<i>match</i>	test	<i>sim</i>	avg	<i>match</i>
02	1.20	1.73	optimal	03	0.55	1.95	optimal
09	0.67	2.50	ok	08	0.55	2.88	optimal
08	0.67	2.88	ok	11	0.45	3.15	nok
11	0.67	3.15	ok	33	0.44	1.85	ok
20	0.50	0.61	ok	34	0.44	1.85	ok

ETE 03 & 04				ETE 05			
test	<i>sim</i>	avg	<i>match</i>	test	<i>sim</i>	avg	<i>match</i>
11	0.64	3.15	optimal	05	0.55	1.95	optimal
06	0.55	2.13	ok	06	0.55	2.13	optimal
08	0.45	2.88	ok	09	0.55	2.50	optimal
10	0.44	1.57	ok	08	0.45	2.88	ok
15	0.44	1.77	ok	11	0.45	3.15	ok

Table 4.2: Top 5 similarity rankings for ETE01 to ETE05

volving mocked classes are ranked very low or as unrelated even though they are highly related. For example, without the mockery aspect test case 35 is not linked to test *ETE06* as the runtime types differ. By addressing mockery classes as described in Section 4.2.2 we can correctly identify test relations.

4.7 Discussion

Lessons learned and limitations. *Separation of Set-up and Tear-down.* Consistent usage of set-up and tear-down methods improves the similarity results, as it helps in revealing the core functionality and focus of test cases. Test suites which a priori do not use set-up and tear-down methods to structure their test might yield less accurate results.

Performance. The performance of the approach is an important criterion especially if the size and complexity of the system under study increases. During our two case studies, we experienced no performance issues with the systems un-

ETE 06							
test	<i>sim</i>	avg	match	test	<i>sim_s</i>	avg	match
12	0.50	1.59	optimal	13	1.20	0.20	optimal
15	0.50	1.77	optimal	35	1.20	0.20	optimal
14	0.40	1.22	optimal	36	1.20	0.20	optimal
33	0.40	1.85	ok	37	0.60	0.10	optimal
34	0.40	1.85	ok	12	0.60	1.59	optimal
03	0.40	1.95	ok	15	0.60	1.77	optimal
05	0.40	1.95	ok	14	0.48	1.22	optimal
06	0.40	2.13	ok	33	0.48	1.85	ok
10	0.30	1.57	nok	34	0.48	1.85	ok
08	0.30	2.88	nok	03	0.48	1.95	ok

Table 4.3: Similarity rankings for ETE06 with and without support

Test Case Name	Test Case	Relation
sniperJoinsAuctionUntilAuctionCloses – ETE01		
notifiesAuctionClosedWhenCloseMessageReceived	≡ 32	hi. rel.
reportsLostWhenAuctionClosesImmediately	≡ 02	hi. rel.
isWonWhenAuctionClosesWhileWinning	≡ 20	related
reportAuctionClosesX	≡ 08, 09, 11	related
sniperMakesAHigherBidButLoses – ETE02		
reportsLostIfAuctionClosesWhenBidding	≡ 08	hi. rel.
bidsHigherAndReportsBiddingWhenNewPriceArrives	≡ 03	hi. rel.
doesNotBidAndReportsLosingIfSubsequentPriceIsAboveStopPrice	≡ 05	related
doesNotBidAndReportsLosingIfPriceAfterWinningIsAboveStopPrice	≡ 06	related
reportAuctionClosesX	≈ 09, 11	related
sniperWinsAnAuctionByBiddingHigher – ETE03 & sniperBidsForMultipleItems – ETE04		
reportsWonIfAuctionClosesWhenWinning	≡ 11	hi. rel.
reportsIsWinningWhenCurrentPriceComesFromSniper	≡ 10	hi. rel.
doesNotBidAndReportsLosingIfPriceAfterWinningIsAboveStopPrice	≡ 06	related
reportAuctionClosesX	≡ 08, 09	related
sniperLosesAnAuctionWhenThePriceIsTooHigh – ETE05		
reportsLostIfAuctionClosesWhenLosing	≡ 09	hi. rel.
doesNotBidAndReportsLosingIfSubsequentPriceIsAboveStopPrice	≡ 05	hi. rel.
doesNotBidAndReportsLosingIfPriceAfterWinningIsAboveStopPrice	≡ 06	hi. rel.
doesNotBidAndReportsLosingIfFirstPriceIsAboveStopPrice	≡ 04	hi. rel.
continuesToBeLosingOnceStopPriceHasBeenReached	≡ 07	hi. rel.
(reportAuctionClosesX)	≡ 08, 11	related
sniperReportsInvalidAuctionMessageAndStopsRespondingToEvents – ETE06		
notifiesAuctionFailedWhenBadMessageReceived	≡ 35	hi. rel.
notifiesAuctionFailedWhenEventTypeMissing	≡ 36	hi. rel.
writesMessageTranslationFailureToLog	≡ 37	hi. rel.
reportsFailedIfAuctionFailsWhenBidding	≡ 12	hi. rel.
reportsFailedIfAuctionFailsImmediately	≡ 13	hi. rel.
reportsFailedIfAuctionFailsWhenLosing	≡ 14	hi. rel.
reportsFailedIfAuctionFailsWhenWinning	≡ 15	hi. rel.
ETE 01 – 06		
transitionsBetweenStates	≡ 18	related
ETE 02 – 06		
bidsHigherAndReportsBiddingWhenNewPriceArrives	≡ 03	related
ETE 02 - 05		
notifiesBidDetailsWhenCurrentPriceMsgReceivedFromOtherBidder	≡ 33	related
notifiesBidDetailsWhenCurrentPriceMsgReceivedFromSniper	≡ 34	related
Common functionality and UI		
UI related tests (e.g. test of class SniperTablesModelTest)	≡ 23 – 31	related
Listeners and common states	≡ 01, 16, 17	related
Functionality not addressed by any ETE		
defectIfAuctionClosesWhenWon	≡ 21	unrel.
defectIfAuctionClosesWhenLost	≡ 22	unrel.

Table 4.4: Final conceptual mapping of end-to-end tests to unit tests. (Abbreviation *hi. rel.* stands for highly related and *unrel.* for unrelated.)

der study. For larger systems further trace reduction techniques might become necessary (Cornelissen et al., 2008). On the other hand, performance depends more on the size of the traces (i.e., amount of functionality covered by a test), than on the number of tests. Test case size is independent of the complexity and

size of the systems.

Future work. *Assertions.* At this stage, our technique does not address the meaning of assertions. As future work, we would like to investigate how the meaning of assertions can influence the ranking of a test case.

Test suite quality inspection. The discovered relations do not only help to see similarity of test cases, they also help to assess the quality of the test suite and discover areas for improvement, e.g., identifying unit test cases that do too much, or identifying behavior which is not addressed by any end-to-end test.

User study. We aim to further investigate the usefulness of our tool through a user study that allows actual developers and testers to work with it.

Threats to validity. Concerning *external validity*, our case studies address relatively small Java systems. Scalability to larger case studies is a key concern that we aim to address in our future work, making use of case studies from the Eclipse plug-in domain we used in earlier studies (Mylyn, EGit) (Greiler et al., 2010).

With respect to *internal validity*, the main threat consists of the manually obtained conceptual mapping. Creating such a mapping is inherently subjective, as illustrated by the process we applied to the Auction Sniper case study.

In order to reduce *threats to reliability* and to improve repeatability, both our tool and the systems under study are available to other researchers.

4.8 Related Work

An initial catalogue of *test smells* negatively affecting understanding was presented by van Deursen et al. (2002), together with a set of corresponding refactorings. Later, a thorough treatment of the topic of refactoring test code was provided by Meszaros (2007). Van Rompaey et al. (2007) continued this line of work by studying automated analysis of these smells.

Tools for assisting in the understanding of test suites have been proposed by Cornelissen et al. (2007), who present a visualization of test suites as sequence diagrams. Greiler et al. (2010) propose higher level visualizations, aimed at assisting developers in seeing plug-in interactions addressed by their test suites.

Galli et al. (2004b) have developed a tool to order broken unit tests. It is their aim to create a hierarchical relation between broken unit tests, so that the most specific unit test that fails can be inspected first. In essence, their technique allows to steer and optimize the debugging process.

Rothermel and Harrold (1998) discuss safe regression testing techniques; regression test selection techniques try to find those tests that are directly responsible for testing the changed parts of a program and subsequently only run these tests. Hurdugaci and Zaidman (2012) operationalize this in the IDE for unit tests.

Yoo et al. (2009) cluster test cases based on their similarity to support experts in test case prioritisation, which outperforms coverage-based prioritisation.

4.9 Conclusion

In this paper we showed how a combination of *dynamic analysis* and the shared word count metric can be used to establish relations between end-to-end and unit tests in order to assist developers in their (test suite) maintenance activities.

We evaluated our *test connection mining* techniques in two case studies, by elaborating the usefulness of the approach to improve understanding. We saw that after using the proposed trace reduction techniques our approach produces accurate test mappings, which can help to 1) identify relevant tests, 2) understand the functionality of a test by describing high-level test cases with related unit test cases and 3) reveal blank spots in the investigated unit test suites.

Contributions. The contributions of this paper are 1) tracing and trace reduction techniques tailored for handling test code, including test specific events such as set-up, tear-down and mocking 2) an assessment of the usefulness of the rankings based on two case studies, 3) the development of a *Test Similarity Correlator*, a framework for mining test connections.

Automated Detection of Test Fixture Strategies and Smells

Abstract

Designing automated tests is a challenging task.¹ One important concern is how to design test fixtures, i.e. code that initializes and configures the system under test so that it is in an appropriate state for running particular automated tests. Test designers may have to choose between writing in-line fixture code for each test or refactor fixture code so that it can be reused for other tests. Deciding on which approach to use is a balancing act, often trading off maintenance overhead with slow test execution. Additionally, over time, test code quality can erode and test smells can develop, such as the occurrence of overly general fixtures, obscure in-line code and dead fields. In this paper, we show that test smells related to fixture set-up occur in industrial projects. We present a static analysis technique to identify fixture related test smells. We implemented this test analysis technique in a tool, called TestHound, which provides reports on test smells and recommendations for refactoring the smelly test code. We evaluate the tool through three industrial case studies and show that developers find that the tool helps them to understand, reflect on and adjust test code.

¹This chapter appeared in the proceedings of the 2013 International Conference on Software Testing, Verification and Validation (ICST 2013). The authors of this publication are Greiler, van Deursen, and Storey.

5.1 Introduction

Modern software development practice dictates early and frequent (automated) testing. While automated test suites written by developers are helpful from a (continuous) integration and regression testing perspective, they lead to a substantial amount of test code. Like production code, test code needs to be maintained, understood, and adjusted, which can become very costly. The long term success of automated testing is highly influenced by the maintainability of the test code (Meszaros, 2007). To support easier maintainability of a system, test methods should be clearly structured, well named and small in size (Freeman and Pryce, 2009). The duplication of code across test methods should be avoided.

One important part of a test is the code that initializes the system under test (SUT), sets up all dependencies and puts the SUT in the right state to fulfill all preconditions needed to exercise the test. In line with Meszaros (2007), we refer to this part of a test as the *test fixture*. Developers can adopt several strategies for structuring their fixture code. The most straightforward option is to place the setup code directly in the test method, which we refer to as an *in-line setup*. A positive aspect of an *in-line* setup is the proximity of the setup code to the test itself. However, when several test methods require the same fixture, an *in-line* setup can lead to code duplication and high maintenance costs (van Deursen et al., 2001). Also, configuring the SUT within the test method might hide the main purpose of the test and result in an *obscure test* (Meszaros, 2007).

An alternative approach is to place the setup code in helper methods that can be called by several test methods, which we refer to as a *delegate setup* (Meszaros, 2007). With a delegate setup, the developer has to make sure the right methods are invoked at the right time (e.g. as a first statement in a test method).

In today's testing frameworks, such as the widely used xUnit family, there is a dedicated mechanism to manage setup code invocations (Beck, 2002; Gamma and Beck, 2003). Therefore, helper-methods containing the setup code can be marked (e.g. using annotations or naming conventions) as specific setup methods, which we refer to as an *implicit setup*.² The advantage of an *implicit setup* is that the framework takes care of invoking the setup code at a certain point in time and for a specific group of tests, but also that the methods are explicitly marked as setup which helps with code comprehension. Often, *implicit* setups are invoked either before each test within a class, or once before all the tests within a class. One main drawback of this approach is that the tests grouped together (i.e. within one class) should have similar needs in the test fixture. Otherwise, tests might only access (small) portions of a broader fixture, which can lead to slow tests and maintenance overhead.

During the evolution of test code, developers have to make conscious decisions about how to set up the test fixture and adjust their fixture strategies, otherwise

²For example, in the JUnit frameworks methods can be either named `setUp()` or marked with annotations such as `@Before` or `@BeforeClass`.

they end up with poor solutions to recurring implementation and design problems in their test code, so-called *test smells* (van Deursen et al., 2001). Unfortunately, until now, no support has been made available to developers during the analysis and adjustment of test fixtures.

To address this shortcoming, we developed a technique that automatically analyzes test fixtures to detect fixture-related smells and guides improvement activities. We implemented this technique in *TestHound*, a tool for static fixture analysis. We evaluate our technique in a mixed methods research approach. First, we analyzed the test fixtures of three industry-strength software systems. Second, we evaluated the usefulness of the technique with 13 developers. In the paper, we show that fixture-related smells exist in practice, and that developers find TestHound helpful during fixture management.

In Section 5.2, we briefly summarize different test smells related to test fixtures. In Section 5.3, our fixture analysis technique is presented, followed by implementation details in Section 5.4. Section 5.5 details our experimental design. In Section 5.6, the evaluation of our technique is presented, followed by a discussion in Section 5.7. In Section 5.8, we present related work, and conclude in Section 5.9.

5.2 Test Smells

In earlier research (Greiler et al., 2012a), we interviewed 25 Java developers on information needs for test code understanding. We observed that the test structure is important for developers to navigate and retrieve tests within a code base. For example, to support easier retrieval of test code, it is a common practice in Java-based systems to organize tests similar to production code (i.e. class to test class, package to test package). Although this practice is chosen to facilitate maintenance, it might lead to groups of tests within one test class that have very different requirements on the system under test. This means that each test might need a different *test fixture* that initializes and configures the system under test and all its dependencies (to fulfill all preconditions of a test). As test code grows and evolves, this strategy can lead to *test smells* with respect to the test fixture.

The code smell metaphor has been introduced by Fowler (1999) who describe a smell as a poor solution to a recurring implementation and design problem. Code smells are not a problem per se, but they may lead to issues such as understanding difficulties, inefficient tests and poor maintainability of a software system. Later, van Deursen et al. (2001) introduced the term *test smells* by applying the concepts of smells to test code. The initial set of test smells has been extended by several researchers (Meszaros, 2007; Van Rompaey et al., 2006; Neukirchen and Bisanz, 2007). We further extend this set, in particular, with test smells related to test fixtures. Apart from the General Fixture Smell (introduced by van Deursen et al. (2001)), we present five new test smells as well as possible refactorings to address these issues:

General Fixture Smell. The general fixture smell occurs if test classes contain broad functionality in the *implicit* setup, and different tests only access part of the fixture. Problems caused by a general fixture are two-fold: firstly, the cause-effect relationship between fixture and the expected outcome is less visible, and tests are harder to read and understand. This can cause tests to be fragile: a change that should be unrelated affects tests because too much functionality is covered in the fixture. Secondly, the test execution performance can deteriorate, and test execution times may eventually lead to developers avoiding to execute tests.

Refactoring. A general fixture can be refactored by creating a minimal fixture, which covers only the setup code common for all test methods. Individual setups can be placed in delegate setups by applying an extract method refactoring. In the case where the test methods do not share too much setup code, an extract class refactoring can be applied.

Test Maverick. Based on the general fixture smell, we derived a related smell: the test maverick smell. A test method is a maverick when the class comprising the test method contains an *implicit* setup, but the test method is completely independent from the implicit setup procedure. The setup procedure will be executed before the test method is executed, but it is not needed. Also, understanding the effect-cause relationship between setup and test method can be hampered. Discovering that test methods are unrelated from the implicit setup can be time consuming. *Refactoring.* Test mavericks can be eliminated by the *extract class refactoring*, placing them in their own class.

Dead Fields. The dead field smell occurs when a class or its super classes have fields that are never used by any test method. Often dead fields are inherited. This can indicate a non-optimal inheritance structure, or that the super class conflicts with the single responsibility principle. Also, dead fields within the test class itself can indicate incomplete or deprecated development activities. *Refactoring.* Dead fields associated with the class should be removed. A possible refactoring for dead fields of the super class are splitting the super class into several classes.

Lack of Cohesion of Test Methods. Cohesion of a class indicates how strongly related and focused the various responsibilities of a class are (Chidamber and Kemerer, 1994). Classes with high cohesion facilitate code comprehension and maintenance. Low cohesive methods are smelly because they aggravate reuse, maintainability and comprehension (Fowler, 1999; Li and Henry, 1993). The smell Lack of Cohesion of test methods (LCOTM) occurs if test methods are grouped together in one test class, but they are not cohesive. *Refactoring.* To reduce LCOTM, the extract class refactoring can be applied to split a test class with too many test responsibilities into different classes.

Obscure In-Line Setup. Meszaros (2007) introduced the smell obscure test to refer to a test that is difficult to understand and thus is not suitable for documentation purposes. Based on this smell, we created the obscure in-line setup. An *in-line* setup should consist of only the steps and values essential to understanding the test. Essential but irrelevant steps should be encapsulated into helper methods. An obscure in-line setup covers too much setup functionality within the

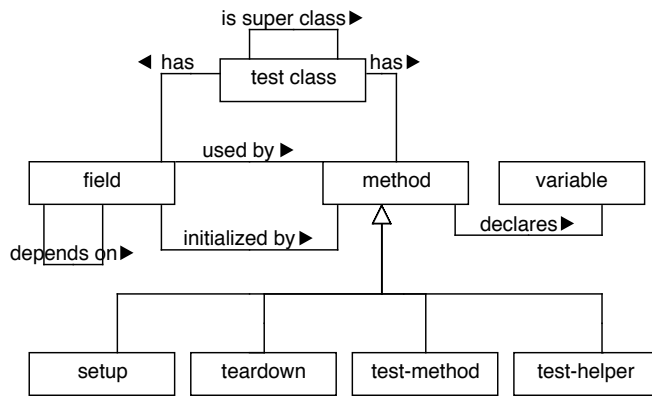


Figure 5.1: *Meta-model Test Fixture Analysis*

test method. This can hinder developers in seeing the relevant verification steps of the test. *Refactoring.* To conquer obscure in-line setups, the setup code can be moved into delegate setup methods, or if the in-line setup is common to all tests, one can use an *implicit setup*.

Vague Header Setup. A vague header setup smell occurs when fields are initialized in the header of a class, but not in *implicit* setup. We consider this a smell as the behavior of the code is not explicitly defined, and depends on the field modifier (static or member), as well as on the implementation of the test framework. Further, field declarations are not restricted to the header of a class, but can occur anywhere within the class. Vague header setups hamper code comprehension and maintainability. *Refactoring.* Field initializations should be placed in an *implicit* setup to specify the behavior and the places to inspect within a class.

5.3 Analysis of Fixture Usage

This section describes the technique we developed to analyze the test fixture organization, fixture usage and fixture smells, and suggest refactorings for the test code. Our reverse engineering technique follows the well-known reconstruction approach: fact extraction, abstraction, and presentation (Tilley et al., 1996).

5.3.1 Fact Extraction

To determine fixture strategies and fixture-related smells we extract several facts for each test class. All relevant entities for our analysis are illustrated in the meta model in Figure 5.1. Firstly, we identify all methods in a class. We differentiate between test methods, setup methods, tear-down methods and test helper

Table 5.1: *Smell indicators*

Indicator	Description
<i>setupFlds</i>	Fields set in implicit setups or class header.
<i>usedSetupFlds</i>	SetupFlds used in test methods.
<i>adHocFlds</i>	Fields solely initialized in test methods.
<i>deadFlds</i>	SetupFlds fields never used in any test method.
<i>localVars</i>	Variables declared in a test method.
<i>headerInit</i>	Fields initialized in the class header.

Table 5.2: *Smell Metrics and Thresholds*

Smell name	Metric
General Fixture	$\frac{usedSetupFlds}{setupFlds - deadFlds} \leq 0.7$
Test Maverick	$usedSetupFlds \equiv 0 \wedge setupFlds \geq 1$
LCOTM	$LCOTM \geq 0.4$
Dead Field	$ deadFlds \geq 1$
Vague Header Setup	$ headerInit \geq 1$
Obscure In-line Setup	$ localVars \geq 10$

methods based on the method’s annotation or naming conventions.³ Further, we extract all global fields of the class, and all local variables for each of the test methods.

5.3.2 Analysis

The analysis consists of two steps. First, we derive indicators for smells based on the extracted facts as summarized in Table 5.1. Second, we use those indicators to measure the existence of test smells based on our metrics (see Table 5.2).

Implicit Fixture Usage Indicators. To determine how much a test class and its test methods use the *implicit* setup, we derive smell indicators *setupFlds*, *usedSetupFlds* and *deadFlds*. *setupFlds* are fields that are initialized in the *implicit* setup procedures or the class header. For example, in Listing 5.1, the fields *repository*, *repository2*, *gitDir* and *store* are seen as *setupFlds*. *usedSetupFlds* represents the number of *setupFlds* of the class that have also been accessed (i.e. read or write) by a test method. This access can happen directly in the test method or via (a chain of) helper methods invoked by the test method. A field only accessed in the setup, but by no test method is not seen as used. In Listing 5.1, test method *testRepository()* uses fields *repository* and *dir*, whereas the fields *repository2* and *store* are not used. Further, we *estab-*

³This depends on the particular test framework.

lish field dependency relationships to determine whether a setup field is used by a test method. Setup fields depend on each other if one field f_a is used to set another field f_b (e.g. $f_b = f_a$). To extract these relations, we have to analyze the data flow. We detect direct assignments, but also whether a field f_a depends on another field f_b based on method calls (e.g., $f_b.set(f_a)$). Hereby, the field used for the method call (f_b) is seen as dependent on the fields (f_a) used as parameter. This means that in Listing 5.1, test method *testRepository()* also uses field *gitDir*, as *repository* depends on *gitDir*. Finally, *deadFields* are fields that are initialized in the *implicit* setup, but are never used by a test method. In our example, field *repository2* is never used by any test method.

Measuring General Fixtures. To identify general fixtures, we calculate the ratio between how many *usedSetupFlds* a test method has, and how many *setupFlds* exist in the class. If this ratio is below a certain threshold, we identify it as a general fixture test method. In our current experiments, we set the threshold to 70%. We leave determining the optimal thresholds, for example through benchmarking, for future work.

Listing 5.1: *Test Class Example*

```
class BlobStorageExampleTest extends GitTestCase {
    //setup field
    Repository repository;
    Repository repository2;
    //header initialization
    Storage store = new Storage();
    //ad hoc field
    Directory dir, gitDir;

    @Before public void setUp() throws Exception {
        super.setUp();
        gitDir = new Directory(".");
        //repository depends on gitDir
        repository = new FileRepository(gitDir);
        repository2 = new FileRepository(gitDir);
    }
    ...
    @Test public void testRepository() {
        dir = new Directory(".");
        loadFile();
        ...}

    private void loadFile() {
        repository.getFile("testfile");
        ...}}

```

Measuring Test Mavericks. If a class has an implicit setup, i.e. has *setupFlds*, and a test method does not use any of the *setupFlds*, we identify a test method as detached from the test class setups.

Measuring Dead Fields. Dead fields are all fields that are initialized by the *implicit setup*, but that are never used by any test method. We differentiate between dead fields inherited by the super class (i.e. inherited fields) and dead

fields declared in the class itself. Note that an IDE would not identify *deadFlds*, because an IDE only shows whether a field is never used within a class. Our analysis reveals whether a field is never used by a test method, even if it is accessed within a helper method or the implicit setups.

Cohesion Indicator. To address how cohesive test methods are in a class, we need another smell indicator: fields solely initialized in test methods but not in the setup procedure. We call these fields *adHocFlds*, as they are only created when they are needed. In Listing 5.1, the field *dir* is an *adHocFld*.

Measuring Lack of Cohesion in Test Methods. To measure how cohesive the test methods of a test class are, we adjusted the Henderson-Seller Lack of Cohesion of Method metrics (Henderson-Sellers, 1996). Differing from the original metric, we exclusively calculate the cohesion between the test methods in a class and exclude any other methods (e.g. helper or setup methods). For our analysis, we consider all fields of the test class (i.e. *setup* and *ad hoc* fields) that have been used (i.e. we exclude dead fields). We calculate the Lack of Cohesion of Test Methods as the following:

$$\text{LCOTM} = \frac{\frac{1}{|F|} * \sum_{i=1}^{|F|} r(f_i) - |M|}{1 - |M|}$$

Where M is the set of test methods defined by the class, F is the set of *setupFlds* and *adHocFlds* (without *deadFlds*) of the class, and $r(f_i)$ is the number of test methods that access field f_i and f_i is a member of F . As we do not consider *deadFlds*, the metric reports a value between 0 and 1, with 0 indicating no lack of cohesion and 1 highest lack of cohesion. We choose 0.4 as an indicator for a smelly test class.

The LCOTM complements the metric for test mavericks and general fixtures, as it also addresses *adHocFlds* and thus reflects on how strongly test methods differ from each other independent of the fixture.

Obscurity Indicator. The counterpart to the implicit setup, is the *in-line* setup. We measure the obscurity of an in-line setup based on the number of local variables directly defined within a test method (i.e. *localVars* indicator).

Measuring obscure in-line setup. We detect an obscure in-line setup if the number of *localVars* exceeds a certain threshold (i.e. 10 variables per method). The rationale behind this threshold is that with the increasing length of the test method, the primary focus of the test may be hidden. The chosen threshold follows the best practices for the length of a method.

Header Indicator. Finally, the last smell indicator is the fields initialized in the header of the class (i.e. *headerInit*).

Measuring Vague Header Setup. We report this when at least one field is initialized in the header of the class.

Testcase class	No. Tests	No. Flds (Setup/ All)	Dead Flds	LCOM	General Fixture	Test Maverick	Obscure Inline	Vague Header
Test Suite org.eclipse.egit.core--All-Tests								
org.eclipse.egit.core.test.op.AddOperationTest	6	5/ 5		0.0				!
org.eclipse.egit.core.GitMoveDeleteHookTest	19	4/ 8		0.04				!
org.eclipse.egit.core.test.indexDiff.IndexDiffCacheTest	1	5/ 5	! 1 deads super: 1/ 3	0.0				!
org.eclipse.egit.core.test.op.RemoveFromIndexOperationTest	5	5/ 5		0.0				!
org.eclipse.egit.core.securestorage.IGitSecureStoreTest	13	2/ 2		0.0				
org.eclipse.egit.core.synchronize.ThreeWayDiffEntryTest	9	9/ 9	! 5 deads super: 5/ 8	0.0				!
org.eclipse.egit.core.test.GitProjectSetCapabilityTest	4	3/ 3		0.0			! 2 methods	!
org.eclipse.egit.core.test.op.CommitOperationTest	7	7/ 7		0.12				!
org.eclipse.egit.core.internal.storage.BlobStorageTest	6	4/ 4		0.5	! 2 methods	! 1 detached	! 1 methods	!

Figure 5.2: Excerpt of the Test Fixture Smell Report for eGit

5.3.3 Presentation

This section explains how we present the information gathered in the analysis. We chose to use a navigable hypertext report to present the outcome to the developers, thus supporting a seamless navigation between overviews and details. The report is split into three parts: the fixture classification, the smell overview and the detail improvement report.

Fixture Classification Report. This report provides a list-based overview of the fixture strategies and used framework mechanisms of all test classes. Further it highlights the inheritance structures.

Test Fixture Smell Report. This report provides an overview of the test smells, also in the form of a list, as illustrated by Figure 5.2. The smells are indicated by an icon and, where relevant, a number showing how often the smell occurred within the test class. To get detailed information about the test class, the developer can click on the test class name and drill into the detail improvement report.

Detail Improvement Report. This report provides detail information on the analysis outcome for a single test class. In the first part of the report, a summary of all smells of the class is given, including a detailed description of the cause. Further, each smell description is enhanced with refactoring suggestions, as illustrated in Figure 5.3. The second part of the report outlines how fields and helper methods are used within each test method of the class. This part details information on the fixture usage, which is hard to obtain from the IDE and the code alone. It is designed to guide refactoring decisions and to support the developer during the smell assessment.





General Fixture			
2 test methods use less than 70% of the fields set during test setup.			
Methods affected: testFailNotFound , testFailWrongType .			
Minimal Fixture			
After removing detached methods from class, refactor the large fixture to a minimal fixture.			
Minimal Fixture: repository gitDir			
Hint: Only half of the fields are shared. Consider an Extract Class refactoring.			
Extract Class 1: testOk			
Extract Class 2: testFailNotFound testFailWrongType testFailCorrupt testFailCorrupt2			
Setup Detail 4 member fields, 0 static fields			
Fields set in Setup		Usage	Inherited
repository		5 out of 6 test methods use this field	
project		3 out of 6 test methods use this field	GitTestCase
gitDir		5 out of 6 test methods use this field	GitTestCase
testUtils		1 out of 6 test methods use this field	GitTestCase

Figure 5.3: Excerpt Detail Improvement Report for *eGit - BlobStorageTest*

5.4 Implementation and Tool Architecture

TestHound is implemented in Java and supports languages which compile to Java byte code by using the Apache BCEL library to extract facts. *TestHound* supports the JUnit and TestNG test frameworks, but can easily be extended to support other frameworks. Although *TestHound* supports only Java, the analysis is language and framework independent and only the facts extraction aspect is language specific. For the generation of the hypertext report, we use the StringTemplate engine.⁴ *TestHound* is available for download⁵ and we are in the process of making the source code available on GitHub. In a future release, the tool will be available as a Maven⁶ plug-in to facilitate integration with the continuous integration process.

5.5 Experimental Design

This section outlines the experimental design of the study, including the research questions, case studies, interviews and questionnaires.

5.5.1 Research Questions

To evaluate the applicability and helpfulness of our technique, we set out to investigate the following research questions:

RQ1 What do the structure and organization of test fixture look like in practice?

⁴<http://www.stringtemplate.org/>

⁵<http://swierl.tudelft.nl/bin/view/MichaelaGreiler/TestHound>

⁶<http://maven.apache.org/>

RQ2 Do fixture-related test smells occur in practice?

RQ3 Do developers recognize these test smells as potential problems?

RQ4 Does a fixture analysis technique help developers to understand and adjust fixture management strategies?

To answer our four research questions, we applied a mixed methods research approach. To answer RQ1 and RQ2, we applied case study research and investigated the code bases of three different Java-based software systems. To answer RQ3 and RQ4, we used interviews and a questionnaire.

5.5.2 Case Studies

We use three different subject systems in our experimental design - one closed and two open source systems.

HealthCare: Closed Source Health Care System. The first subject system is developed by a company based in Canada, that offers health care related software solutions. Part of the system is a Java back-end, which provides an API to other systems. This back-end comprises 750K lines of code and has 945 test methods, using the *TestNG*⁷ framework.

eGit: Open Source Eclipse-Integrated Version Control System. eGit⁸ integrates the Git version control system into the Eclipse IDE. It consists of 130K lines of code and has 479 test methods, all written in *JUnit*⁹.

Mylyn: Open Source Task Management System Mylyn¹⁰ provides task management functionality within the Eclipse IDE. It consists of 500K lines of code and has 1644 test methods, written in JUnit.

5.5.3 Interviews and Questionnaire

We set out to evaluate our tool and technique by presenting it in a one hour session to a group of 13 professional software developers. These developers worked for the company of the HealthCare system. All developers have experience in writing and maintaining test code, and approximately half of the participants have been working on the code base of the HealthCare system. In this session, we covered the general functionality and purpose of TestHound, as well as the report produced for the HealthCare system. After the presentation, we interviewed five software developers who had contributed to the code base, with each interview taking 30 minutes. During these interviews, the participants could browse through the report produced by TestHound, ask questions and express their opinions on TestHound in depth. We recorded and transcribed each interview.

⁷<http://testng.org>

⁸<http://www.eclipse.org/egit/>

⁹<http://www.junit.org>

¹⁰<http://www.eclipse.org/mylyn/>

To capture the opinions of all participants of the presentation, we designed a questionnaire addressing the perception of the audience on software maintenance and the helpfulness of “TestHound”. The questionnaire was filled in by all 13 developers and is available online.¹¹

Pilot Sessions. To improve the experimental design of the interviews and questionnaire, we conducted three pilot sessions with experienced testers. Two pilot participants were co-workers, and the third participant was the second author of this paper.

5.6 Evaluation

5.6.1 RQ1: What do the structure and organization of test fixture look like in practice?

This section highlights the basic structure and organization of the test code we analyzed. The results are summarized in Table 5.3.

Package Structure. In all three case studies, the package structure of the test code closely followed the package structure of the system under test. In eGit and the HealthCare system, test code and production code is not separated by an additional package (e.g. test). In contrast, for the Mylyn system, the package “org.eclipse.mylyn.commons.core” is mapped to the package “org.eclipse.mylyn.commons.tests.core”. In all three systems, the test code is often mapped to classes. For example, in Mylyn, the class “CoreUtil” is tested by the class “CoreUtilTest”.

In the HealthCare system, this mapping is followed rigorously, and this design decision has a significant impact on the modularity of the test code. The test code of this system consists of only 36 test classes that comprise 933 unique test methods (two of which are parameterized tests). Some of the test classes have more than 4,000 lines of code. For example, one test class comprises 112 test methods and approximately 4,500 lines of code. The Mylyn system consists of 232 test classes that comprise 1,644 test methods. Three of these test classes comprise more than 100 test methods, with a maximum of 172 test methods in the TextileLanguageTest (i.e. more than 1,500 lines of code). In eGit, 87 test classes comprise 479 test methods. The test class with the most tests has 19 tests and 600 lines of code.

Framework Fixture Functionality. In all three systems, the majority of the tests use the *implicit* setup mechanisms of the test frameworks. Interestingly, in the HealthCare system, only the functionality to automatically invoke *implicit* setups, either before one class or one test method, is used. The more fine-grained directives which TestNG offers are not used. In the eGit system, several separate test suites exist and the usage pattern of the *implicit* setup constructs differs: the test suite addressing the core of the system often invokes the setups before each

¹¹<http://swirl.tudelft.nl/bin/view/MichaelaGreiler/TestHound>

Table 5.3: *Fixture Management Strategies*

Project	#test classes	#test methods	Implicit setup		No setup	Tear down
			member	class		
eGit	87	479	56	47	5	79
HealthCare	36	933	26	25	9	25
Mylyn	232	1644	164	0	68	152

Table 5.4: *Fixture Problems*

Project	General Fixture		Test Maverick		LCOTM	
	#methods	pct.	#methods	pct.	#classes	pct.
HealthCare	299	≈ 32%	84	≈ 9%	7	≈ 19.4%
Mylyn	377	≈ 23%	82	≈ 5%	36	≈ 15.5%
eGit	65	≈ 13.5%	17	≈ 3%	12	≈ 13.8%

Project	Dead Field		Obscure In-line		Vague Header	
	#fields	pct.	#methods	pct.	#classes	pct.
HealthCare	180	≈ 33%	100	≈ 10.7%	26	≈ 72%
Mylyn	66	≈ 12.1%	17	≈ 1%	35	≈ 15%
eGit	110	≈ 23.6%	8	≈ 1.6%	79	≈ 91%

test method, and the fields are mostly non-static. On the other hand, in the test suites addressing the user interface functionality, setups are most often invoked before each class and the fields are static. This design decision is probably due to performance considerations. User interface-related tests often need more setup and involve more expensive resources. In Mylyn, only the *implicit* setups that are executed before each test are used. In all systems, the tear down mechanisms of the test frameworks are used less frequently than the setup mechanisms.

5.6.2 RQ2: Do fixture related test smells occur in practice?

Table 5.4 summarizes all smells detected in the three projects, whereby showing the absolute number and the percentage of entities affected by a smell. Each of the smells occurred several times in practice. In the following, we will present some highlights.

General Fixture Smell. The general fixture smell occurred for 32% of the test methods in the HealthCare system, for 23% of the test methods in Mylyn, and for 13.5% of the tests in eGit. An example from the eGit system is *ProjectReferenceTest*. In this test class, none of the setup fields are used by all test methods (i.e. the fixture is disjointed). To improve the class design, an extract class refactoring is recommended. In the HealthCare system, only a few classes contribute the majority of general fixture methods. In particular, three classes comprise 172 of the 299 general fixture methods (i.e. ≈58%). In Mylyn, the largest test class with 172 tests contributes 168 general fixture methods. This class has only two fields, whereby one is only used by three test methods. In eGit, fewer general fixture methods are detected and they are more distributed among classes, as compared

with the other systems.

Test Mavericks Smell. Test mavericks occur less frequently than general fixture methods. Also, they are more distributed among classes. In the HealthCare system, the largest class (with 112 tests) contributes the largest set of detached methods (23 methods). In Mylyn, the class `TaskListExternalizationTest` has the largest number of test mavericks (10 out of 28). In eGit, 4 of the 8 methods in `ChangeTest` are test mavericks.

Dead Fields Smell. All three systems contains many dead fields, and most of them are inherited by super classes and not needed. In case fields declared in the actual test class are dead, it often seems to be because of obsolete functionality or open issues. In the HealthCare system, more dead fields exist than in the other systems. There are two main reasons: first, as discussed most tests inherited from only two large super classes and inherit fields that are never used. Second, in this system many static methods are access via the fields, which is unnecessary and often even not recommended. For example, via a field `context` the static method `getBean()` is invoked (i.e. `context.getBean()`), whereby `getBean()` should be access via the class.

Lack of Cohesion of Test Methods Smell. In the three systems, 14-19% of the classes have a LCOTM value greater or equal 0.4. In Mylyn, a class with high LCOTM (0.8) is the `EncodingTest`. Each of the test methods in the class uses different combinations of the `setUpFlds`. In eGit, an example of a test with high LCOTM is `ProjectReferenceTest`. Here, all test methods share one field, and in addition, each test method addresses an additional field. In the HealthCare system, the test class with the highest LCOTM (0.89) has two used `setUpFlds` that are only used by 4 out of 23 test methods.

Obscure In-line Setup Smell. In the HealthCare system, 10% of the methods contain an obscure in-line setup. The average number of variables declared within these tests is 14.4, with a maximum of 29 variables. In the Mylyn and eGit system, less than 2% of the test methods are reported to have an obscure in-line setup. In terms of test size, for example in Mylyn, a test method `testSynchChangedReports` in Class `BugzillaRepositoryConnectorTest` with 24 `localVars` has 113 lines of code.

Vague Header Setup. In the HealthCare system, header initializations occur in 72% of the test classes, and in eGit, in 91% of the test classes. In Mylyn, this smell occurs in only 15% of the test classes.

5.6.3 RQ3: Do developers recognize these test smells as potential problems?

During the tool demonstration and interviews, it became clear that developers do indeed recognize the reported test smells as potential problems, and that they see a strong connection between smelly tests and maintenance overhead. In the questionnaire, as illustrated in Figure 5.4, 12 of 13 developers agreed with the statement that wrong fixture management can lead to code quality problems, and

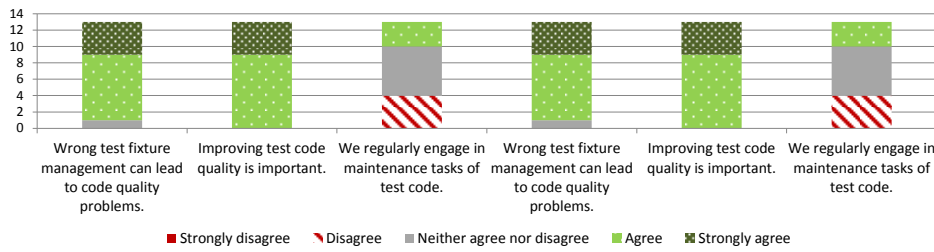


Figure 5.4: Answers of the questionnaire about maintenance attitude and tool expectations

all indicated that improving test quality is important. Only three indicated they they regularly engage in maintenance of test code, whereas four indicated that they do not regularly maintain test code. In the interviews, we investigated why test code is not regularly maintained. All of the interviewed developers said that they had expected their test code base to be very messy. Interviewee number two (i.e. P2) says: “We know our classes are too large and wrongly focused. We start to write test code and then, next step, we improve.” Soon it became clear that time for the next step is limited, as P1 says: “We do not have the option to say ‘Oh, that’s ugly, I’ll spend a day to clean it up’ if it does not give us immediate business.”

On the other hand, developers express that they are slowed down by the smelly classes (that our tool also identified). P1 says “If you have to debug a bug revealed by one of these large test classes, then it takes 5 minutes to run and it makes you think ‘I don’t want to do that anymore’.”

How do these test code quality problems emerge? P1 says that “people think: ‘Ah, this test has to do with a `ContainerType`’, and then add the test to the `ContainerTypeTest` class, even though it has nothing to do with the other tests in this class.” He adds “I skip the implicit setup because often the initial setup is made by the first person who created the class. Maintainers are adding stuff to this setup as they go along, but maybe this is not as common for the other methods and some methods are even unrelated. And then, you make it just in-line, ugly in-line, instead of using the provided framework functionality.” Also, other developers explain that they do not look at the implicit fixture because their experience tells them it is often not related to the test methods in a class.

Two interviewees did not see the value of detecting the dead fields smell for their system. P2 explains: “I am not bothered by the inherited fields of the super class. If two classes have the same functionality, we immediately move this in the super class so they can share it.” Also P3 does not see a problem with this design and says “I find the information on fields misleading. In our code base, inheritance is used as a convenient way of accessing helper methods.” Even though two participants are not concerned with their design decisions, systems which use inheritance instead of composition to allow code reuse are known to

be vulnerable to the fragile base class problem, which hinders maintainability (Mikhajlov and Sekerinski, 1998). The other three interviewees refer positively to the identification of the dead field smell.

5.6.4 RQ4: Does a fixture analysis technique help developers to understand and adjust fixture management strategies?

The results of the questionnaire show that developers expect a fixture management tool to be helpful during understanding and adjusting fixture management strategies, as illustrated in Figure 5.4. All participants agree that a fixture management tool could help improve the test code quality and 12 developers think the tool shows relevant information. In the interviews, all developers were positive, as demonstrated by P4: *“I really like the tool. I think it presents a lot of useful information. I think it can definitely be very beneficial for our company.”*

Not all developers are sure to look at the test fixture smell reports regularly. In the interviews, developers strongly felt that to allow adoption, the tool must be integrated with the regular build infrastructure. P1 says: *“It’s not enough to have the tool run. It should be part of the infrastructure and result in a failed build.”* Because time for quality improvement is limited, P3 suggests: *“We are not actively looking for opportunities to improve the code, but one gets the lucky one, when the threshold is exceeded.”* He then adds: *“If the tool calls our attention to these problems, we would schedule blocks of time to make the internal quality better.”* P5 likes that: *“I cannot get others to review my code all the time. So, a tool that tells me that things look odd, that’s good.”*

Regarding refactoring test code, P1 says: *“You want to assess how much risk is involved with a refactor. Sometimes you come to that point that you are less likely to change a test because it is smelly. But with the tool telling you that this test did not use anything, then you are more trusting to refactor test code.”* P1 mostly appreciates refactorings that can be done easily and quickly: *“It is all about low-hanging fruit: what can you do easily and quickly.”*

TestHound is designed not only to indicate smells in the test code, but also to guide the developer during refactoring by providing information on the test fixture usage profile, which is hard to obtain manually. This tool characteristic is also valued by the developers, as P1 says: *“I would look at the test methods and they might look unrelated, but without a lot of digging and work I might not know that they were not relying on anything from the class or the super class.”* And P3 especially liked the detail report: *“The summary report is good to get an idea why the build failed, but then I want to go and see these variables in the detail report.”* Also, P5 likes the refactoring suggestions: *“The tricky part is to actually understand why something is indicated as smelly. That’s the learning part. The refactoring suggestions help. That’s interesting to see.”*

We also asked participants to list the additional features they would like to see. P4 expresses that the different smells should be integrated in one high-level

metric: “*This would gives us an overall assessment, so that if you make some improvements you should see it in the metric.*”

Another positive outcome of the evaluation is that developers state the tool makes them think differently about their test code. P5 said: “*I found particularly interesting that the tool made me think about how the tests that I write might not be good tests. They do their job, but they may not be maintainable.*” P3 said: “*The report was definitely very useful. It triggered a lot of ideas to improve and discussions.*”

5.7 Discussion and Threats to Validity

In the following section, we discuss some of the key findings, observations and threats to validity.

Test Class to Class Convention. A simple way to start organizing test suites is to adopt the *Testcase Class per Class* pattern (Meszaros, 2007), a commonly used approach that is supported by IDEs like Eclipse which can generate test class templates for a given class. This is at odds with a different pattern, *Testcase Class per Fixture*, in which “we organize Test Methods into Testcase Classes based on commonality of the test fixture.” (Meszaros, 2007). Our empirical study shows that this pattern is not followed as often as it should, resulting in the smells and maintainability problems we detected. Based on this, we think it is necessary to rethink traditional mapping strategies and to develop further grouping recommendations and naming conventions, which take into account the evolution of a class as it starts to require more test fixtures.

Frequency of Vague Headers. One might argue that because vague headers occur frequently, they might not be a potential problem. During the interviews, we asked developers to explain the behavior of vague headers. Even though developers are familiar with the test framework and did place vague headers themselves, for several incidences they were uncertain or wrong about the concrete behavior.

Violation of the Single Responsibility Principle. Another observation we made is that the problem of not being able to have a non-smelly test class for a class can indicate a problem with the class under test, such as having too many responsibilities. Sometimes the solution can be not only to split the test class but also to split the class under test into several classes.

Inheritance Structure. In the test code we analyzed, we saw that some super classes are inherited by many test classes. This leads to dead inherited fields, because inherited setup functionality may not be needed. In all three systems, the dead fields are often the same ones (from certain super types), but repeatedly dead for many subclasses. While unused inherited fields are not a problem per se, the large superclass may become fragile (conform (Mikhajlov and Sekerinski, 1998)), making developers reluctant to adjust it.

Performance Improvement. Based on our case studies, we conclude that re-

factoring of test mavericks and general fixtures can lead to interesting performance improvements, especially considering that with continuous integration, test suites might run several times a day. In a future study, we want to gain a deep understanding of potential performance improvements associated with the application of the suggested refactorings of smelly test fixtures.

Threats to Validity. In terms of generalizability, in its current form, our implementation only works for Java-based systems that use JUnit or TestNG test frameworks. On the other hand, we think that this technique is not only easily transferable to other xUnit testing frameworks, but also to other languages. Further, our evaluation is limited to three software systems. We chose three systems that are quite different in nature (domain, open versus closed source), and assume that similar results will occur in other software systems. We chose the closed source case study because of the availability of software engineers to take part in the study and its closed source nature. The two open source systems were selected because they are well-known and used software systems. Further, we were familiar with the systems through earlier studies and thus could more easily test that the analysis was accurate. The developers we interviewed also felt this tool could be used to analyze other systems they had worked with.

With respect to internal validity, the analysis may be incomplete or have bugs. To conquer this threat, we implemented many test cases. The developers also indicated the results were consistent with their understanding of the system. Finally, the developers may have been positively biased towards the tool due to the nature of the experimental design. We tried to offset this somewhat by collecting the responses to the questionnaire anonymously.

Our method has some limitations when establishing dependency relationships between setup fields. This can lead to false-positive dead fields. To mitigate the risk of wrong results, we manually inspected all dead fields, and found only a few false-positive cases. For example, in eGit, 3% of the fields could not be mapped to a field usage. For future work, we will enhance the recognition of field usages, and we plan to assess the accuracy of the results in additional case studies. The metrics designed for smell detection are based on field and variable declarations. Actions performed on persistence data storages (such as databases or files) are only detected when a handle (i.e., object reference) is used for access.

5.8 Related Work

Earlier work introducing test smells has been discussed in Section 5.2. Scant research focuses on automatic *detection* of test smells. Among them, Van Rompaey et al. (2007) tried to detect the test smells *General fixture* and *Eager test* by means of metrics. In a subsequent paper, they describe a tool which used well-known software metrics to predict a broader variety of potential problems and test smells (Breugelmans and Van Rompaey, 2008). Our study differs in several aspects. First of all, we focus on test fixture management and analyze the test code

for specific fixture problems that are relevant in practice, and provide concrete refactoring suggestions. In contrast to our work, Borg and Kropp (2011) describe automated refactorings for acceptance tests based on the FIT framework. To the best of our knowledge, fixture-related test smells and refactoring have not been studied in detail so far.

In general, *code and design smells* have been researched in previous work. For example, Moha et al. (2010) outline a method called DECOR and its implementation to detect several code and design smells, and evaluate their technique in several case studies. Lanza and Marinescu use metrics to identify classes that might have design flaws (Lanza and Marinescu, 2006; Marinescu, 2001).

5.9 Concluding Remarks

The goal of this paper is to understand the nature of fixture-related problems in developer test suites. To that end, the contributions of the paper are 1) five *new* test fixture smells, 2) a technique to analyze test fixtures and automatically detect six test fixture smells, 3) an implementation of the technique in a tool called *TestHound*, 4) an investigation of three industrial-strength case studies that shows that test fixture smells occur in practice and 5) an evaluation with 13 developers that shows that the tool is helpful to understand, reflect on and adjust the test fixture.

In our future work, we plan to further research the evolution of test smells and investigate in depth how test class-to-class mappings influence the emergence of test fixture smells. Furthermore, we intend to apply *TestHound* to a range of further systems, broaden the scope of our fixture analysis, and assess performance implications of the proposed refactorings.

Acknowledgments: We would like to thank all interview and questionnaire participants for their time and commitment.

Strategies for Avoiding Text Fixture Smells During Software Evolution

Abstract

An important challenge in creating automated tests is how to design test fixtures, i.e., the setup code that initializes the system under test before actual automated testing can start.¹ Test designers have to choose between different approaches for the setup, trading off maintenance overhead with slow test execution. Over time, test code quality can erode and test smells can develop, such as the occurrence of overly general fixtures, obscure in-line code and dead fields. In this paper, we investigate how fixture-related test smells evolve over time by analyzing several thousand revisions of five open source systems. Our findings indicate that setup management strategies strongly influence the types of test fixture smells that emerge in code, and that several types of fixture smells often emerge at the same time. Based on this information, we recommend important guidelines for setup strategies, and suggest how tool support can be improved to help in both avoiding the emergence of such smells as well as how to refactor code when test smells do appear.

6.1 Introduction

Modern software development often includes the use of extensive automated test suites. While automated tests are helpful from a continuous integration and regression testing perspective, they lead to a substantial amount of test code (Zaid-

¹This chapter will appear in the 2013 10th Working Conference on Mining Software Repositories. The authors of this publication are Greiler, Zaidman, van Deursen and Storey.

man et al., 2011). This test code is often executed frequently, and needs to be maintained and understood. The long term success of automated testing is highly influenced by the maintainability of the test code (Meszaros, 2007). To support easier maintainability of a system, test methods should be structured clearly, well named, small in size, and code duplication across test methods should be avoided (Meszaros, 2007; Freeman and Pryce, 2009).

One important part of a test is the code that initializes the system under test (SUT), sets up all dependencies and puts the SUT in the right state to fulfill all preconditions needed to exercise the test. In line with Meszaros (2007), we refer to this part of a test as the *test fixture*. Developers have several options for structuring their test fixture code. The most straightforward option is to place the setup code directly in the test method, which we refer to as an *in-line setup*. An alternative approach is to place the setup code in helper methods that can be called by several test methods, the so-called *delegate setup* (Meszaros, 2007).

Today’s testing frameworks, such as the widely used xUnit family, have dedicated mechanisms for managing setup code invocations (Beck, 2002; Gamma and Beck, 2003). Therefore, helper-methods comprising the setup code can be marked (e.g., using annotations or naming conventions) as specific setup methods, and the test framework takes care of invoking them at a specific point in time. We refer to this as an *implicit setup* as the invocation happens implicitly.²

Developers must decide how to set up test fixtures and adjust their *fixture strategies* during the evolution of test code. Otherwise, they end up with poor solutions to recurring implementation and design problems in their test code, also known as *test smells* (van Deursen et al., 2001). To support developers during the analysis and adjustment of test fixtures, we previously developed a tool called *TestHound*³ to automatically detect test fixture smells and guide test code refactoring (Greiler et al., 2013a).

An evaluation of TestHound showed that developers are concerned about test fixture smells, and that TestHound helps them to discover and address those smells. But, we also learned that resolving these smells after they have been in the code for a long time can be problematic. Developers would benefit greatly from having the tool available in the continuous integration environment so that they are made aware of changes in the densities of test smells immediately.

In this paper, we investigate the evolution of test fixture smells and which software changes lead to increased test smell densities to determine the best time to alert developers about smell changes. We look at when and how test fixture smells are introduced, and what role the setup strategies play in smell evolution. Our contributions in this paper are:

1. a technique for analyzing multiple revisions of a software system for *fixture-related* test smells, and examining trends in smell evolution;

²For example, in the JUnit framework, methods can be named *setUp()* or marked with annotations such as *@Before* or *@BeforeClass*.

³<http://swierl.tudelft.nl/bin/view/MichaelaGreiler/TestHound>

2. an implementation of this technique in a tool called *TestEvoHound*, which mines Git and SVN repositories for test fixture smells;
3. insights in test fixture smell evolution in real-world situations based on an investigation of five well-known open source systems;
4. strategies for avoiding test fixture smells.

Our investigation shows that fixture management strategies strongly influence fixture smell evolution. Also, we find that fixture smells remain stable over long periods of time, until certain code changes cause drastic changes in smell densities. Making developers aware of these changes can help prevent the introduction of test smells with only small adjustments to the test code in a continuous and incremental fashion. Further, we show that classes with a larger number of test methods have more test fixture smells than classes with fewer test methods, and thus recommend that classes with many test methods be avoided or refactored.

Section 6.2 briefly summarizes different test fixture smells. Section 6.3 details the experimental design used to investigate the evolution trends of test fixture smells. Section 6.4 outlines the measurements implemented in the *TestEvoHound* tool. Section 6.5 details the results of our investigation, followed by a discussion of the findings in Section 6.6. In Section 6.7, we present related work and conclude in Section 6.8.

6.2 Test Smells

The *code smells* metaphor was first introduced by Fowler (1999), who describes a code smell as a poor solution to a recurring implementation and design problem. Code smells are not a problem per se, but they may lead to issues such as understanding difficulties, inefficient tests and poor maintainability of a software system. Later, van Deursen et al. (2001) introduced the term *test smells* by applying the smell metaphor to test code. Since then, their initial set of test smells has been extended (Meszaros, 2007; Van Rompaey et al., 2006; Neukirchen and Bisanz, 2007). In (Greiler et al., 2013a), we enhanced these test smells with additional *fixture-related* smells, derived metrics to aid in their detection, and implemented a technique to automatically detect test fixture smells in a tool called TestHound.

In this paper, we investigate the evolution of these test fixture smells, which are summarized below, in order to better understand how they can be avoided and how tool support would be most beneficial for developers. We refer the reader to (Greiler et al., 2013a) for more details on the *test fixture smells*.

General Fixture Smell. The general fixture smell occurs if test classes contain broad functionality in the *implicit* setup, and if several tests only access part of the fixture. Problems caused by a general fixture are two-fold. Firstly, the cause-effect relationship between fixture and the expected test outcome is less visible, and tests are harder to read and understand. This can lead to fragile tests: a change that should be unrelated affects tests because too much functionality is

covered in the fixture. Secondly, test performance can deteriorate and long test execution times may eventually cause developers to stop running tests altogether. We identify a test method as a general fixture method when it uses less than 70% of the fields initialized in the setup.

Test Maverick Smell. A test method is a maverick when the class providing the test method contains an *implicit* setup, but the test method is completely independent from the implicit setup procedure. The implicit setup is executed before the test method, but it is not needed. In such cases, understanding the cause-effect relationship between setup and test method can be hampered because discovering that test methods are unrelated from the implicit setup can be time consuming. We identify a test method as a maverick when it does not use any of the setup fields initialized.

Dead Fields Smell. The dead field smell occurs when a class or its super classes have fields that are never used by any of the test methods. Often, dead fields are inherited from a super class. This can indicate a suboptimal inheritance structure, or that the super class conflicts with the single responsibility principle (Martin, 2008). Also, dead fields within the test class itself can indicate incomplete or deprecated development activities. We identify dead fields as all fields that are initialized by the *implicit setup*, but never used by any of the test methods.

Lack of Cohesion of Test Methods Smell. Cohesion of a class indicates how strongly related and focused the various responsibilities of a class are (Chidamber and Kemerer, 1994). Low cohesive classes are smelly because they negatively affect code reuse, maintainability and comprehension (Fowler, 1999; Li and Henry, 1993). The smell Lack of Cohesion of Test Methods (LCOTM) occurs if test methods are grouped together in one test class without being cohesive. To measure LCOTM, we adjusted the Henderson-Seller Lack of Cohesion of Method metric (Henderson-Sellers, 1996). Differing from the original metric, we focus on the cohesion between test methods in a class.

Obscure In-Line Setup Smell. Meszaros (2007) introduced the smell obscure test to refer to a test that is difficult to understand, and thus, unsuitable for documentation purposes. From this smell we derived the obscure in-line setup smell. An *in-line* setup should consist of only the steps and variables essential to understanding the test; necessary but irrelevant steps should be encapsulated into helper methods. An obscure in-line setup covers too much setup functionality within the test method, and this can prevent one from seeing the test's relevant verification steps. We measure the obscurity of an in-line setup based on the number of local variables directly defined in a test method.

Vague Header Setup Smell. A vague header setup smell occurs when fields are solely initialized in the header of a class. We consider this a smell as the behavior of the code is not explicitly defined and depends on the field modifier (static or member) as well as on the implementation of the test framework. Vague header setups might hamper code comprehension and maintainability, as fields can be placed anywhere in the class. Further, in many test frameworks exception messages are more expressive for fields initialized in the setup. We report a vague

header smell when at least one field is solely initialized in the header of a class.

6.3 Experimental Setting

To understand test fixture smell evolution, we used case study research and investigated five research questions within five subject systems, as detailed below.

6.3.1 Research Questions

Our research questions focused on the evolution of test fixtures and test fixture smells.

RQ1: Do test fixture strategies change over time?

RQ2: Do test fixture smell densities increase over time?

RQ3: How are test fixture smells spread throughout a project?

RQ4: Which changes cause alterations in fixture smell trends?

RQ5: Are test fixture smells resolved?

We investigated different test fixture strategies and the changes made to test fixtures over time to get a general understanding of the characteristics of the systems under investigation (RQ1). To understand whether test fixture smell densities increase during the life of a project, we looked at test fixture smell trends (RQ2). We also looked at fixture smell dispersion to understand how smells spread throughout a software system (RQ3). We investigated what causes test fixture smells to change statistically and by manual investigation of severe changes (fluctuations) in smell trends (RQ4). To see whether smells are resolved by the developers (RQ5), we followed smelly classes throughout their evolution.

6.3.2 Case Studies

To investigate the evolution of test fixture smells, we selected five well-known, Java based, open source projects that have test suites and automatic build files. We display important characteristics of the latest analyzed revision of these projects (i.e., size, analysis duration, and number of classes comprising setup methods) in Table 6.1.

Checkstyle is a tool to help programmers write Java code that adheres to a coding standard. We analyzed 2252 Checkstyle revisions. The latest revision contained 549 test methods.

PMD is a tool to analyze Java source code for potential problems such as dead or duplicated code. We analyzed 1900 PMD revisions. The latest revision contained 739 test methods.

Table 6.1: *Fixture Management Strategies*

Project	KLOCs	# Test classes	# Test cases	Test No. setup	No. revisions	Period in life cycle of project	Date
Voldemort JUnit	130	132	520	71	2900	start → end	04/2011-10/2012
PMD	174	118	739	102	1900	mid → end	09/2007-11/2012
Checkstyle	66	156	549	131	2251	start → end	06/2001-10/2010
Jsoup	20	23	372	23	973	start → end	12/2009-09/2012
Java Azure SDK	39	30	358	14	300	start → end	10/2011-10/2012

Jsoup is a Java HTML parser that provides an API to extract and manipulate data using DOM, CSS and JQuery-like methods. We analyzed all 973 revisions of Jsoup, and the latest revision contained 372 test methods.

Azure Java SDK provides the Azure (i.e., Microsoft Cloud Platform) libraries for Java. We analyzed all 300 revisions of Azure. The latest revision contained 358 test methods.

Voldemort is a distributed key-value storage system. We analyzed all 2900 revisions of Voldemort. The latest revision contained 520 test methods.

6.4 Analysis of Fixture Smell Evolution

In this section, we introduce *TestEvoHound*, a tool we developed to automatically analyze the evolution of test fixtures and test fixture smells over multiple revisions of a software system. We also detail the measurements taken in order to answer our research questions.

6.4.1 TestEvoHound

We developed *TestEvoHound* to analyze fixture smell evolution. *TestEvoHound* works with Git and SVN repositories, and is available for download.⁴ When analyzing code, *TestEvoHound* executes four tasks.

During the *Revision Checkout* task, *TestEvoHound* checks out each revision of the project under analysis, and for each revision it starts the *Build Process* task. Here, the tool searches for ANT or MAVEN build files, initiates the build process and compiles the source code (including tests). Then, the *Test Fixture Smell Analysis* task invokes the TestHound tool to analyze the current revision for smells and stores the outcome. Finally, when all revisions have been analyzed, the *Trend Analysis* occurs. Here, the *TestEvoHound* tool calculates the trends and measurements among all revisions, as described in the next subsection. This information is stored in comma-separated value format to allow easy visualization by tools like Excel or R.⁵

⁴<http://swierl.tudelft.nl/bin/view/MichaelaGreiler/TestEvoHound>

⁵<http://www.r-project.org/>

6.4.2 Measurements to Answer the Research Questions

RQ1. To see whether test fixture management changes over time, we looked at the presence and absence of the *implicit* setup mechanisms provided by the test frameworks. This means that, for each project and over all revisions, we analyzed how many of all test classes contained an *implicit* setup and whether this changed over time. We also analyzed how many fields were declared in a test class, as fields are also a way to create a state accessible by all test methods of a test class.

RQ2. To answer whether test fixture smell density increases over time, we measured the occurrence of the six different fixture smells for each revision, and charted whether the ratio between smelly entities to all entities changed. We looked at this ratio because all of the systems under study increased in size over time. Thus, the number of smelly entities has to be considered in relation to the overall number of entities. This ratio demonstrates whether the quantity of smells is rising or falling. In the remainder of this paper, we refer to a series of ratios as the trend of smell evolution.

For the general fixture, the test maverick, and the obscure in-line setup smells, we compared the number of test methods affected by these smells with the overall number of test methods in the code base. We calculated the trend in dead field evolution by comparing the number of dead fields with the overall number of fields. For the LCOTM and vague header smells, we compared the number of test classes affected with the overall number of test classes.

RQ3. To better understand whether test fixture smells are widely spread among all test classes, or whether certain test classes are more prone to fixture smells, we investigated the dispersion of fixture smells. To do this, we looked at histograms of the test fixture smells.

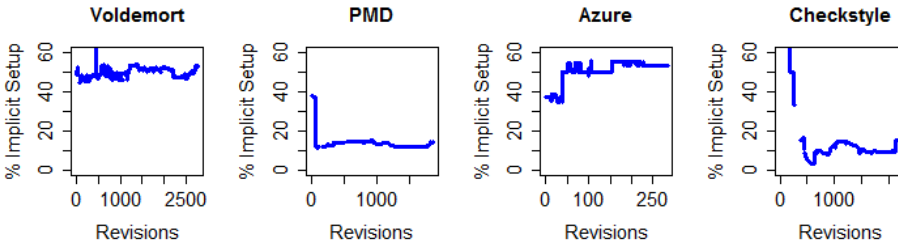
RQ4. To understand which software changes cause test fixture smell densities to change, we statistically tested whether the number of test methods or fields within a class correlates with the smell density of a class. Then, we manually investigated the code base and the commit logs for periods where severe fluctuations in the trends of test fixture smells (increase or decrease) occurred.

For the statistical tests, we used Spearman correlations provided by the statistical program *R* (package *hmisc*) to investigate the following two hypotheses:

H1 The more test methods placed within a test class, the higher the smell density of a test class is.

H2 The more fields placed within a test class, the higher the smell density of a test class is.

RQ5. By tracking smelly classes over time and investigating decreases in test fixture smells, we see whether and how test fixture smells are resolved.

Figure 6.1: *Implicit Setup Trends*

6.5 Investigation of Test Fixture Smell Evolution

In this section, we detail our findings from the investigation of the five research questions. Each subsection covers one of the research questions, with the exception of RQ4, to which we dedicate two subsections (Sections 6.5.4 and 6.5.5).

6.5.1 Evolution of Test Fixtures

Our analysis shows that the test fixture strategies used across the projects differ greatly. In Jsoup, the developers completely refrained from using the *implicit* setup mechanism available in JUnit. In Checkstyle and PMD, approximately 10% and 13% (respectively) of the test classes comprise an *implicit* setup. In Voldemort and Azure, approximately 50% of the classes have an *implicit* setup. Just as the setup management styles differ across all five projects, the fixture smells also differ, as illustrated in Fig. 6.3.

Setup Trends. To answer whether the setup strategies change over time (RQ1), we looked at the trends for the presence of implicit setups. As can be seen in Fig. 6.1, the usage pattern for the implicit setup changed. For example, in Azure, the density of setups per class increased over time, and in PMD, it decreased. On the other hand, apart from some severe fluctuations in density, the ratio stays quite stable for all four projects. We excluded Jsoup because we did not encounter a single implicit setup.

Setup Fluctuations. To understand the severe setup fluctuations, we manually investigated the changes between revisions that caused the fluctuations. In Checkstyle, the drop in *implicit* setup usage at the beginning is because the code base started with three test cases, each containing an *implicit* setup (i.e., 100%). Over time, tests without an *implicit* setup were added, causing the percentage to drop to 10%.

In PMD, the percentage of test classes that have an *implicit* setup drops at the beginning. This is due to a simple structural change where 38 test classes were removed by changing how methods were invoked. Before the change, many of these test classes contained only one test method that invoked another method.

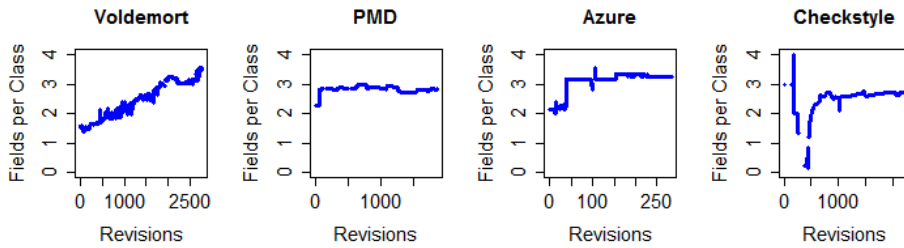


Figure 6.2: *Field Trends*

After the change, this functionality is covered by the parent test class within the setup, making 38 test classes obsolete. The commit log states: “*code refactoring: testAll() moved to parent, rules are now added in setUp() using addRule()*”, confirming our observation.

In Azure, the increase of implicit setups is due to the addition of six test classes, all of which have an implicit setup. The commit log states: “*table tests have been added*”.

Fields. As the fields of a class are a way to create a state accessible by all methods of a test class, we also analyzed how many fields are declared within each test class over time. In Checkstyle and PMD, even though only a small percentage of the classes contain a setup, they still have a similar number of class fields as compared with Azure and Voldemort (2.5 to 3 fields per test class). The only exception is Jsoup, which has only two class fields within all test classes. The trend of fields per test class, as illustrated in Fig. 6.2, reveals that in Voldemort, the number of fields per test class continually increases, which is an indication that the test classes become more complex. In the other three systems, there are distinct events where the number of fields fluctuates and then stays quite stable

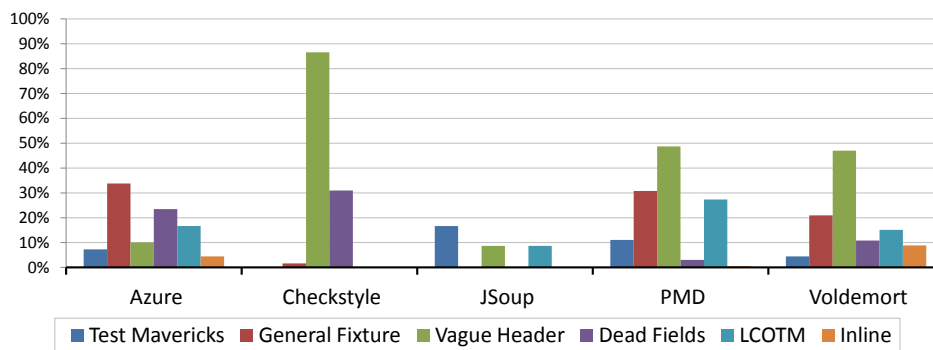


Figure 6.3: *Test Smell Density Among Projects*

over time. For PMD, the 38 deleted test classes were quite simple and did not comprise fields, so their deletion led to an increased ratio of fields per test class. In Azure, the added test classes (“table tests”) also led to an increase in fields per class, which is indicative of an increase in complexity.

6.5.2 Discovery of Test Fixture Smell Trends

Lehman’s law of increasing complexity states that when a system evolves, its complexity increases unless work is done to maintain or reduce it (Lehman, 1984). Therefore, we expect that due to the increasing complexity of a given system – the test fixture grows, and more test methods are placed within the same test class – the potential for test fixture smells (such as the general fixture, LCOTM, test mavericks and dead fields) increases.

Over the investigated periods of all five projects, the number of test classes and test methods increased. On the other hand, the ratio of test methods per test class stayed quite stable for PMD and Voldermort. In Azure and Checkstyle, the number of tests per class increased slightly over time, and in JSoup we observed the strongest increase. In Checkstyle, a peak occurred at the beginning because the test efforts started within a single class that comprised up to 24 test methods, and new test classes were introduced slowly. Even though the mean value for tests per class seems small and stable across all systems, the inspection of the dispersion of tests per class (see Fig. 6.4) shows, some test classes contain a significantly larger number of test methods (30, 40 or even 80) than other test classes. Over time, this imbalance grows.

Interestingly, the experiments revealed that even though test code becomes more complex (more tests and more fields per test class), a general growth of test fixture smells over time does not occur. As depicted in Fig. 6.5, the test fixture smell trends often stay stable over time, and fluctuate greatly only at a distinct point in time. Continual increases or decreases are exceptional, as in the case of Voldemort. In the following sections, we take a closer look at how smells dispersed in a system, and what causes the fluctuations in the smell trends.

6.5.3 Dispersion of Test Fixture Smells

Over the course of several revisions, we investigated how frequently smelly entities occur per test class. We used histograms to visualize whether all classes contain the same number of fixture smells or whether some classes are more prone to test smells.

For general fixtures, test mavericks and obscure in-line setups, smells cluster in a few classes. To illustrate the outcome, this paper shows the dispersion of the general fixture smell for the first, middle and last analyzable revisions of each project in Fig. 6.6. For all projects, some classes contribute a disproportionately large number of general fixture methods, and over time, these classes accumulate more general fixtures. In the last revision of *PMD*, three test classes contributed

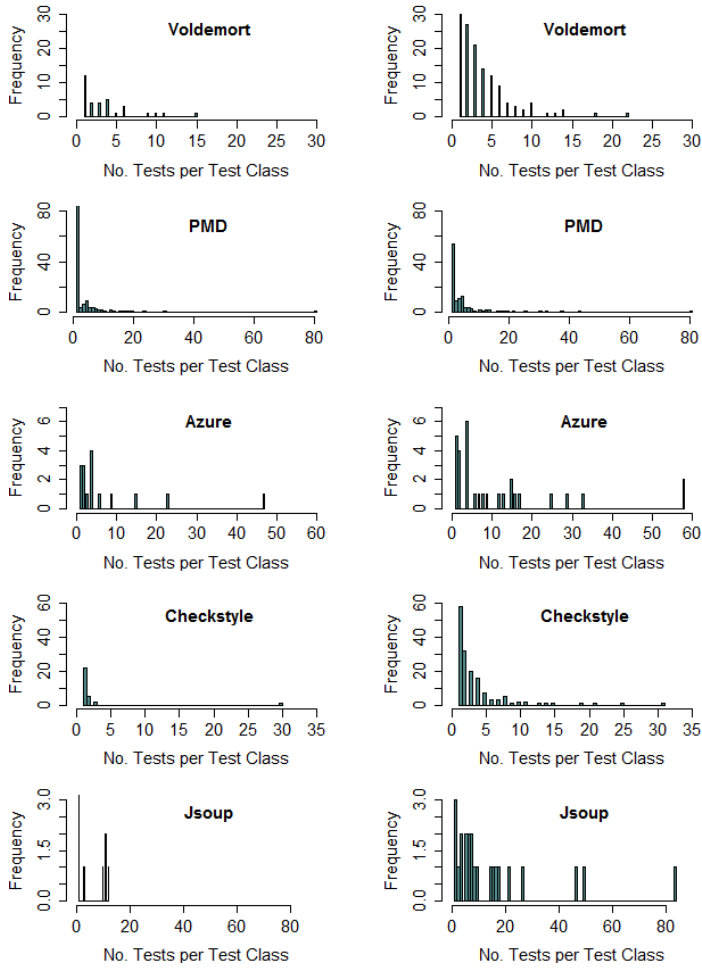


Figure 6.4: *Histograms of Tests per Class for first and last Analyzable Revisions*

45% of all general fixture methods (i.e., 2.5% of the classes), and 5.1% of the test classes contributed 60.1% of the general fixtures. In the first analyzed revision, three classes contributed 40% of the smells. A similar effect can also be observed for Azure, Checkstyle and Voldemort for the general fixture and test maverick smells. Jsoup does not contain general fixture methods, but it has a strong tendency for test mavericks to cluster. We can also observe that over time, the relative number of test smells per class increases (i.e., the ratio of smelly entities to entities per class).

On the other hand, dead fields did not cluster. As vague headers or a LCOTM are either present or absent in a class, their distribution is not of interest.

Summary. Since some classes are more prone to test smells, such as general

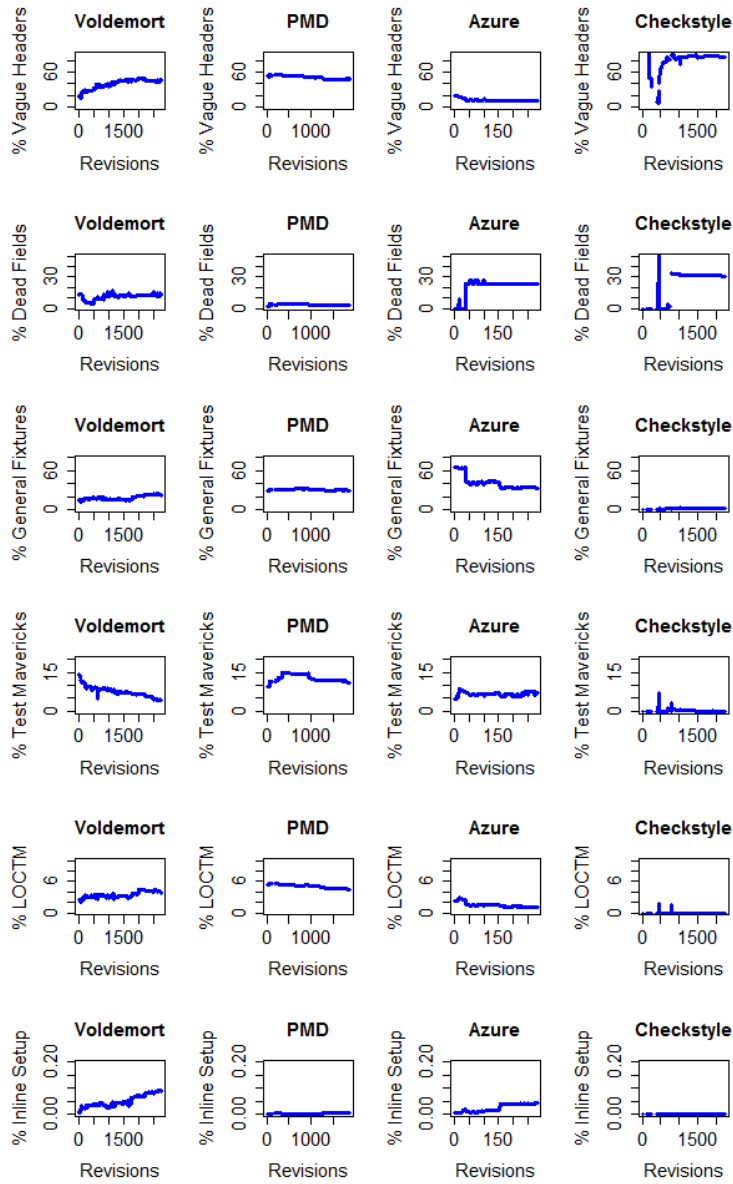


Figure 6.5: All Smell Fixture Trends

fixture, test maverick and obscure in-line setup, refactoring activities can be directed to these classes. This shows that one can reduce the majority of test fixture smells by cleaning a few smelly classes.

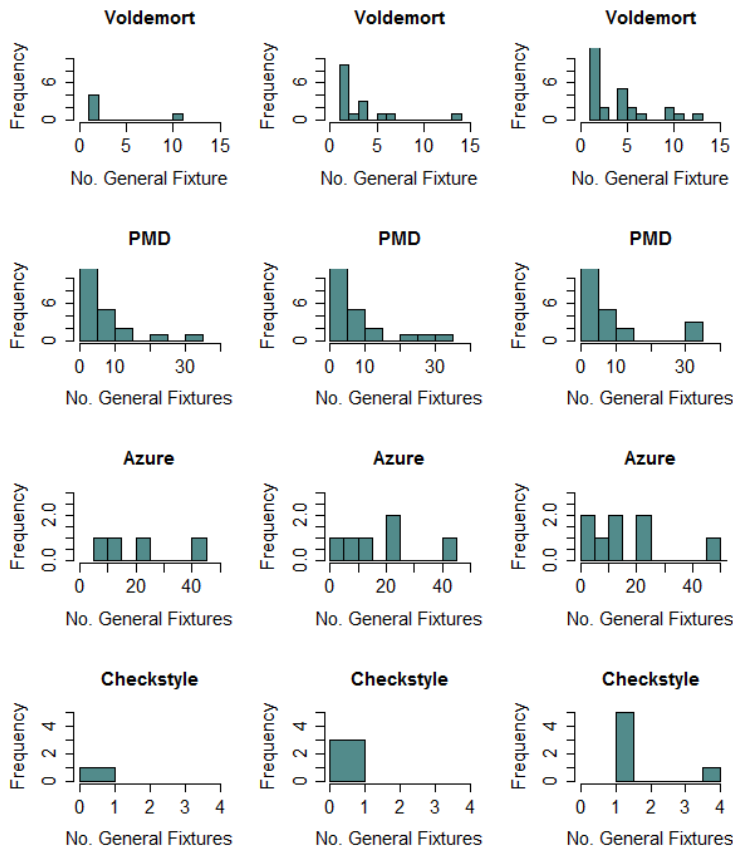


Figure 6.6: Dispersion of General Fixtures for first, middle and last Analyzable Revisions

6.5.4 Development of Test Fixture Smells

The smell trends do not show a general continual increase of smell densities over time as expected, but considering the smell distribution, there is a tendency for some classes to accumulate more smells (absolute and relative) over time than others. We also see that some classes comprise an unproportionally large number of test methods. The more test methods contained within a single class, the more diverse the requirements for the test fixtures can become. Because of this, we expect that test classes with a higher number of test methods have a higher smell density than their smaller counterparts. Also, we expect that test classes with more fields have a higher smell density than test classes with fewer fields.

To investigate these hypotheses, we used statistics and correlated the number of test methods, or fields per class, with the percentage of test methods affected by a smell. We excluded Jsoup from this experiment as the project has few smells

and few fields. The results are given in Tables 6.2 and 6.3. We used the thresholds defined by Hopkins (2000), where a correlation is considered moderate when the value is higher than 0.3 (or lower than -0.3), and strong when the value is higher than 0.5 (or lower than -0.5). For a correlation to be significant, the p-value of the test needs to be below 0.01. This means the chance that the correlation is due to random chance is less than 1 percent.

Data Points. We did not use each data point (i.e., all classes for each revision) because the two variables correlated (i.e., test methods or fields, and a particular smell factor) might be stable over time. This can cause certain combinations (observations) to seem more likely than others. Therefore, we reduced the dataset to include only unique combinations. For example, in case test class “TestA” has three test methods and one general fixture method over a period of 10 revisions, we include this incident only once in our data set, whereby we look for uniqueness considering more than 21 different characteristics of a class.⁶ Further, as a class without a field can not have a general fixture, LCOTM, test maverick, dead field or vague header smell, we excluded classes without a field. For LCOTM, we also only considered data points with a valid LCOTM value (i.e., LCOTM is undefined when a class has a single method). Therefore, the results for LCOTM detail the number of data points separately.

Test Methods. For three of the four projects, the results indicate that for test smells such as general fixture, LCOTM, test maverick and obscure in-line setup, *a higher number of test methods per class* correlates with a higher density of these smells. For Checkstyle, the correlation cannot be shown as very few test smells exist. We also investigated whether more dead fields exist in classes with more test methods. This correlation does not hold for Checkstyle and PMD. For Azure and Voldemort, the results show a weak negative correlation, indicating that with a higher number of test methods per class, the dead fields density decreases.

Fields. The second characteristic we expect to find related to the smell density of a class is the number of fields per class.

We observe a correlation between *smell density and the number of fields* for the general fixture and LCOTM smells within the PMD and Azure systems, as listed in Tables 6.2 and 6.3. A weak correlation also exists for a higher obscure in-line setup density and fields in PMD and Voldemort. Azure is the only other system which shows a correlation between a higher test maverick density and fields, and a weak negative correlation for dead fields (i.e., more fields are correlated to a lower dead field density).

We investigated why an increased number of fields also increases fixture smell density for the PMD and Azure systems, and saw that this has to do with the style of fixture management. For the majority of new test methods, developers introduce a new dedicated field, and often there is one field for each test method. This means that if new test methods are added to a class, it is very likely that

⁶Detailed information on the attributes covered for each class is outlined at <http://swierl.tudelft.nl/bin/view/MichaelaGreiler/TestEvoHound>

Project	N	Tests per Class							
		GF		TM		OI		DF	
		Cor	p	Cor	p	Cor	p	Cor	p
Voldemort	705	0.63	0	0.34	0	0.06	0.09	-0.41	0
PMD	268	0.51	0	0.56	0	0.37	0	0.05	0.40
Checkstyle	437	0.11	0.03	-0.09	0.06	Na	Na	-0.14	<0.001
Azure	118	0.51	0	0.40	0	0.48	0	-0.46	0

Project	N	Fields per Class							
		GF		TM		OI		DF	
		Cor	p	Cor	p	Cor	p	Cor	p
Voldemort	705	0.23	0	-0.15	0	0.32	0	0.24	0
PMD	268	0.72	0	0.23	<0.001	0.38	0	0.03	0.67
Checkstyle	437	-0.06	0.24	-0.04	0.43	Na	Na	0.22	0
Azure	118	0.58	0	0.64	0	0.25	0.005	-0.40	0

Table 6.2: Statistical correlations between 1) no. of tests per class and smell increase and 2) no. of fields per class and smell increase

Tests per Class			
Project	N	LCOTM	
		Cor	p
Voldemort	687	0.60	0
PMD	263	0.67	0
Checkstyle	430	0.11	0.02
Azure	118	0.48	0

Fields per Class			
Project	N	LCOTM	
		Cor	p
Voldemort	687	0.14	<0.001
PMD	263	0.55	0
Checkstyle	430	-0.02	0.74
Azure	118	0.57	0

Table 6.3: Statistical correlations between 1) no. of tests per class and increase in LCOTM and 2) no. of fields per class and increase in LCOTM

all of them are general fixture methods, and the lack of cohesion between the test methods increases further. With some test classes in PMD, either almost all test methods are general fixtures, or they do not have anything to do with the fields of the class, making them test mavericks. For example, in the “JDKVersionTest”, all 33 test methods are general fixture methods, whereby each method uses very few of the 37 fields declared. The LCOTM value in this class is as high as 0.99 (with 1 representing a completely noncohesive class).

6.5.5 Fluctuations in Test Fixture Smells

As is apparent from Fig. 6.5, fixture smell trends tend to change drastically at certain points in time. We manually investigated code base changes that occurred during the periods of severe smell trend changes to understand “Which changes cause alterations in fixture smell trends” (RQ4).

Vague Header Smell. A drastic change in the vague header smell is seen in Checkstyle alone. Checkstyle started out with a single test class that contains a vague header (i.e., 100%). In the next revisions, several tests without vague headers were added, until revision 433, where the smell continually increases due to changes in the inheritance structure and refactoring activities.

Dead Field Smell. For the dead field smell, we analyzed the drastic changes in Azure and Checkstyle. The main test fixture smell in *Checkstyle* is the *dead field* smell, which drastically increases around revision 760-769 (as illustrated by Fig. 6.7). The investigation of the changes between these revisions revealed that all additional dead fields are inherited from a single base class that is extended by almost all test classes (BaseCheckTestCase). In this test class, a helper method was introduced, and one field (which was no longer needed) was forgotten and remained in the class as legacy. In the commit log, the developer notes: “Added a helper method to create a configuration for a check...”. This dead field remained for the duration of our investigation. With *Azure*, a strong increase at revision

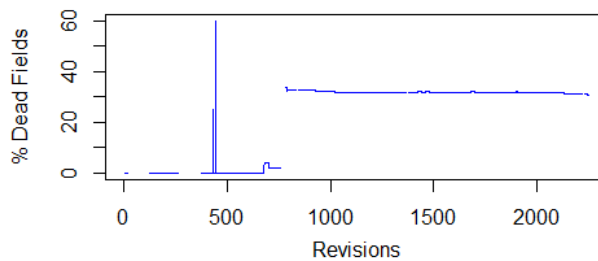


Figure 6.7: *Dead Field Density in Checkstyle*

39 is also due to dead fields inherited from one super class, which is extended by six test classes.

General Fixture Smell. The main fluctuation in general fixture smell trends can be seen in *Azure*. For *Azure*, this is also the predominant smell. The trend, displayed in Fig. 6.8, shows two distinct decreases: at revision 40 (from over 60% to less than 42%) and at revision 150. In revision 40, a drastic decrease occurred because new test classes providing 87 tests (testing the Table client) were added. The commit log shows “*Table Client commit [...]*”. With all these added tests, only two new general fixture methods were introduced. Also at revision 150, 3 new test classes and 76 tests were added, most of them without general fixture methods. In both cases, none of the already existing general fixtures have been

resolved.

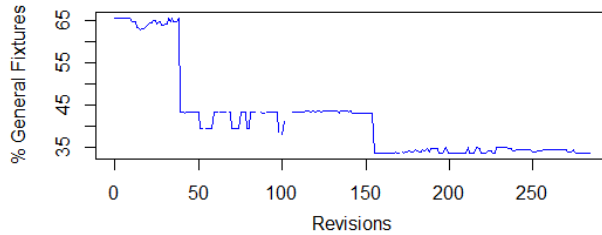


Figure 6.8: *General Fixture Trend Azure*

Test Maverick Smell. There are stronger fluctuations in the test maverick density in PMD and Jsoup, as well as a drastic continual decrease of the test maverick density in Voldemort.

In *PMD*, 12-14% of the methods are test mavericks, and the smell density increases and then decreases again, as illustrated in Fig. 6.9. The first increases are due to the aforementioned removal of several test classes and test methods, and the introduction of vague headers in classes. The manual investigation of the strong decrease around revision 900 reveals that the Java version was changed during this period. The commit log contains the following comment: “*Remove support for Java 1.4 runtime. [...] changes are made to code which made checks for running on a 1.4 JVM (found via searches).*” This change causes a drop in the number of test mavericks as the *implicit setup* of two test classes, used only to set the Java version, was removed.

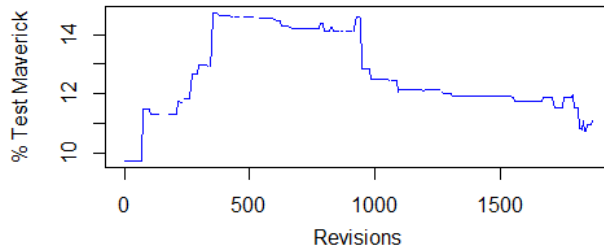


Figure 6.9: *Test Maverick Trend in PMD*

In Jsoup, test mavericks are caused by two classes that contain a vague header setup.⁷ The fluctuations (illustrated in Fig. 6.10) occurred when the two classes were introduced to the code base (revisions 20 and 80). Over time, more test methods were added to these classes, which make no use of the fields of the classes. Depending on whether more non-smelly tests are added, or more tests

⁷ElementTest and UrlConnectTest

are added to these two classes, the smell density increases or decreases. In the latest revision, these two classes cause over 60 test maverick methods.

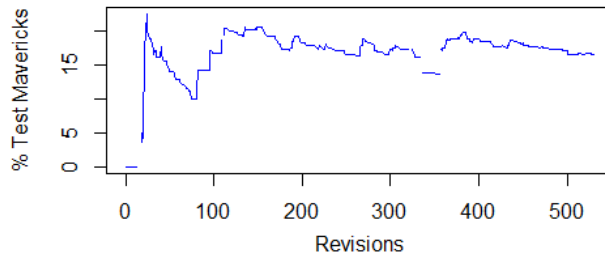


Figure 6.10: *Test Maverick Smell Trend in Jsoup*

The test maverick density in *Voldemort* steadily decreases. This is because fewer test mavericks are introduced over time, whereby the number of test methods increases (as illustrated in Fig. 6.11). The test maverick trend increases visible in the graph are due to the introduction of a field (via a vague header) which is only used by a few of the methods, and to a new test class which has a setup used by only a few test methods. The decreases are because new fields are added to classes with *implicit setups*, and previous test mavericks now use the fields.

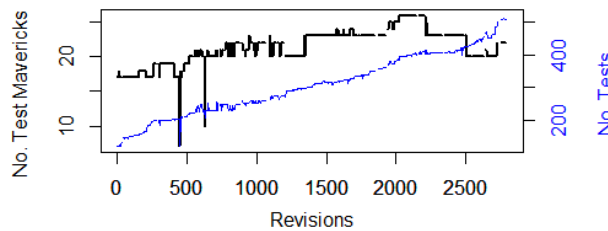


Figure 6.11: *Test Maverick Smell in Voldemort*

LCOTM Smell. The LCOTM smell trends do not increase over time. In PMD, the relative number of test classes suffering from the LCOTM smell even decreases. On the other hand, when inspecting the smelly classes, we observe that their LCOTM value does not decrease, but it either stays stable or increases. In Azure, the more severe fluctuations in LCOTM trends are because of the drastic increase of non-smelly classes.

Obscure In-line Setup Smell. In Azure, the number of obscure in-line methods increases (around revision 150). The majority of smelly methods come from two new test classes (“CloudQueueTests” and “CloudBlobContainerTests”). The other obscure in-line setups were added to an already existing class (“TableClientTests”). In Voldemort, there is a continuous increase in the obscure in-line setup smell, where a few classes, such as “RoutedStoreTest” and “AbstractRe-

balanceTest”, accumulate more smells over time. Interestingly, in Jsoup, even though the test setup was placed within test methods (or helper methods), not a single test method suffers from the obscure in-line setup smell.

6.5.6 Test Fixture Smell Resolution

Previous research on test smells suggests that test smells do not get resolved unless the test class or methods are deleted (Peters and Zaidman, 2012). We investigated whether this phenomenon can also be observed in our subject systems and for *test fixture smells*.

In general, we did not observe a major reduction of test smells. Several times, test smells decreased because new non-smelly test methods were added, thus reducing the percentage of smelly methods, as was seen with Azure. Other times, changes to the code base caused the smells to be reduced. For example, changing the Java version resolved many of PMD’s test mavericks. In the majority of cases, we saw that fixture smells are either not resolved, or are only resolved by deletion. Here, we will summarize the exceptional cases of smell resolution without deletion.

In Checkstyle, during a short period (revisions 442-444; also visible in Fig. 6.7), an overly general inheritance structure causes several dead inherited fields and test mavericks. The log states: *“Refactored the tests to only use the Checker interface”* and *“Refactoring the way the Checker is configured. Not happy with the current approach - it was hack”*. This change addresses the problem of the dead fields and the test mavericks, and shows that the developers made an effort to resolve the smells by changing the inheritance structure and using the functionality of the super class. Another time, test mavericks appear for a short time while the helper method in “BaseCheckTestCase” is implemented.

In Voldemort, a large number of test mavericks are resolved over time (visible in the continual decrease in the test mavericks trend) because new fields, which are then used by test methods, were added to existing classes (also visible in Fig. 6.2). For example, this occurs in test “ConsistentRoutingStrategyTest”.

In PMD, we see that some of the obscure in-line setups are resolved. For example, in the test class “RuleSetFactoryTest”, an *“Extract Method”* refactoring (Fowler, 1999) was performed, which resolved two in-line smells. In Voldemort, in-line setups are also resolved from time to time. On the other hand, in Azure, none of the obscure in-line methods are resolved.

General fixture, LCOTM and vague header smells are seldomly resolved in all systems. Exceptions are cases where the final functionality of a test is not yet implemented. For example, in Voldemort, developers added a test method that comprised only a few statements and a “todo” comment. After some time, the method was fully implemented and the general fixture resolved.

6.6 Discussion

6.6.1 Findings

Our study revealed many interesting findings about fixture management strategies and their influence on test smells, as well as how fixture strategies and smells evolve during the lifetime of several open source projects. In summary, our investigation of test fixture smells showed that:

- the style of fixture management varies greatly between projects;
- the projects' test suites also suffer from different patterns of test fixture smells;
- the number of test methods per test class correlates with the density of test fixture smells;
- more fields in a class do not necessarily correlate with a higher test fixture smell density;
- drastic increases in test smells are often caused by structural changes (such as refactorings with forgotten legacy functionality);
- once introduced, test smells tend to stick around and do not get resolved;
- most likely, fixture smells disappear because the test class or test method is deleted.

6.6.2 Implications for Automated Test Fixture Smell Detection

As we saw from a previous study (Greiler et al., 2013a), developers are concerned with the quality of their test code and see test fixture smells as potential problems. On the other hand, they are under pressure to develop production code, not to improve test code. The developers indicated that in terms of improving test code, they look for “low hanging fruit”, i.e., easy changes that do not involve major refactoring. The developers also indicated that they want to be notified when smells are introduced. We believe that the findings of this study can be used to design smell alerting mechanisms for developers evolving test code.

We saw that the test fixture management strategy greatly varies between projects, and that projects stay with the style they started with. For project leads, this means that they have to make conscious decisions at the beginning, and know which test fixture smells are more likely to occur as a consequence of the chosen fixture management strategy.

Also, even though some classes become smellier over time, we saw that test smell trends do not continually increase. This suggests that developers eventually learn the right strategies for writing test cases for a given system, and that new classes tend to be less smelly than old ones, which is in line with the information developers gave in the interviews (Greiler et al., 2013a).

Further, we also saw that simple structural changes in test classes can lead to both a drastic increase as well as a decrease in test fixture smells. First, because small changes can have a major impact on smell development, we believe getting

developers' attention when they are about to introduce test smells is an important first step to smell avoidance. Second, as test smells tend to accumulate in a few classes, refactoring these classes can greatly reduce test fixture smells.

Further, as the number of test methods per class impacts the smelliness of test code, developers should reconsider the current practice of grouping test methods within a class. Alerting developers about the lack of cohesion of test methods might be a first step in reducing class size.

6.6.3 Strategies and Recommendations

Based on the observations and findings of this study, we defined the following strategies and recommendations for smell avoidance:

- Keep test classes small (and therefore coherent), by reducing the number of test methods within a class.
- Keep inheritance structures flat, and also limit the scope of super classes (e.g., do not implement one super test class that provides functionality for all base test classes).
- Use composition instead of inheritance to provide test classes with helper functionality.
- Create data classes that provide test inputs to avoid overloading test classes with field declarations that are only used for a single test method. This improves performance and understandability of test classes.
- Be aware that declaring fields in the header may impact tests in terms of understandability and test performance.
- Reconsider the “one test class per class” organization in case test methods within a single test class require diverse states and configurations for the system under test.
- Carefully consider the consequences of the chosen fixture strategies and the impact those decisions will have on the projects, as this is not likely to change.

6.6.4 Threats to Validity

In terms of generalizability, our current implementation only works for Java-based systems that use JUnit or TestNG test frameworks and have automatic build files (i.e., Maven or ANT) available. On the other hand, it should be easy to adjust our tool's build process so that developers using other means to build systems can analyze their history. Further, we believe that this technique is not only easily transferable to other xUnit testing frameworks, but also to other languages. Also, our evaluation was limited to five software systems. However, we chose systems that are well known, publicly accessible and actively developed.

With respect to internal validity, the analysis may be incomplete or contain bugs. To conquer this threat, we implemented multiple test cases. Also, the extensive manual inspection of the systems under study confirmed our confidence

in the correctness and precision of the analysis. For future work, we plan to assess the accuracy of the results in additional case studies.

We also had to make some changes to the systems under study. For *Checkstyle*, a wide range of revisions had a non compilable test class checked in⁸. To be able to compile and analyze the rest of the tests, we deleted this class in case it failed to compile. For several revisions in *PMD*, the Maven build file did not link to the correct dependencies. So, we updated the dependency information in order to be able to analyze a wide range of revisions.

6.7 Related Work

As discussed in Section 6.2, test smells have been studied previously, with some research efforts focusing on the automatic detection of test smells. Among them, Van Rompaey et al. (2007) tried to detect *General fixture* and *Eager test* test smells by means of metrics. Subsequently, they described a tool which used well-known software metrics to predict a broader variety of potential problems and test smells (Breugelmans and Van Rompaey, 2008). Our study of test smells differs in several ways. First, we focus on test fixture management and analyze test code for specific fixture problems relevant in practice. We also provide concrete refactoring suggestions for developers. In contrast to our work, Borg and Kropp (2011) described automated refactoring for acceptance tests based on the FIT framework. To the best of our knowledge, fixture-related test smells and refactoring have not been studied so far.

Co-evolution of software and test code has been investigated by Zaidman et al. (2011). Pinto et al. (2012) investigated the evolution of test code in order to better understand how test repair tooling can assist developers during test maintenance.

Galli et al. (2004a) presented a taxonomy of (Smalltalk) unit tests in which they distinguish tests based on, for example, the number of test methods per method under test, and whether or not exceptions are taken into account.

In general, *code and design smells* have been researched in previous work. For example, Moha et al. (2010) outline a method called DECOR and its implementation to detect several code and design smells and evaluate their technique in several case studies. Lanza and Marinescu use metrics to identify classes that might have design flaws (Lanza and Marinescu, 2006; Marinescu, 2001). The metrics and smells presented in this paper address properties of code exclusively present in test code, i.e., the creation and tear down of the test fixtures.

6.8 Conclusion

In this paper, we investigated the evolution of test fixture smells so that we could discover the most beneficial time to alert developers about smell changes,

⁸LocalizedMessageTest.java.

and to learn which software changes lead to test smell increases. Therefore, we investigated when and how test fixture smells are introduced, and which roles the setup strategies play. Our findings indicate that test fixture smells do not continually increase over time, even though system complexity increases. There is a correlation between the number of tests within a class and smell density. An important insight is the clustering effect of test smells; the few classes that contribute the majority of test smells are the classes developers should be made aware of.

Our contributions in this paper are:

1. an implementation of a tool which supports the mining of repositories;
2. a technique (and implementation) to analyze several revisions of a software system for fixture-related test smells, and to understand the trends in smell evolution;
3. an investigation of test fixture smell evolution in five well-known open source systems;
4. strategies and guidance on how to avoid test fixture smells.

In future work, we plan to integrate the TestEvoHound tool in the continuous integration environment in order to give immediate feedback to developers based on the findings of this paper.

Conclusion

7.1 Contributions

In this dissertation, we investigated the testing practices used to test plug-in-based systems, revealed several challenges during integration testing, and developed and evaluated three techniques to assist developers during test suite comprehension and maintenance. In particular, the following contributions have been made:

- In Chapter 2, we presented an in-depth and systematic investigation of the testing practices and challenges within a plug-in community (i.e., Eclipse);
- we revealed how plug-in-based systems are tested in practice;
- we identified several barriers hindering the adoption of automated integration and system testing; and
- we highlighted the role of the community, which serves as compensation strategy for limited testing activities.
- In Chapter 3, we distilled several information needs developers have during test suite comprehension for plug-in systems;
- we developed a static and dynamic analysis technique that provides the developer with an abstraction of the system under test and its test suites, by recovering five architectural views which highlight the integration with other plug-ins and how this integration is tested;
- we provided an implementation of that technique in a tool called *Eclipse Test Suite Exploration* (ETSE) tool; and
- we conducted an evaluation of this technique based on case study research, showing its applicability, scalability and precision.

- In Chapter 4, we presented a dynamic analysis technique that automatically derives similarity-based relations between test cases, to support developers during test suite maintenance and comprehension tasks;
- we performed an assessment of the usefulness of the similarity-based relations based on two case studies; and
- we provided an implementation of this technique in a framework for mining test connections called *Test Similarity Correlator*.
- In Chapter 5, we presented five new test smells related to test fixtures;
- we developed a static analysis technique to automatically detect six test fixture smells;
- we provided an implementation of the technique in a tool called *TestHound*;
- based on three industrial-strength case studies, we showed that test fixture smells occur in practice; and
- provided an evaluation involving 13 developers that showed that the technique is helpful to understand, reflect, and adjust the test fixture.
- In Chapter 6, we presented an investigation of the evolution of test fixture smells based on five case studies;
- we showed that test fixture smells often emerge at the same time and cluster in few test classes; and
- based on these findings, we derived several strategies to avoid test fixture smells.

7.2 Research Questions Revisited

In this dissertation, we investigated four overarching research questions, whereby each question was addressed by two different chapters. Let us revisit each of the four overarching research questions of this thesis here.

What makes testing of modular and dynamic systems challenging?

Modular and dynamic software systems, such as plug-in systems, can be complex compositions, integrating multiple plug-ins from different developers into one product, and thus, raising concerns about the compatibility of their components (Pohl and Metzger, 2006; Rehmand et al., 2007; Weyuker, 1998). Incompatibility, be it because of combinations of plug-ins or versions, can be hard to strive against, and may restrict the benefits plug-in systems offer. Within two separate studies we were able to identify several challenges developers face when testing plug-in systems. Summarized, those challenges are:

Plug-in systems are conglomerates of several different plug-ins, with different owners. This makes the *responsibility* for integration or system testing is less clear, and the lack of *ownership* of the overall system leads to a test maintenance overhead. In plug-in systems, functionality can change after deployment, which makes testing a challenge as *end user requirements* are often unclear or even unknown. Integration and system test efforts are often limited to the a small set of well-known combinations of plug-ins. Several interviewed developers express that testing many combinations of plug-ins, versions, libraries, and operating systems is sheer impossible, and that they often only cover the latest versions of their plug-in dependencies. They also express to be reluctant to update their dependencies as this would require them to perform extensive manual testing.

Challenges developers face during plug-in testing are not restricted to creation and performance of testing, but include also comprehension of plug-in test code and understanding how well tests cover the system under test. During test execution of a plug-in test, hundreds of plug-ins and their extensions are loaded. Keeping track of which plug-ins and extensions are currently active in a test environment is a challenging task and an information need. Also, developers express the need to understand how test suites and test cases should test the extensions of foreign plug-ins. This can be challenging, starting with the need to be able to locate foreign tests that address certain extensions, services or plug-ins. As plug-in test suites often comprise a substantial amount of test code, test suites themselves are modularized and nested. This nesting and modularization of test suites makes it difficult to understand what is actually tested by which sub-test suite, and how the test environment has been set up.

In general, our studies revealed that integration testing per se is experienced as difficult by developers. Why this is the case was the focus of the second overarching research question:

What makes integration testing more difficult than unit testing?

Our investigation revealed that while there is a rich body of literature on unit testing (Gamma and Beck, 2003), literature on integration and system testing for plug-in-based systems is scarce. This unavailability of *plug-in testing knowledge* makes it hard for beginners and less experienced developers and testers to test plug-in based systems. Another pain point is the long *test execution time* of integration, GUI, or system tests. Developers report that due to the long execution time their work flow is disrupted, and their work performance decreases. Also, whereby *setting-up a test environment* for unit testing requires minimal effort as standard tooling (e.g., JUnit) exists, for integration, system, and GUI testing, the situation is different. Developers report that already setting up automated builds that execute integration tests that require the plug-in framework is difficult and not well supported. Developers also report that tool support for integration, system or GUI testing, for example the *Plug-in Development Environment tooling* and test infrastructure, does not meet their requirements. All of these technical hurdles have the effect that testing beyond unit scope is experienced as “*annoy-*

ing”, “*distracting*”, and “*painful*”.

Another difference between integration and unit tests surfaces during test design. Designing integration tests is challenging as developers have to account for the proper coverage of several combinations of plug-ins, but also third party-libraries, different operating systems or containers the system will have to run with, and also account for several versions of each of them. Selecting the right combinations is non trivial, and neither is the task to understanding which combinations are already covered by a test suite. Prompted by these challenges we wanted to provide developers with better tool support for integration testing activities, as visible in the third research question.

How can we support developers during understanding high level tests?

Through the challenges it was clear that tool support to understand high level tests, such as integration or system tests is needed. During interviews with 25 developers we were able to distill several information needs developers have when confronted with test suite comprehension. For example, developers explain that they have to understand what is tested, what is left out, and how the integration with other components is covered by tests. Whereby, the “what” can stand for components, services, plug-ins, or extension-points. Based on the information needs identified, we developed two distinct techniques that assist developers during understanding of high level test.

First, we developed five architectural views that can be used to understand test suites for plug-in-based systems from an integration perspective. The first view, the *Plug-in Modularization View* provides the developer with structural and organizational awareness with respect to the code-dependencies of plug-ins. Equipped with this basic structural knowledge, the second step is the analysis of the extension relations between plug-ins and the way they are exercised by the test suite, realized through the *Extension Initialization View*. The *Extension Usage* and *Service Usage Views* complete the picture by providing the developer with insight in the way the test suite exercises the actual methods involved in the extensions and services. Finally, the *Test Suite Modularization View* helps to relate this information to the different test suites executed. All these views are recovered from existing systems by means of static and dynamic analysis, and integrated in the Eclipse IDE. Note that, while some of the views are specifically developed for plug-in systems, the views can be adjusted to reflect the test suites and extension mechanism of other modular and dynamic systems such as service-oriented systems.

Based on an empirical study of the use of these views in several open source systems we could show that the technique is accurate and scales well to the system at hand. Further, we performed an initial user study with three developers and presented the tool to approximately 70 practitioners during the *Eclipse Testing Day*.¹ The overall responses were positive, and all of the three study participants

¹http://wiki.eclipse.org/Eclipse_Testing_Day_2011

expressed that the tool gives them a new perspective on plug-in test suites and that the visualizations are very useful.

The second technique, is a combination of *dynamic analysis* and the shared word count metric to establish relations between end-to-end and unit tests. These relations reflect the similarity between tests, whereby for example for one end-to-end tests, all unit tests are ranked based on their similarity (with the most similar one on top). In this study, we could show that the automatic derived relations reflect very well the similarity of the functionality covered by the test, and the rankings and relations resonate well with the understanding of similarity of a human expert. The technique could identify different states the system was in during testing, differentiated well between different objects involved in tests, and handle mocking activities. Based on the rankings, users can not only see which tests cover the same functionality, but are also able to identify misnamed tests and be aware of forgotten functionality. The evaluation of the technique shows that these rankings are suitable for developers unfamiliar with a test suite to get acquainted with the test code, and in general can support developers in their test suite maintenance activities.

How can we support developers during test maintenance?

Maintenance of test suites can become very costly, taken into account that nowadays software developers rely on extensive automated test suites, which can comprise a substantial amount of code. Therefore, long term success of automated testing is highly influenced by the maintainability of the test code (Meszaros, 2007). To support easier maintainability of a system, developers should follow best practices in style, such as that test methods should be clearly structured, well named and small in size, and code duplication should be avoided (Freeman and Pryce, 2009). Counterparts of best practices are so called *smells*, indicating poor solutions to recurring implementation and design problems (Fowler, 1999), or also called *test smells* if applied to test code (van Deursen et al., 2001).

To support developers during test suite maintenance, we investigated one specific, but very important part of a test: the code that initializes and configures the system under test (SUT), and puts the SUT in the right state required by the test. In line with Meszaros (2007), we refer to this part of a test as the *test fixture*. Especially for integration or system level tests, this part can contribute a substantial amount of the overall test code, and developers have to make conscious decisions about how to structure their fixture code and adjust their strategies during the evolution of the test code, otherwise test performance, understandability and maintainability are jeopardized.

In the course of this thesis, we investigated fixture related *test smells*, whereby we enhanced the set of test smells with three new *test fixture smells*. Furthermore, we designed a static analysis technique that can automatically detect six test fixture smells, and support developers in understanding and resolving the smells encountered. In two consecutive studies, we could show that test fixture smells occur in practice, and that developers are concerned about them. During a study

in which 13 developers participated, we could show that our tool *TestHound*, which implements the technique, is experienced as helpful by developers. As developers expressed the need to have continuous and immediate feedback on test fixture smells, we further investigated the evolution of test fixture smells to better understand how to avoid them, and to identify how a tight tool integration would be most beneficial. This study revealed that test fixture smells do not continually increase, but, at certain points in time, several emerge at the same time. Also, test fixture smells have the tendency to cluster in few particular smelly test classes. Test classes with a large number of test methods are particularly susceptible for test smells. Based on these findings, we developed several recommendations for test fixture smell avoidance, such as keeping test classes small and coherent, and using composition instead of inheritance to provide helper functionality to test classes.

In general, our technique to detect test fixture smells, and our findings on test smell avoidance can support developers to make or keep test code more comprehensible, maintainable, and to increase test performance.

7.3 Open Issues and Future Work

Testing practices. In this dissertation, we investigated the testing practices for plug-in-based systems, whereby we targeted one specific community, i.e., the Eclipse community. We assume that an expansion of this investigation to other plug-in communities and system, such as the Mozilla or Adobe plug-in system and community would be interesting, to reveal specific differences and commonalities of these communities. Also, in our work we focused on techniques and tools available for the Eclipse IDE. As many other widely used testing tools exist, such as the HP Quality center, Microsoft Test Manager, IBM Rational Functional Tester, or Selenium, an investigation of to which degree a change in tooling also impacts the adopted test practices is of interest. Subsequent efforts to reveal testing practices in other communities already exist, as for example Pham et al. (2013) investigate how social coding sites such as GitHub influence the testing practices.

Test quality. While, most of the techniques and tools developed in this dissertation focus on test comprehension, they also often gave an idea of the quality of a given test suite. We assume that an investigation of the extend in which the techniques can be useful to assess test quality might be beneficial. For example, the *ETSE* tool highlights the degree to which the integration between several plug-ins of the system under test has been covered in the test run. We can see that this information might be a base to derive adequacy criteria reached by a particular test suite. Further, we envision that the technique to establish similarity relation between test cases can be used to reveal blank spots in a test suite, as well as redundant test cases.

Dynamic Reorganizations of Test Suites. In this thesis, we concentrated on analyzing *test fixtures*, as well as *test similarities* covering the main test functionality to support developers during their test maintenance tasks. In future work, we envision to dynamically generate views on test suites. Such views should allow the developers to dynamically group test cases together that share specific properties, such as grouping tests that need the same test fixture, that test the same feature, or that need a comparable amount of time to execute.

Nowadays, developers have to group and organize test cases manually, for example by placing test methods within the same class, or defining test suite groupings. Unfortunately, to this point, developers can only select one specific type of organization per test or test suite, whereby, each organization has specific advantages and disadvantages with respect to test code comprehension, performance of test executions, and reusability of test code, depending on the task at hand. This implies having to deal with the disadvantages of a given test organization. Based on the insights gathered in previous studies, we envision that by combining our dynamic analysis techniques for measuring test case similarity, and our static analysis technique for test fixture analysis with software reconnaissance we are able to allow dynamic test reorganization.

Test Fixtures of Plug-in Tests. In this dissertation, we looked at test fixtures of unit tests in general. Plug-in tests are of interest for this research direction, as they often have many requirements on the state of the system under test. For example, the *Plug-in Development Environment* in Eclipse provides the developer with a specific plug-in testing framework to write tests, the *PDE tests*. The test runner of the *PDE tests* does not only need Java, but also requires the plug-in framework (i.e., the Eclipse framework) to be started. Furthermore, different tests then require a particular set of plug-ins to be loaded, before testing can start. In this dissertation, several of the systems under study have been plug-in based systems, nevertheless, we envision that future research tailors the technique for automated detection of *test fixture smells* to plug-in based systems. In such a context, the loading of plug-ins or third-party libraries that are unnecessary for the test execution can be seen as another form of a *test fixture smell*.

Bridging Research and Practice. In this dissertation, we sought to establish a close connection between industry and academia in order to

1. be aware of the problems faced in practice,
2. to reflect whether our solutions have the potential to be suitable for industry, and
3. to allow a *knowledge transfer* from academia to industry and vice versa.

As discussed in the introduction (Section 1.3), to ensure a connection to practice, we studied open and closed source systems, and adopted research methods

such as case study research, user studies and grounded theory. To establish contacts to industry, we participated in industry events and used social media (such as blogging and micro blogging) to communicate our (preliminary) results and research directions. As soon as we were able to arouse the interest of companies or individuals they were also willing to allocate time to participate in our studies. Involving industry in our studies was a time-consuming task. Next to establishing connections and communicating intents and results, following for example a grounded theory approach rigorously is a major investment of time. Nevertheless, we believe these efforts are justified, as they constitute an important means to establish connections between software engineering research and practice. Independent from the research topics that will follow, *bridging research and practice* will remain difficult, yet will become increasingly important. As Albert Einstein stated: *“In theory, theory and practice are the same. In practice, they are not.”*

Bibliography

- Adolph, S., Hall, W., and Kruchten, P. (2011). Using grounded theory to study the experience of software development. *Empirical Softw. Eng.*, 16(4):487–513.
- Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley.
- Beizer, B. (1990). *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.
- Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional.
- Borg, R. and Kropp, M. (2011). Automated acceptance test refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools (WRT)*, pages 15–21. ACM.
- Breugelmans, M. and Van Rompaey, B. (2008). TestQ: Exploring structural and maintenance characteristics of unit test suites. In *1st Int’l Workshop on Academic Software Development Tools and Techniques*.
- Bryant, A. and Charmaz, K., editors (2007). *The SAGE Handbook of Grounded Theory*. SAGE.
- Cabral, I., Cohen, M. B., and Rothermel, G. (2010). Improving the testing and testability of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond, SPLC’10*, pages 241–255, Berlin, Heidelberg. Springer-Verlag.
- Chatley, R., Eisenbach, S., Kramer, J., Magee, J., and Uchitel, S. (2004). Predictable dynamic plugin systems. In *7th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 129–143. Springer-Verlag.
- Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. on Softw. Engineering*, 20(6):476–493.

- Chikofsky, E. and Cross, J.H., I. (1990). Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17.
- Corbi, T. A. (1989). Program understanding: challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306.
- Corbin, J. M. and Strauss, A. (1990). Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13:3–21.
- Cornelissen, B., Moonen, L., and Zaidman, A. (2008). An assessment methodology for trace reduction techniques. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 107–116. IEEE CS.
- Cornelissen, B., van Deursen, A., Moonen, L., and Zaidman, A. (2007). Visualizing testsuites to aid in software understanding. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 213–222. IEEE Computer Society.
- Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702.
- Creswell, J. W. and Vicki (2006). *Designing and Conducting Mixed Methods Research*. Sage Publications, Inc, 1 edition.
- Dagenais, B. and Robillard, M. P. (2010). Creating and evolving developer documentation: understanding the decisions of open source contributors. In *SIG-SOFT FSE*, pages 127–136.
- Demeyer, S., Ducasse, S., and Nierstrasz, O. (2003). *Object-oriented reengineering patterns*. Morgan Kaufmann.
- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 1. a. edition.
- Feathers, M. (2004). *Working Effectively with Legacy Code*. Prentice Hall.
- Fowler, F. J. (2002). *Survey Research Methods*. Sage Publications, Thousand Oaks, CA.
- Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley.
- Freeman, S. and Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1st edition.
- Galli, M., Lanza, M., and Nierstrasz, O. (2004a). Towards a Taxonomy of SUnit Tests? In *International Smalltalk Conference*.

- Galli, M., Lanza, M., Nierstrasz, O., and Wuyts, R. (2004b). Ordering broken unit tests for focused debugging. In *Int'l Conf. Softw. Maintenance (ICSM)*, pages 114–123. IEEE.
- Gamma, E. and Beck, K. (2003). *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley.
- Garousi, V. and Varma, T. (2010). A replicated survey of software testing practices in the Canadian province of Alberta: What has changed from 2004 to 2009? *J. Syst. Softw.*, 83:2251–2262.
- Glaser, B. and Strauss, A. (1967). *The discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction.
- Golafshani, N. (2003). Understanding reliability and validity in qualitative research. *The Qualitative Report*, 8(4):597–606.
- Greiler, M., Deursen, A. v., and Storey, M.-A. (2012a). Test confessions: a study of testing practices for plug-in systems. In *Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012*, pages 244–254, Piscataway, NJ, USA. IEEE Press.
- Greiler, M., Gross, H.-G., and van Deursen, A. (2010). Understanding plug-in test suites from an extensibility perspective. In *Proceedings 17th Working Conference on Reverse Engineering*, pages 67–76. IEEE.
- Greiler, M. and van Deursen, A. (2012). What your plug-in test suites really test: An integration perspective on test suite understanding. *Empirical Software Engineering*, pages 1–42.
- Greiler, M., van Deursen, A., and Storey, M.-A. (2011). What Eclipsers think and do about testing: A grounded theory. Technical Report SERG-2011-010, Delft University of Technology.
- Greiler, M., van Deursen, A., and Storey, M.-A. (2013a). Automated detection of test fixture strategies and smells. In *Proceedings of the International Conference on Software Testing, Verification and Validation*.
- Greiler, M., van Deursen, A., and Zaidman, A. (2012b). Measuring test case similarity to support test suite understanding. In *Proceedings of the International Conference on Objects, Models, Components, Patterns*, pages 91–107. Springer.
- Greiler, M., van Deursen, A., Zaidman, A., and Storey, M.-A. (2013b). Strategies for avoiding text fixture smells during software evolution. In *Working Conference on Mining Software Repositories, to appear*.
- Gubrium, J., Holstein, J., Marvasti, A., and McKinney, K., editors (2012). *The SAGE Handbook of Interview Research*. SAGE Publications, Inc.

- Harrison, W. (2006). Eating your own dog food. *IEEE Softw.*, 23:5–7.
- Hartmann, J., Imoberdorf, C., and Meisinger, M. (2000). UML-Based integration testing. In *International Symposium on Software Testing and Analysis*, pages 60–70. ACM.
- Henderson-Sellers, B. (1996). *Object-oriented metrics: measures of complexity*. Prentice-Hall.
- Hermans, F., Pinzger, M., and van Deursen, A. (2011). Supporting professional spreadsheet users by generating leveled dataflow diagrams. In Gall, H. and Medvidovic, N., editors, *Proceedings 33rd International Conference on Software Engineering (ICSE 2011)*. ACM.
- Hindle, A., Godfrey, M. W., and Holt, R. C. (2010). Software process recovery using recovered unified process views. In *Proceedings 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–10. IEEE Computer Society.
- Hopkins, W. G. (2000). *A new view of statistics*. Internet Society for Sport Science.
- Hurdugaci, V. and Zaidman, A. (2012). Aiding developers to maintain developer tests. In *Conf. Softw. Maintenance and Reengineering (CSMR)*, pages 11–20. IEEE CS.
- Jorgensen, P. C. and Erickson, C. (1994). Object-oriented integration testing. *Communications of the ACM*, 37(9):30.
- Kagdi, H., Collard, M. L., and Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131.
- Koochakzadeh, N. and Garousi, V. (2010). Tecrevis: a tool for test coverage and test redundancy visualization. In *Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques*, TAIC PART’10, pages 129–136, Berlin, Heidelberg. Springer-Verlag.
- Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.
- Lee, J., Kang, S., and Lee, D. (2012). A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC ’12*, pages 31–40, New York, NY, USA. ACM.
- Lehman, M. M. (1984). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221.

- Li, W. and Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122.
- Mariani, L., Papagiannakis, S., and Pezze, M. (2007). Compatibility and regression testing of cots-component-based software. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 85–95, Washington, DC, USA. IEEE Computer Society.
- Marinescu, R. (2001). Detecting design flaws via metrics in object-oriented systems. In *Proc. of the Int'l Conf. on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 173–182. IEEE CS.
- Marquardt, K. (1999). Patterns for plug-ins. In *Proceedings 4th European Conference on Pattern Languages of Programs (EuroPLOP)*, page 37pp, Bad Irsee, Germany.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 1 edition.
- Mayer, J., Melzer, I., and Schweiggert, F. (2003). Lightweight plug-in-based application development. In *International Conference NetObjectDays, NODE 2002*, pages 87–102. Springer-Verlag.
- Memon, A., Porter, A., and Sussman, A. (2010). Community-based, collaborative testing and analysis. In *Proceedings of the FSE/SDP workshop on Future of software engineering research, FoSER '10*, pages 239–244, New York, NY, USA. ACM.
- Mens, T., Fernández-Ramil, J., and Degrandt, S. (2008). The evolution of eclipse. In *Proceedings 24th IEEE International Conference on Software Maintenance (ICSM)*, pages 386–395. IEEE.
- Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.
- Mikhajlov, L. and Sekerinski, E. (1998). A study of the fragile base class problem. In *Proceedings European Conference on Object-Oriented Programming (ECOOP)*, pages 355–382. Springer-Verlag.
- Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11:309–346.
- Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Trans. on Softw. Engineering*, 36(1):20–36.

- Muccini, H. and Hoek, A. V. D. (2003). Towards testing product line architectures. In *In: International Workshop on Testing and Analysis of Component Based Systems*, pages 111–121.
- Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R., and Wong, K. (2000). Reverse engineering: a roadmaps. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 47–60, New York, NY, USA. ACM.
- Neukirchen, H. and Bisanz, M. (2007). Utilising code smells to detect quality problems in ttcn-3 test suites. In *Proc. of the Int'l Conf. on Testing of Communicating Systems and the Int'l Workshop on Formal Approaches to Testing of Software (TestCom/FATES)*, pages 228–243. Springer.
- Ng, S. P., Murnane, T., Reed, K., Grant, D., and Chen, T. Y. (2004). A preliminary survey on software testing practices in Australia. In *Proceedings of the 2004 Australian Software Engineering Conference*, ASWEC '04, pages 116–, Washington, DC, USA. IEEE Computer Society.
- Onwuegbuzie, A. J. and Leech, N. L. (2007). Validity and qualitative research: An oxymoron? *Quality & Quantity*, 41(2):233–249.
- Peters, R. and Zaidman, A. (2012). Evaluating the lifespan of code smells using software repository mining. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 411–416. IEEE.
- Pezzè, M. and Young, M. (2008). *Software Testing and Analysis*. Wiley.
- Pham, R., Singer, L., Liskin, O., Figueira Filho, F., and Schneider, K. (2013). Creating a Shared Understanding of Testing Culture on a Social Coding Site. In *Proceedings of the 35th International Conference on Software Engineering (to appear)*.
- Pinto, L. S., Sinha, S., and Orso, A. (2012). Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 33:1–33:11, New York, NY, USA. ACM.
- Pohl, K. and Metzger, A. (2006). Software product line testing. *Commun. ACM*, 49:78–81.
- Raja, U. and Tretter, M. J. (2009). Antecedents of open source software defects: A data mining approach to model formulation, validation and testing. *Inf. Technol. and Management*, 10:235–251.
- Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

- Rehmand, J., Jabeen, F., Bertolino, A., and Polini, A. (2007). Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133.
- Reis, S., Metzger, A., and Pohl, K. (2007). Integration testing in software product line engineering: A model-based technique. *Lecture Notes In Computer Science*, pages 321–335.
- Rigby, P. C. and Storey, M.-A. (2011). Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 541–550.
- Rothermel, G. and Harrold, M. (1998). Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419.
- Rountev, A., Milanova, A., and Ryder, B. (2004). Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387.
- Rubio, D. (2009). *Testing with Spring and OSGi*, chapter 9, pages 331–359. Apress, Berkeley, CA.
- Shavor, S., D’Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., and McCarthy, P. (2005). *The Java Developer’s Guide to Eclipse*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Stephen, G. A. (1994). *String searching algorithms*. World Scientific Publishing Co.
- The OSGi Alliance (2011). *OSGi Service Platform Core Specification; Release 4, Version 4.3*. <http://www.osgi.org>.
- Tilley, S. R., Smith, D. B., and Paul, S. (1996). Towards a framework for program understanding. In *Workshop on Program Comprehension (WPC)*, pages 19–28.
- van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., and Riva, C. (2004). Symphony: View-driven software architecture reconstruction. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA’04)*, pages 122–134. IEEE Computer Society Press.
- van Deursen, A., Moonen, L., Bergh, A. v. d., and Kok, G. (2001). Refactoring test code. In *Proc. of the Int’l Conf. on Extreme Programming and Flexible Processes (XP)*, pages 92–95. University of Cagliari.
- van Deursen, A., Moonen, L., van Den Bergh, A., and Kok, G. (2002). Refactoring test code. In *Extreme Programming Perspectives*, pages 141–152. Addison Wesley.

- Van Rompaey, B., Du Bois, B., and Demeyer, S. (2006). Characterizing the relative significance of a test smell. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, pages 391–400. IEEE CS.
- Van Rompaey, B., Du Bois, B., Demeyer, S., and Rieger, M. (2007). On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Trans. on Softw. Engineering*, 33(12):800–817.
- Voelter, M. (2001). Pluggable component: A pattern for interactive system configuration. In Dyson, P. and Devos, M., editors, *Proceedings of the 4th European Conference on Pattern Languages of Programms (EuroPLoP '1999), Irsee, Germany, July 7-11, 1999*, pages 291–304. UVK - Universitaetsverlag Konstanz.
- von Krogh, G., Spaeth, S., and Lakhani, K. R. (2003). Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241.
- Weiss, S., Indurkha, N., Zhang, T., and Damerau, F. (2004). *Text Mining: Predictive Methods for Analyzing Unstructured Information*. SpringerVerlag.
- Wermelinger, M. and Yu, Y. (2008). Analyzing the evolution of eclipse plugins. In *Proceedings of the 2008 international working conference on Mining software repositories, MSR '08*, pages 133–136, New York, NY, USA. ACM.
- West, J. and Siobhán, O. (2008). The role of participation architecture in growing sponsored open source communities. *Industry & Innovation*, 15(2):145–168.
- Weyuker, E. J. (1998). Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59.
- Yin, R. K. (2003). *Case study research: design and methods*. Sage Publications, 3rd edition.
- Yoo, S., Harman, M., Tonella, P., and Susi, A. (2009). Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis, ISSTA '09*, pages 201–212, New York, NY, USA. ACM.
- Yoon, I., Sussman, A., Memon, A., and Porter, A. (2013). Testing component compatibility in evolving configurations. *Inf. Softw. Technol.*, 55(2):445–458.
- Zaidman, A., van Rompaey, B., Demeyer, S., and van Deursen, A. (2008). Mining software repositories to study co-evolution of production & test code. In *Proceedings 1st International Conference on Software Testing Verification and Validation (ICST)*, pages 220–229. IEEE Computer Society.
- Zaidman, A., Van Rompaey, B., van Deursen, A., and Demeyer, S. (2011). Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364.

Appendix: Grounded Theory Study

A.1 Resulting Collection of Codes

As a result of the interview analysis process, a collection of codes emerged. Our coding process was open, allowing codes and concepts to emerge freely; Through (constant) comparison and grouping, the coding structure as presented here emerged. Here we summarize the eventual set of codes resulting from this analysis.

We use a simplified presentation into a three-level hierarchy (category, concept, code). With each concept we associate a question, where each code belonging to the concept can be read as an answer to that question

In principle, the codes (or even concepts) can be grouped in multiple ways, sometimes in additional subgroups. For example, *test execution time* is listed as a *barrier*, but could also be grouped under *integration testing*. Here we present the dominant decomposition, putting codes in the most relevant concept only.

A.1.1 Category 1. Practices

Testing practices that Eclipsers mention or adopt.

Concept 1.1. Supporting Processes

Which processes (such as requirement documentation, issue tracking) are in place to support testing activities? Who tests the code/system?

1.1.1	<i>Issue tracker</i>	Requirements are documented in an issue tracking system such as Bugzilla.
1.1.2	<i>Requirement source</i>	It is clear who defines the requirements.
1.1.3	<i>Developer testing</i>	Testing is only done by the developers.
1.1.4	<i>Hybrid testing</i>	The QA team as well as the developer team are involved in testing.

1.1.5	<i>Tester status</i>	Pure testing activities have a lower status than development.
-------	----------------------	---

Concept 1.2. Unit Testing

In what way is unit testing used in Eclipse projects?

1.2.1	<i>Key practice: Unit testing</i>	Unit testing is the key test practice.
1.2.2	<i>Preference</i>	Unit testing is the preferred practice.
1.2.3	<i>Coverage</i>	Coverage of the code is measured.
1.2.4	<i>No coverage</i>	Coverage of the code is not measured.
1.2.5	<i>Confidence</i>	A unit test suite gives confidence when making a change or when refactoring.
1.2.6	<i>Limited confidence</i>	Relying too much on automated tests or coverage can be risky.
1.2.7	<i>Unit testability</i>	Units are designed to be testable by unit tests.
1.2.8	<i>Unit non-tested</i>	Units that cannot be made testable are not subjected to unit testing.
1.2.9	<i>Fast execution</i>	Even a substantial unit test suite executes fast.

Concept 1.3. Beyond Unit Testing

How are test practices other than unit testing applied?

1.3.1	<i>Test automation</i>	Project aims at obtaining a large degree of test automation.
1.3.2	<i>Hardware integration</i>	For embedded systems, integration testing is guided by hardware integration.
1.3.3	<i>Continuous integration</i>	Automated build and test servers are used to conduct continuous integration.
1.3.4	<i>Unit vs integration testing</i>	The amount of integration testing done depends on the amount of unit testing done.
1.3.5	<i>Fault location</i>	During integration testing fault localization is hard.
1.3.6	<i>GUI testing</i>	User interface testing tools are used to do automated testing from the GUI.
1.3.7	<i>GUI maintainability</i>	Problems with maintainability of GUI test cases are reported.
1.3.8	<i>GUI non-tested</i>	No test effort is made to cover the GUI with automatic tests.

A.1.2 Category 2. Plug-in Specific Integration Testing

Testing practices that are specifically targeting plug-in-based systems.

Concept 2.1. PDE tests

How are tests written using the Eclipse Plug-in Development Environment (PDE) test framework?

2.1.1	<i>Workbench dependencies</i>	The PDE runner is used since the test depends on the workbench.
2.1.2	<i>PDE as integration test</i>	The PDE-Junit framework is used to write integration tests.
2.1.3	<i>PDE as unit test</i>	The PDE-Junit framework is used to write unit tests.
2.1.4	<i>Headless PDE</i>	The PDE tests are executed without the UI (i.e. in headless mode).

Concept 2.2. Plug-in characteristic

To what specific test practices does the plug-in nature lead?

2.2.1	<i>No influence</i>	The plug-in characteristic has no influence on testing.
2.2.2	<i>Modularization</i>	The plug-in mechanism is used for modularizing test suites.
2.2.3	<i>Extension points</i>	Test strategies for Eclipse extensions and extension points are adopted.
2.2.4	<i>Registration untested</i>	The plug-in and extension point registration mechanisms are untested.
2.2.5	<i>Plug-in testability</i>	Eclipse plug-ins can be hard to test if they do not expose their (internal) functionality.
2.2.6	<i>Eco-system integration</i>	Plug-ins are exercised in the context of the Eclipse runtime environment.
2.2.7	<i>GUI based</i>	Automated GUI testing is used to test GUI based Eclipse applications.
2.2.8	<i>No eclipse integration</i>	Tests do not require the Eclipse or OSGi runtime.

Concept 2.3. Cross-feature integration

How is integration with third party plug-ins tested?

2.3.1	<i>Plug-in independence</i>	Plug-ins are considered independent, and combinations are not tested.
2.3.2	<i>Play nicely</i>	Plug-ins are supposed to work together.
2.3.3	<i>Demand driven</i>	Integration between plug-ins is only tested if there is a specific feature / bug requiring the execution of multiple plug-ins.
2.3.4	<i>Manual combinations</i>	Different combinations are installed and compatibility is manually tested.
2.3.5	<i>No automated cross-tests</i>	No automated tests for cross-feature integration exist.

2.3.6	<i>Unpredictable</i>	It can not be foreseen which combinations will be incompatible.
2.3.7	<i>Combination issues</i>	Actually experienced issues between different plug-ins are reported.

Concept 2.4. Versioning

How is testing against different versions of the platform or third party components conducted?

2.4.1	<i>Build system</i>	The software is build using different versions of the platform.
2.4.2	<i>External systems</i>	Testing against different versions of external systems beyond the Eclipse eco-system is conducted.
2.4.3	<i>No automated versions</i>	No automated tests exist to test against different versions.
2.4.4	<i>Manual versions</i>	Version compatibility is tested manually.
2.4.5	<i>Limited versions</i>	Only a limited set of versions is tested.
2.4.6	<i>Assert compatibility</i>	Version ranges include versions that are asserted but not tested to work.
2.4.7	<i>Update rarely</i>	Updating versions of dependencies or the platform is done rarely.
2.4.8	<i>Unfeasible</i>	It is unfeasible to test for all version combinations.

A.1.3 Category 3. Test Barriers

Barriers that hinder adoption of integration test practices.

Concept 3.1. Testing Barriers

Which test barriers reduce the amount of plug-in specific integration testing?

3.1.1	<i>Responsibility</i>	Unclear who is or feels responsible for system integration issues.
3.1.2	<i>End user requirements</i>	System customization can lead to unclear or unknown end user requirements.
3.1.3	<i>Ownership</i>	Lack of ownership or controllability over dependent plug-ins and code.
3.1.4	<i>Plug-in testing knowledge</i>	Lack of technical knowledge needed to successfully perform a plug-in specific test strategy.
3.1.5	<i>Set-up build system</i>	Too much time, effort and knowledge needed to get the infrastructure ready to use an integration testing approach.
3.1.6	<i>Execution time</i>	Execution time of tests is too long.
3.1.7	<i>Eclipse testability</i>	Eclipse is a highly coupled, and hard to test system. See also 2.2.5/ <i>Plug-in testability</i>
3.1.8	<i>PDE integration tooling</i>	The PDE framework has not been designed for integration testing.

A.1.4 Category 4. Compensation Strategies

Actions taken in order to compensate for not adopting certain test strategies.

Concept 4.1. Self hosting

How is the project team involved in testing in addition to traditional test activities?

4.1.1	<i>Self-hosting</i>	The development team itself is also acting as user.
-------	---------------------	---

Concept 4.2. Community involvement

How are users or customers involved in testing activities?

4.2.1	<i>Manual testing</i>	Users are involved in conducting manual tests.
4.2.2	<i>Software usage</i>	Users are involved in using early versions of the software, e.g., pre-releases.
4.2.3	<i>Operating systems</i>	The community participates in testing against different operating systems.
4.2.4	<i>Multiple versions</i>	The community participates in testing against different versions of the workbench or required plug-ins.
4.2.5	<i>Compatibility</i>	The community participates in testing the compatibility between several plug-ins.
4.2.6	<i>GUI community</i>	The community is involved in user interface testing.
4.2.7	<i>Filing bug reports</i>	Community files bug reports.
4.2.8	<i>Feedback</i>	Users try out early versions and give feedback.
4.2.9	<i>Customer involvement</i>	Customers are involved in the software engineering process.

Concept 4.3. Developer involvement

How are developers involved in testing activities?

4.3.1	<i>Automated testing</i>	Downstream projects are exercising plug-ins used.
4.3.2	<i>Release train</i>	Multiple Eclipse plug-ins are released at the same time in the release train.
4.3.3	<i>Ecosystem</i>	Projects make an effort to work together to form a coherent ecosystem.
4.3.4	<i>Plug-in symbiosis</i>	Projects improve other projects they depend on.
4.3.5	<i>Providing patches</i>	The community provides patches for bugs.

Concept 4.4. Openness

How open are the projects and processes used?

4.4.1	<i>Releases</i>	The release strategy includes nightly / unstable releases.
-------	-----------------	--

4.4.2	<i>Communication</i>	Open communication is setup to ensure traceability, visibility and transparency.
4.4.3	<i>Test request</i>	The community is explicitly requested to participate in testing.
4.4.4	<i>Opening closed software</i>	Development of closed source projects is opened up to strengthen customer involvement.

A.2 Key Quotations

The coding process was based on transcripts we made of the interviews. Here we give, for each of the participants, the most important quotes together with a selection of the codes we used to tag those quotes. In the paper itself the quotes are organized by code, i.e., presented when we are discussing a given part of the emerging theory.

Participant P1

1.3.8	<i>GUI non-tested</i>	<i>"We don't do functional testing since 80% of the code is already tested with unit tests"</i>
1.2.5	<i>Confidence</i>	<i>"When you design unit tests in a good way, then refactoring is easier."</i>
1.2.9	<i>Fast execution</i>	<i>"The good thing about unit tests is that they are fast, whereby PDE tests are slow."</i>
3.1.6	<i>Execution time</i>	
2.1.3	<i>PDE as unit test</i>	<i>"Legacy code can be problematic when unit testing, therefore sometimes PDE unit tests are misused as mini integration tests."</i>
3.1.7	<i>Eclipse testability</i>	
2.1.1	<i>Workbench dependencies</i>	<i>"The problem begins when a JUnit test grows into a PDE test, because of the dependencies on the workbench, or other Eclipse APIs."</i>
2.1.3	<i>PDE as unit test</i>	
3.1.7	<i>Eclipse testability</i>	
1.3.4	<i>Unit vs integration testing</i>	<i>"There are different ideas on how to test, but with most of them I do not agree. I think basically only unit testing is good."</i>
1.2.2	<i>Preference</i>	

Participant P2

2.2.1	<i>No influence</i>	<i>"I am not sure if there is a need to test if extensions correctly support the extension point, because it is mostly a registration thing."</i>
4.2.8	<i>Feedback</i>	<i>"It was sort of this open source process, and we expected people that actually use it to come up with ideas as well and to improve it."</i>
1.1.2	<i>Requirement source</i>	<i>"Usually you start small, with something you can show, something that is working. Then, you get feedback from the community, and then you develop further."</i>
1.3.8	<i>GUI non-tested</i>	<i>"We don't have GUI tests, because we don't have much user interaction."</i>

Participant P3

1.1.4	<i>Hybrid testing</i>	<i>"We have both. Some people are pure testers, some are pure developers because they think like developers and have not done testing. It's listening to the skill-sets of your people. Knowing and not pushing people to do things that they are not good at."</i>
1.1.5	<i>Tester status</i>	

2.2.7	GUI based	<i>"Eclipse wizards are really interesting in terms of test design. You really have to think about how to structure tests so they can be reused."</i>
1.1.4	Hybrid testing	<i>"We have our testers in very close contact with the developers. It's a way of getting discussions going."</i>
4.2.9	Customer involvement	<i>"We work very closely with the customer to do things in an agile way. He can change things, he can swap out features. We also have a weekly meeting that we call show & tell. The customer comes to that and he sees how things are going."</i>
4.2.9	Customer involvement	<i>"Without the continuous involvement of the customer throughout, and without the customer being there, you probably find many more problems at the end."</i>
4.2.9	Customer involvement	<i>"I would say we are in a reasonably unique situation in the sense that our customer is an internal person. So, we can always have a meeting with him."</i>
1.1.2	Requirement source	<i>"We manage our requirements over many sprints: when we make them, we discuss them with the whole team. It takes a long time, but it means that everybody on the team knows how it's gonna work. Then any problems come up in the meeting and not after 3 weeks of developing. Then the features are very prominent in everybody's minds."</i>
1.2.5	Confidence	<i>"Another situation where we use unit tests a lot is when we are going back to an older piece of code that has to be refactored."</i>
1.2.5	Confidence	<i>"It gives you a certain level of comfort to know that when you make a change and you break something, that it would be apparent in your test case."</i>
2.2.2	Modularization	<i>"The modularization abilities of OSGi are interesting for test design. Then you really have to think about how to structure tests so they can be reused."</i>
1.3.1	Test automation	<i>"We make a joint decision whether it should be automated or whether it should be manually tested. There are a lot of things that come into play when automating a test. We look at how easy is it will automate, or is it going to be brittle, and weigh that up against how important that is."</i>
1.3.3 1.2.6	Continuous integration Limited confidence	<i>"One of the traps that a team can fall into is, because the tests are running every night, they think that the safety net is bigger than it is."</i>
1.2.4	No coverage	<i>"We do not measure coverage yet. One of the things we are working on is to get a useful coverage criteria. We measured it in the past, but we did not know what the numbers really meant for us. So now, we are investigating what these numbers mean for us."</i>

1.3.6	GUI testing	<i>"I think it is so incredibly important, to always have that customer perspective. Anything that will affect the user, we prefer to write as an acceptance test."</i>
-------	-------------	---

Participant P4

3.1.5	Set-up build system	<i>"the difficulty of integration testing Eclipse plug-ins starts with the set-up of the build – that's difficult."</i>
3.1.4	Plug-in testing knowledge	<i>"We only use unit testing because integration testing is complicated. Most people just do not do it. Often it's even not necessary."</i>
4.4.2	Communication	<i>"The advantage of open communication is that others see it as well and can participate. Then, solutions to problems can even come from outside of the original team."</i>
3.1.4	Plug-in testing knowledge	<i>"Why [testing] is so difficult? For Web projects, you find good templates. For Eclipse, you don't. There are some approaches, but nothing you could call best practice. Testing has to be easier. Especially for testing plug-ins, we would need some best practices. It would be great to have concrete tutorials and concrete solutions. Big companies, they have their processes and strategies working, but it's difficult for small companies."</i>

Participant P5

1.1.3	Developer testing	<i>"Tester and developer, that's one person. From our view, it does not make sense to have a dedicated test team, which has no idea about what the software does and can only write some tests. The person that writes the code, that one knows what has to be tested."</i>
1.1.3	Developer testing	<i>"I think that the developer is better suited for integration testing than the tester because they know the application better. Otherwise, you have to explain to the tester again how that works and what the requirements are. That's double the work."</i>
1.3.2	Hardware integration	<i>"We have automated integration tests which run on the PC and on the devices. We have four device setups, running the same software, and the tests are run on each of the devices."</i>
1.3.3 3.1.6	Continuous integration Execution time	<i>"There are tests that run fast, which are executed whenever you commit in SVN through CruiseControl. Then every two hours, a larger build is run, which also executed the tests requiring more time. And in the night, the very slow tests run, which take several hours."</i>

2.2.2	Modularization	<i>"Unit tests run in the same bundle as the code they test; integration tests run in their own bundle."</i>
2.2.5 3.1.7	Plug-in testability Eclipse testability	<i>"Sometimes you have to extend the bundle you want to test with a bit of extra functionality in order to be able to do proper integration testing."</i>
1.3.6	GUI testing	<i>"We specify all GUI tests with eFitness."</i>
3.1.6	Execution time	<i>"Integration testing just takes longer, and therefore, it runs during the night. Then it does not disturb anybody when all the devices beep and bleep."</i>
4.2.9	Customer involvement	<i>"The customer plays a important role during the development. We always try to deliver something - in relatively short intervals - to always get feedback. The customer, for example, uses Scrum, and we send a snapshot every 2-3 weeks, and the customer then tests all the new features that have been added in the last sprint. All this is actually quite extensively tested by the customer."</i>
1.1.2	Requirement source	<i>"Requirements are defined at the beginning of a project. That's developed together with the customer. In principle, the outcome is a functional specification, which is also used for testing."</i>
1.3.1 1.2.6	Test automation Limited confidence	<i>"At the beginning, most parts have been tested manually. 3-4 years ago, we started investing a lot of time in automating tests, and we want to automate as much as possible. The only problem might be to trust the test outcome too much."</i>
1.2.3	Coverage	<i>"That's integrated in the continuous integration process. We have around 60-80% coverage."</i>

Participant P6

1.1.2	Requirement source	<i>"The requirements are recorded in Bugzilla, but not in detail. The person developing a feature enhancement, that one aligns his ideas with the ones of the others as much as he thinks is useful and necessary. That's a self-responsible process. We assume that all committers take responsibility for the overall product."</i>
4.2.1 2.3.2 2.3.7 4.4.1	Manual testing Play nicely Combination issues Releases	<i>"We do not test cross-feature integration automatically. There is the possibility that other plug-ins have side effects with ours, but if you are part of the simultaneous release, then there is a set of rules that you must obey. One rule states: 'Play nicely together'. Surely, that's only written on paper. We ensure that by having regular builds in which we put all things together, and then we ask the community to try the combinations. And usually they do that in the last weeks before the release candidate goes out. And until now, pre-existent problems became apparent in this phase."</i>

3.1.6 2.1.4	Execution time Headless PDE	<i>“The long execution time is really bad. A big problem. Turnaround times must be as short as possible. And that’s a problem with the PDE builds. We run them most of the time in headless mode, if possible, meaning the tests do not need the UI.”</i>
3.1.6 2.1.4	Execution time Headless PDE	<i>“But if we need the UI, then it is really slow, and then this also means it’s multithreaded, which makes it even more difficult. On the other hand, we have to test that too.”</i>
3.1.6 1.2.9	Execution time Fast execution	<i>“Even with unit tests, it is important that they are fast. You always have to keep an eye on where you spend the time during a turnaround, and then it is also important to look at the tests to see where you can speed things up.”</i>
2.1.1 2.1.3	Workbench dependencies PDE as unit test	<i>“A PDE test is similar to a unit test. Actually, it is a unit test, with the only difference that it is based on a large web of objects. So, typically the whole workbench is started.”</i>
2.1.2 1.3.3	PDE as integration test Continuous integration	<i>“Then, we have tests like start the workbench, create two projects, create a file and save it. Those are the real integration tests – they need everything. You really cannot mock that anymore. That does not pay off. These tests run as PDE tests and also in the continuous integration.”</i>
1.3.5	Fault location	<i>“A disadvantage of integration testing is that faults are hard to locate, because if something goes wrong you have executed one million lines. It is more difficult to understand where the fault occurred than if you execute 10 lines.”</i>
3.1.6 1.2.2 1.3.4	Execution time Preference Unit vs integration testing	<i>“We prefer unit tests. Fast running, small tests. Integration tests, or functional tests, are also nice and important, but we would prefer to test everything without the UI.”</i>
3.1.7 3.1.3	Eclipse testability Ownership	<i>“It is not our code that we have problems with. It is the code of the Eclipse platform, that from JFace and SWT. And that code we do not like to test. We have to do it anyway. We do not like it, because it is hard to produce small tests, which means you always have a lot of infrastructure.”</i>
1.1.2 4.2.8 4.3.5	Requirement source Feedback Providing patches	<i>“We have Bugzilla, and through that we communicate the most. Also with the users and the committers. People add patches, and suggestions about how this or that can or should be done.”</i>
3.1.7	Eclipse testability	<i>“The problem is that the Eclipse platform is very hard to test, because components are highly coupled and interfaces are huge, and all is based on a singleton state. This is very hard to decouple.”</i>

Participant P7

3.1.7 3.1.3	<i>Eclipse testability Ownership</i>	<i>"[Project 7] is a plug-in, but it is not an end-user plug-in. It is a half-way plug-in. Other plug-ins build on top of [Project 7], so integration testing would need to include some other components. It is not the final, the whole thing."</i>
3.1.2 3.1.3	<i>End user requirements Ownership</i>	<i>"Integration testing should be done at least in strong collaboration with the developers of the end-user plug-in. One example is the syntax highlighting. Only when I know about the language and have the syntax highlighter can I test it and see whether it was successful or not. I need some third party component."</i>
3.1.3	<i>Ownership</i>	<i>"And you never know, once you write a good test, it will be obsolete with the next version of Eclipse."</i>
1.3.7	<i>GUI maintainability</i>	<i>"We tried GUI testing for a while but it was too much work."</i>
4.4.2	<i>Communication</i>	<i>"We can just call each other, but it is better to use the mailing list and let others know that there are some considerations. And also our users read the mailing list."</i>
4.4.2	<i>Communication</i>	<i>"We try to communicate via the mailing list because we want to give some visibility into the activity of the project."</i>
4.2.5	<i>Compatibility</i>	<i>"We have [major changes that break the API] once in a while. We know most of the users and we normally investigate their source code to see if a change is going to be a problem. If it is going to be a real problem, we usually do not do it. We only do small changes."</i>
4.4.2 4.2.5	<i>Communication Compatibility</i>	<i>"If there is really an API change, then we mail all the users personally, saying that something new is coming."</i>
4.2.1 4.4.1	<i>Manual testing Releases</i>	<i>"So, what now happens is that [person X] also manages a lot of applications of [our project], like the Cobol IDE and another IDE, and he just makes a pre-release within his small group that use the tools, and lets them test it. So, it is really manual testing."</i>
1.2.3 1.2.1	<i>Coverage Key practice: Unit testing</i>	<i>"My part is mainly tested with unit tests. I have 85% coverage, but actually I measured it only when I wrote the tests. Now, I only maintain it and I do not measure anymore."</i>
1.2.6	<i>Limited confidence</i>	<i>"It is very easy to get high coverage with unit tests, like you generate mock objects and you run it. But, the hard thing is: what do you check when the method is finished, i.e., you must know about the post-conditions. So, are people actually doing useful testing, or are they just going through the motions with the unit tests?"</i>

2.4.3	<i>No automated versions</i>	<p><i>“Our project requirements specify that the software has to work with older versions of the platform, ranging from Eclipse 3.1 to 3.5. We do not have automated tests to verify the compatibility, but when we release [Project 7], then we test it. So, when we do manual testing, then we test for different versions of Eclipse.”</i></p>
4.4.1	<i>Releases</i>	
2.4.4	<i>Manual versions</i>	

Participant P8

4.4.2	<i>Communication</i>	<p><i>“We have a lot of Skype-to-Skype communication, because it is more efficient. For topics relevant for the bigger community we are often copying the communication records to the Bugzilla.”</i></p>
4.2.4 3.1.1 4.3.2 2.4.6 2.4.5	<i>Multiple versions Responsibility Release train Assert compatibility Limited versions</i>	<p><i>“We only test the latest available versions of our dependencies, those that are together in the release train. Those we officially support. That does not mean that others are not working, but in the worst case everybody has to try it out themselves.”</i></p>
4.2.4 3.1.1 2.4.6 2.4.5	<i>Multiple versions Responsibility Assert compatibility Limited versions</i>	<p><i>“Our ranges are most of the time bigger than what we officially support. For the platform we have a minimum requirement of 3.4, but this does not mean that if somebody really still runs with 3.4 that we will commit our valuable time to solve problems. Then, he has to bother himself.”</i></p>

Participant P9

4.2.4 4.4.1 2.4.3 2.4.4	<i>Multiple versions Releases No automated versions Manual versions</i>	<p><i>“To test those versions, we do it manually, but we also have some users that do that with the unstable versions.”</i></p>
4.4.1 4.2.7 4.2.2	<i>Releases Filing bug reports Software usage</i>	<p><i>“We have 3-tiered releases. So, with unstable releases. And some of the experienced users use them, and they can also report bugs for those, and if that all works, than we release them as stable releases.”</i></p>
4.2.2 4.2.1 2.3.5	<i>Software usage Manual testing No automated cross-tests</i>	<p><i>“The tests that I perform are very simple manual tests, the real tests are coming from the users, that are doing all kind of different things with [Project 9]. We are just testing if there is no basic regression.”</i></p>
2.3.1 2.3.3 2.3.7 2.3.6	<i>Plug-in independence Demand driven Combination issues Unpredictable</i>	<p><i>“We do not have problems with plug-ins that work in the same domain. We only had problems with plug-ins that do something else, like Subversion and Mylyn. If you would have [a similar plug-in], it would make sense to test that. But at this point, we do not expect a lot of interaction with other plug-ins.”</i></p>

3.1.1 2.3.7 4.2.5	Responsibility Combination issues Compatibility	<i>"The users also use a number of other plug-ins and we got some reports about problems. The combination of plug-ins does not work. The users filed reports in our issue tracker, but we do not know if the problems are caused by our plug-in or by the others."</i>
2.4.7	Update rarely	<i>"We use a very old version of the main plug-in we depend on. Sometimes we update, but there is always the risk that it will break something and then you have to do extensive [manual] testing."</i>
1.1.1	Issue tracker	<i>"We have even an open repository, we want it to be easy to participate."</i>

Participant P10

2.2.7 1.3.6	GUI based GUI testing	<i>"There are no specific types of tests for [integrating multiple plug-ins], but it is covered by the end user tests, and by the GUI tests, which communicate amongst plug-ins, but the internal coverage is more random."</i>
1.3.1	Test automation	<i>"Two years ago, many manual tests were still being executed, but we already had the requirement to automate. But when tests had to be executed daily we could not do it anymore. There are still manual tests, but not a lot. A lot of effort has been put on automation, because it became an obligation."</i>
1.3.3 1.2.5	Continuous integration Confidence	<i>"Having stability in product quality costs time and money. Continuous integration is needed to reach that. Also, the status of the QA has been increased because of CI, and we can also see that it helps to meet the development goals."</i>
1.2.3	Coverage	<i>"We are just in the process of setting up coverage measurements. We see it's becoming a requirement, but until now management has not explicitly asked for it."</i>
1.3.7	GUI maintainability	<i>"It happens that during product evolution, suddenly something works differently and the tests do not work anymore. [There are] synchronization problems, sometimes the test has not been set-up in a clean way, or timing problems occur. To cope with that takes a lot of time."</i>

Participant P11

4.3.5 4.3.4	Providing patches Plug-in symbiosis	<i>"Yes, for the GEF part, we find and report bugs, and we provide patches. In fact, perhaps it is not our own product, but our product relies on this other product. So it is normal to improve the other parts that we need."</i>
----------------	--	---

4.2.7	Filing bug reports	<i>"The community is involved in development by filing bug reports and feature requests."</i>
4.2.8	Feedback	<i>"Open source is great because we can provide feedback for each other, and help others. I think it is good that the community is involved in the development."</i>
1.3.1	Test automation	<i>"Some tests are difficult to automate - it would be too much effort to write tests or maintain them. A human person is better and faster to test that. So, our QA team tests such parts, something we won't automate. And also new features, in order to have feedback, not only on functionality but also on usability."</i>
1.3.3	Continuous integration	<i>"Yes, we are using continuous integration. We are using Hudson. We have several builds in parallel, for branches, for open source versions, and some commercial versions."</i>
1.2.3	Coverage	<i>"I do not know the exact number for the coverage. We have around 80%. We have a lot of generated code, which we do not exclude. We would have to configure the tool. But in fact we are using it more to see whether some important parts are not tested automatically."</i>

Participant P12

1.3.5	Fault location	<i>"Somewhere it fails. But, most of the time when it fails during integration testing, and I analyze it and understand the problem, then I write a unit test so that I can find the problem faster."</i>
4.2.4	Multiple versions	<i>"It can happen that something breaks because of a new Eclipse Version. Such things come back from the user community quickly."</i>
4.2.4 4.2.3 2.4.8	Multiple versions Operating systems Unfeasible	<i>"Testing is done by the user-community and they are rigorous about it. We have more than 10,000 installations per month. If there is a bug it gets reported immediately. I do not even have a chance to test [all possible combinations]. There are too many operating systems, there are too many Eclipse versions."</i>
1.1.2 4.3.5	Requirement source Providing patches	<i>"The main amount [of requirements] comes from me, because I know the use case from another project, and I know how the library has to work so that you can integrate it in Eclipse, because I have done that already. In addition, there are requirements, and some very good ideas from the community, and also often very nice solutions."</i>
1.3.1 1.2.5	Test automation Confidence	<i>"Test automation is very important, because you can just start working to fix a bug or implement a new feature even if you haven't worked on this piece of code for some time. It's a safeguard that nothing gets broken."</i>

4.4.2	Communication	<i>"We try to keep everything open. There is no one-to-one communication. Everything goes through the forums at Source Forge, so that we share the ideas and also to better document them."</i>
-------	---------------	---

Participant P13

1.2.5	Confidence	<i>"It gives you a certain level of comfort to know that when you make a change and you break something that this would be apparent in your test case."</i>
4.3.2	Release train	<i>"Some testing is performed downstream, when packages of multiple plug-ins are produced. Some packages have plug-ins like Mylyn, [project 13], and a whole ton of other projects. Then, there are people that test whether the packages behave reasonably."</i>
4.3.1	Automated testing	<i>"That is one of the things I totally rely on, e.g., the Web Tools Platform uses [project 13] heavily, and they have extensive JUnit tests, and so I am quite sure that when I break something, somebody downstream will rapidly notice and report the problem."</i>
4.2.1	Manual testing	<i>"If there are problems, people definitely report them, so you do find out about problems."</i>
2.4.6	Assert compatibility	<i>"A lot of people put version ranges in their bundle dependencies, and they say we can run with 3.3 up to version 4.0 of the platform. But I am willing to bet that 99% of the people do not test that their stuff works, they might assert it, but I do not believe that they test."</i>
4.4.2	Communication	<i>"Being part of the release train, there is this requirement to have open plans, and to communicate clearly with your community what changes you plan to make."</i>
4.4.2 1.1.1	Communication Issue tracker	<i>"I try to avoid writing any large planning documents. But, not a single code change goes into the code base without a corresponding Bugzilla ID."</i>
2.2.3	Extension points	<i>"Our test cases make use of extension points, so we end up testing if extension point processing is working correctly."</i>
1.2.4	No coverage	<i>"We don't measure coverage, but we should. Some people have measured it in the past, and tests have been added to improve the test coverage, but this is another one of those things that has not really happened much in recent years because of a lack of time."</i>
1.3.8	GUI non-tested	<i>"We try to make a point of surfacing as little visible stuff in the UI as possible. All our UI testing is essentially ad hoc and manual."</i>

Participant P14

1.3.4 1.2.7	Unit vs integration testing Unit testability	<i>“Try to get to a level that you write unit tests, always, whenever you can. And write your code in such a way that the structure and the classes can be tested with unit tests. And then, at max. you use one integration or PDE test to probe the code. Ultimately, unit tests are our best friends, and everything else is already difficult. ”</i>
2.1.3	PDE as unit test	<i>“We use integration tests to refactor a code passage, or to fix a bug, when you cannot write a unit test. Then, at least you write an integration test that roughly covers the case, to not destroy something big. That, we use a lot.”</i>
2.2.6 2.1.1	Eco-system integration Workbench dependencies	<i>“We have to test the integration of our code and the Eclipse code, and then you automatically have the need for PDE tests. And then, you test in a different way, because you do not have so much interaction between the code you write and the test code, and you have more test requirements. There are more players in the game.”</i>
1.2.9	Fast execution	<i>“With us it is common practice that everyone runs the unit test suite before committing. We have around 3000 unit tests, which take 2 minutes to execute.”</i>
3.1.6	Execution time	<i>“Our integration tests are only executed in the nightly build. I guess they take approximately half an hour to execute.”</i>
3.1.7 2.1.1	Eclipse testability Workbench dependencies	<i>“The problem is that code pieces are strongly interwoven with each other. You can’t just call one piece of code, but you must first, to even be able to call the code, set all pre-requirements and put the system in the right state, and this often means that you have to boot and instantiate the whole system. And at a certain point, writing a unit test does not pay off anymore.”</i>
1.3.7	GUI maintainability	<i>“We had a QF-Test suite, but it became apparent that those are too rigid to use them further [if software evolves]. That’s why we stopped using them.”</i>
1.3.7	GUI maintainability	<i>“We also have a couple of Fit tests. We had good experiences, although we had problems with maintainability. In the end, it was too much trouble compared to the benefits we got.”</i>

1.1.4	Hybrid testing	<i>“Testing is a multi-stage process, because of Scrum. In fact, the development team manually tests it before calling it done and submitting it. Then, the quality representative who is on the same team, takes over, manually tests it again, and determines if it’s ready. If he accepts, then it goes to the “expert users”, where we also sit in and explain the results of the sprint. If there are complex scenarios they might also test it again. Finally, it is applied to the production system.”</i>
4.2.9	Customer involvement	<i>“The “expert users”, they are from the customer and use the software in production. Those are the really expensive people, the people that bring in the money at the customers. They use the system themselves and in real production.”</i>
2.4.5 2.4.7	Limited versions Update rarely	<i>“We always have one specific version for platform and libraries that we use, if we update that, that’s a major effort. That we do only rarely.”</i>

Participant P15

1.2.1 2.2.8	Key practice: Unit testing No eclipse integration	<i>“The majority of the tests are written with JUnit, and the main test suites comprise tests that do not depend on Eclipse.”</i>
1.3.7	GUI maintainability	<i>“In my experience, automating UI testing is very expensive with no big benefits, especially [if you have a lot of] change.”</i>
3.1.5	Set-up build system	<i>“There are a few tests that depend on Eclipse, but these are actually currently not run on a daily basis, i.e., as part of the regular tests. It is a technical problem, just the ability to run them from the command line, i.e., in the same way as the other tests are run. This was impossible until Eclipse 3.6.”</i>
4.2.9 4.2.2	Customer involvement Software usage	<i>“People book time at hatches, to try the software. It is not like they go to test it, but they use it, so they are users, but they might do things that help uncover a couple of bugs and issues.”</i>
4.2.9 4.2.8 4.4.1	Customer involvement Feedback Releases	<i>“A lot of the product is used by the scientists in the complex. The developers have access to the users all the time. So it tends to be: they develop for 4 months, make a release, make it available to the scientists and then fix it as they go along. They are a big source of feedback by saying what improvements they like, and which features they need. So, they give a lot of feedback to the developers.”</i>

1.1.1	<i>Issue tracker</i>	<i>“There is a separate reporting system which the scientists have to submit feature requests, enhancements and bugs. But some of that is also done directly, face to face with some developers that are dedicated to help them.”</i>
-------	----------------------	---

Participant P16

2.2.3 2.2.5 2.2.4	<i>Extension points Plug-in testability Registration untested</i>	<i>“We tried a lot. We test our functionality by covering the functionality of the extension point in a test case, i.e., testing against an API. The small glue code where the registry gets the extension, that’s not tested, because it is just hard to test that. And for these untested glue-code parts we had the most bugs.”</i>
4.3.3	<i>Ecosystem</i>	<i>“I was actually the only one making an effort to integrate our project with the other, like with EMF. To make sure it works fine. I often started the integration process, but now this is really driven by the other projects and the community. Our project has almost nothing to do with it anymore. Also not with the compatibility issues.”</i>
2.3.4 2.3.5 2.3.6 2.4.8	<i>Manual combinations No automated cross-tests Unpredictable Unfeasible</i>	<i>“There are no automated tests for that, because that is quite a complex topic, also because you never know who you will have problems with. That’s why we test that manually a lot.”</i>
3.1.5	<i>Set-up build system</i>	<i>“We had some PDE tests, but they were not developed by us. They came from the library and its test suite, which we took over. But the PDE tests included have never been part of the continuous integration, because it was too much effort to set that up.”</i>
4.4.1 4.2.2 4.4.3	<i>Releases Software usage Test request</i>	<i>“We are explicitly paying attention to regularly making new milestones publicly available. And then, we always write to our community ‘Please test our software’, because we want to get a clean release. That’s why we motivate the people to test their applications with our milestone.”</i>
4.2.4 4.2.2 4.2.7	<i>Multiple versions Software usage Filing bug reports</i>	<i>“It’s great that people not only work on a stable version, but also use a new version, so to say bleeding-edge, and report if something breaks. Then, we can fix that in the next release. Those are typical regressions.”</i>
4.4.2	<i>Communication</i>	<i>“There are things we discuss on the mailing lists because we know others want to take part. Then we make internal discussions public.”</i>

4.4.2	Communication	<i>"If people want to be involved, they are very welcome. Especially before a release, then we always ask what would be interesting or the most important for the community. This information serves as input for the planning. Nevertheless, the planning takes place internally, because we do not want somebody from outside dictating what we have to work on."</i>
4.3.4	Plug-in symbiosis	<i>"I report problems I discover in other Eclipse projects often. And this happens also the other way around."</i>
4.2.7	Filing bug reports	<i>"To just mention some numbers: At the moment, we have 262 open bugs and 2700 closed bugs. Many bug reports came from end-users. So, the community has been very active and most helpful."</i>
2.2.5	Plug-in testability	<i>"With the use of OSGi, the amount of black box testing increases, because the different classloaders prevent you from accessing the code. You can't access it anymore."</i>
1.1.2	Requirement source	<i>"Half of the requirements are determined by the developers, on behalf of the company or customers. So, of course the developers make sure that bugs and features important to the customers are fixed or implemented. The other half of the requirements are community driven. We have an eye on the highest voting."</i>
2.4.5 1.3.3 2.4.1	Limited versions Continuous integration Build system	<i>"The continuous integration server builds [Project 16] on top of two versions of the runtime: the latest stable version, and the current version. Though, we always recommend to keep the stable version, because that's the environment we know it works. And then, we have one version, for us, to try out the latest dependencies and see if everything works fine."</i>

Participant P17

4.2.3 4.2.4	Operating systems Multiple versions	<i>"The community helps to test the system for different operating systems, and versions. They are very active with that."</i>
3.1.6	Execution time	<i>"For this product, we don't have any pure unit tests. We always use PDE tests, which are a lot more complex, and take quite some time to run, because OSGi has to be started and bundles have to be loaded, and so on."</i>
3.1.6	Execution time	<i>"It's a difference between 10 seconds and 1 minute: With 1 minute you switch to Twitter or Facebook, and you're disrupted in your flow."</i>

3.1.6	Execution time	<i>“When I test my stuff, around 1500 SOAP calls are issued, which takes its time. This explains the 15 hours we need for testing – it’s not only due to the slow execution, but also because of the network traffic.”</i>
4.1.1	Self-hosting	<i>“In our company, we have different set-ups, based on Linux or Windows. This leads already to a high coverage because we use our own products on a daily basis. Then you are aware of problems and report that immediately.”</i>
1.3.8	GUI non-tested	<i>“We do not test the GUI. The problem is that all tools are essentially useless.”</i>
3.1.5	Set-up build system	<i>“In addition to the long execution time, it is a hassle to get [the GUI tests] run on the build sever.”</i>
1.3.8	GUI non-tested	<i>“What I prefer to do is factor out the key logic, which I then cover well with unit tests. The glue logic between button and controller is untested. You write it once, and then it does not change anymore.”</i>
4.2.6 4.2.1	GUI community Manual testing	<i>“The community definitely plays a role in GUI testing. I think, there is almost only the community that comes and says: ‘Look there, the button is wrong, and I do not like that.’ For these tasks, the community is very active.”</i>
2.3.1	Plug-in independence	<i>“No, we don’t test plug-in combinations. With Project17, I don’t see it as relevant, since there should be no problems. Things are clearly separated.”</i>
1.1.1 1.1.2 4.2.9	Issue tracker Requirement source Customer involvement	<i>“In my current project, all requirements come directly from the customer. A lot of [team communication] goes via Mylyn task, and we have external Mylyn tasks filed by our customers.”</i>
2.4.2	External systems	<i>“When I integrate external systems, I run my single test suite against all versions of this external system. Using this, I can be sure, for example, that my system works well with all 18 versions of [the product to integrate]. I need this, because I cannot rely on them that their systems work the same tomorrow as they do today.”</i>
2.4.2	External systems	<i>“We test our plug-ins, which are connecting external systems, against many, many versions of the end systems. For the integration of Bugzilla, for example, we support all Bugzilla versions from the last 3 years. That is around 17 releases and we have the tests run against each version.”</i>

Participant P18

1.2.1	Key practice: Unit testing	<i>"At our company, testing is quite standard. We have different stages. We have unit testing, and that's where we put the main effort - at least 70% of the total expenses."</i>
1.2.2	Preference	<i>"Unit testing, that's where you find the most bugs."</i>
2.2.2	Modularization	<i>"I believe OSGi is very helpful for creating clear structures. The test strategy is not changed, but there are structural changes."</i>
1.1.3	Developer testing	<i>"Every developer is also a tester. That's also due to test driven development."</i>
3.1.6	Execution time	<i>"We split the unit tests into fast and slow running tests. Slow running tests consider e.g., time outs. Fast running tests need for every module one second, at a max."</i>
1.1.5 1.1.3	Tester status Developer testing	<i>"As software developers, they feel comfortable writing code, but when they have to write tests, then they do not see that as code. And then, to teach them to develop joy and discipline to write tests, that I find difficult."</i>
2.3.3 2.3.1	Demand driven Plug-in independence	<i>"We handle problems between several plug-ins in a bug-driven way. If there is a bug we write a test, but we do not think ahead which problems could there be. Also, with unit tests, most of the bugs are already caught."</i>
1.3.1	Test automation	<i>"We automate everything. That's our main principle. Tests that are not automated, are not tests for us."</i>
1.3.7	GUI maintainability	<i>"We haven't been 100% satisfied with capture-replay, because too much is captured. After a capture, we always have a review to remove unnecessary code."</i>

Participant P19

4.3.5 4.3.4 4.4.2	Providing patches Plug-in symbiosis Communication	<i>"It is very important for us that our tests are executable by the contributors. This means that all the infrastructure, the repositories, like Bugzilla and TRAC, must be publicly available, so that somebody that writes a patch has the opportunity to run the same tests we execute internally. So, he can run those and check whether the patch is okay. We expect for every contribution, that also the according tests are provided. That's a prerequisite."</i>
-------------------------	---	--

4.4.1	Releases	<i>“We have quite a similar process for the open source and the closed source tools. Quite agile. We release every 3-6 months and we try to keep that synchronized between the open source and the commercial products. Usually, [the open source product] is released first and then the closed one, which is based on it, follows around 1-2 weeks later.”</i>
4.3.5 4.3.4	Providing patches Plug-in symbiosis	<i>“[Who contributes] varies a lot: end-users, but also our partners, or people that integrate their systems with ours. That’s always different.”</i>
4.4.1 4.2.8 4.2.2	Releases Feedback Software usage	<i>“We release every 3 to 6 months, but we also have periodic weekly builds, and an early access version based on the weekly build, in order to get early user feedback.”</i>
2.2.5	Plug-in testability	<i>“Testing is more difficult, especially because of the separate classloaders. That makes it complicated to access the internals. Therefore some methods which should be protected are public to enable testing.”</i>
4.2.5 2.3.4 2.3.5	Compatibility Manual combinations No automated cross-tests	<i>“we have no automated tests for cross-product problems, but we do manual testing. Then, we install [product 19] with the SpringSource Tool suite, or with other products, like the IBM rational team concert, or with other distributions, like MyEclipse, to test for interoperability. The user community plays an important role in testing for interoperability.”</i>
4.2.7 4.3.5 4.2.8	Filing bug reports Providing patches Feedback	<i>“I would say the majority of the bug reports come from the community. We have accepted more than 800 patches during the life span of this project. 1/7 of all bugs that have been resolved have been resolved through community contributions. That’s quite a high rate. If there are many votes on a bug, then the bug gets a higher priority. If there are many comments on a bug, then we know this is a critical bug, and we take the community feedback serious.”</i>
4.2.4 4.2.3 4.2.5	Multiple versions Operating systems Compatibility	<i>“But the actual testing is really performed by users that run maybe 64 Windows, or Ubuntu, and they use a different version, or wired syntax-things etc. The feedback of the user is very valuable for the quality of the system.”</i>
2.2.3	Extension points	<i>“In some cases, we have extensions just for testing in the test plug-ins. Either the extensions are just loaded or they implement some test behavior.”</i>
2.2.3	Extension points	<i>“We recommend that if somebody writes an extension, they should look at the relevant tests, because those tests demonstrate how to use the API.”</i>

1.1.3	Developer testing	<i>“We have no dedicated test team. It would be great [to have a separate test team], because then the developer can better concentrate on the development. On the other hand, I think it’s important as a developer to feel responsible for the quality, and that you learn how you can better test your software.”</i>
4.4.2 4.4.4 1.1.1	Communication Opening closed software Issue tracker	<i>“We attempt to communicate more openly for the commercial products. We have a public issue tracker. But we do not include all our tasks as we do with the open source projects. The commercial side is a bit different. This means the majority of issues are support requests, but partially you see which features we’re working on. Often a commercial request also causes a change in the [open source parts], and then we attempt to make that transparent, as much as possible.”</i>
4.3.5 4.3.4	Providing patches Plug-in symbiosis	<i>“People from the outside contribute by providing patches or feature enhancements, in case some API is missing, or the API is not working as expected. Also, when we find bugs in the platform, we always try to provide the required patch. If possible.”</i>

Participant P20

1.2.7 1.2.8	Unit testability Unit non-tested	<i>“We try to encapsulate the logic as much as possible to be able to test with unit tests. What cannot be encapsulated is not tested.”</i>
3.1.6 1.2.9	Execution time Fast execution	<i>“We have 7000-8000 normal JUnit tests, which run within 2 seconds. Running the same within the PDE runner takes 1.5–2 minutes. Thinking of the ‘red, green, refactor’ paradigm, all tests must be executed at least 3 times. With PDE this becomes a problem.”</i>
4.3.3 2.3.7	Ecosystem Combination issues	<i>“We actively try to establish collaborations with other committers to ensure that plug-ins work together, but there are still things that do not work together.”</i>
1.3.4 1.2.3 1.2.1	Unit vs integration testing Coverage Key practice: Unit testing	<i>“We think that with a high test coverage through unit tests, integration tests are not necessary.”</i>
1.3.4	Unit vs integration testing	<i>“Personally, I would like to see integration testing done in our project. Otherwise, you do not know whether two parts can work together. But my team members think differently.”</i>
4.3.1 4.3.4	Automated testing Plug-in symbiosis	<i>“We are in the special position of being a framework. This means that if a user downloads a new version and runs his application based on ours, then this is already like a test.”</i>

Participant P21

1.1.4	Hybrid testing	<i>“Automated tests are only created by developers. Manual testing is done partly by developers. Regression testing is done by someone from the customer.”</i>
2.1.1 2.1.3	Workbench dependencies PDE as unit test	<i>“Our PDE tests do not really look at the integration of two components. There are often cases where you actually want to write a unit test, but then it’s hard to write, because the class uses something from the workbench. Then, it does not work anymore. So, that’s why those tests are not classic integration tests but, from my point of view, more unit tests that unfortunately need the platform, and that’s why they are PDE tests.”</i>
3.1.5	Set-up build system	<i>“It is already quite complicated to automate the JUnit stuff to run with the build and make sure that reporting is working. And then another framework, I honestly did not want to take the trouble.”</i>
1.2.9	Fast execution	<i>“The normal unit tests run in seconds. We had several thousands, but normally they are incredibly fast. And that, you could easily execute during development.”</i>
3.1.6 1.2.9 2.1.1	Execution time Fast execution Workbench dependencies	<i>“Our practice is to use normal unit tests every time we can, because they are much faster executed. We use PDE tests only if we really need the runtime.”</i>
3.1.6	Execution time	<i>“We have two test suites: one runs with the plain JUnit test runner, and the other runs with the PDE test runner. The split is important, because we have so many tests and some involve the UI. During development you can not run the PDE tests, because they take half an hour to execute.”</i>
2.1.2 3.1.8	PDE as integration test PDE integration tooling	<i>“We use the PDE JUnit framework to write integration tests, although we are not happy with it. It’s not really suited for that.”</i>
1.3.7	GUI maintainability	<i>“We put an immense effort into writing UI tests, and in the end often more test code existed than code to test. I doubt that makes sense.”</i>
1.3.8	GUI non-tested	<i>“In the current project, we completely abandoned automatic UI testing.”</i>

Participant P22

2.4.1 1.3.3	Build system Continuous integration	<i>“We have several builds to support different Eclipse versions. We set-up different target platforms to support Eclipse 3.4, 3.5 and 3.6.”</i>
----------------	--	--

2.4.1	Build system	<i>“Most tests must pass on all supported Eclipse platforms, with some exceptions, like tests that have to do with the P2 provisioning system.”</i>
3.1.5 3.1.4	Set-up build system Plug-in testing knowledge	<i>“When we set-up the build for Eclipse 3.4 and 3.5, that was clearly a huge effort, until we knew how to do it. Then, for Eclipse 3.6 the set-up was okay.”</i>
3.1.5	Set-up build system	<i>“Running OSGi in different runtime environments is a complex story which makes it hard to automate tests. I repeatedly hit a wall trying to start OSGi in another runtime in such a way that I can execute the integration tests.”</i>
3.1.5	Set-up build system	<i>“Setting up integration tests with all requirements can be so complex that I regularly experience people saying ‘that’s not worth the effort’.”</i>
3.1.4	Plug-in testing knowledge	<i>“At the customer site, I am often the expert for Eclipse, and the rest knows little about it. This is visible in the testing practice, because often they don’t know how to write tests that have to do with Eclipse code, or how to execute them.”</i>
2.2.8 1.2.1 3.1.6	No eclipse integration Key practice: Unit testing Execution time	<i>“We try to keep away from Eclipse when writing tests. So practice is to have unit tests, which are the majority of tests. Those are plain Java tests and they run with JUnit, but do not need Eclipse or the OSGi runtime. I just do not want to start Eclipse or OSGi when I quickly want to run the tests.”</i>
2.1.1	Workbench dependencies	<i>“Often plain unit tests are not sufficient. Therefore, we also have test suites that start the runtime, such as the Eclipse workbench or OSGi.”</i>
1.3.4	Unit vs integration testing	<i>“We try to have only few integration tests.”</i>
1.3.4	Unit vs integration testing	<i>“We have two test suites: one with JUnit tests and one with PDE tests.”</i>
2.2.3	Extension points	<i>“We have dedicated test extensions. A test checks whether the test extension is loaded and executed.”</i>

Participant P23

3.1.7 3.1.4	Eclipse testability Plug-in testing knowledge	<i>“OSGi wires bundles at runtime, this means a lot of magic is taking place in the XML configuration files, and to test those, this is complicated with traditional test practices.”</i>
3.1.7	Eclipse testability	<i>“We all know that SWT Widgets are hard to mock. And you only find out about problems at runtime. To overcome that we use JUnit and JMock, and the PDE test runner.”</i>
1.3.6	GUI testing	<i>“We have PDE tests for testing the UI, but we try to have a minimum of logic there.”</i>
1.3.7	GUI maintainability	<i>“Functional tests can become a maintenance nightmare. So, we try to keep them focused, because it can become too much for a team to maintain.”</i>

Participant P24

1.3.1	<i>Test automation</i>	<i>“There has to be a commitment from the customer to do testing. If the benefits are not visible to the customer they see it as a waste of time.”</i>
1.2.6	<i>Limited confidence</i>	<i>“We had a problem with some software we developed. When more than 20 users used the software in parallel it crashed. At the same time, many tests existed for this product which passed.”</i>

Participant P25

2.4.5	<i>Limited versions</i>	<i>“We have trust that the Eclipse core is well tested. So, we have exactly one version and we do not use version ranges or variation.”</i>
1.2.1	<i>Key practice: Unit testing</i>	<i>“We have a lot of unit tests.”</i>
1.3.1	<i>Test automation</i>	<i>“Integration testing is done only manually, because it is too complex to automate.”</i>
1.3.3	<i>Continuous integration</i>	<i>“Our unit tests run all the time. We have continuous integration.”</i>

Index of Codes to Participants

2.4.6	Assert compatibility, P8, P13	3.1.7	Eclipse testability, P1, P5–7, P14, P23
4.3.1	Automated testing, P13, P20	2.2.6	Eco-system integration, P14
2.4.1	Build system, P16, P22	4.3.3	Ecosystem, P16, P20
2.3.7	Combination issues, P6, P9, P20	3.1.2	End user requirements, P7
4.4.2	Communication, P4, P7, 8, P12, 13, P16, P19	3.1.6	Execution time, P1, P5, 6, P14, P17, 18, P20–22
4.2.5	Compatibility, P7, P9, P19	2.2.3	Extension points, P13, P16, P19, P22
1.2.5	Confidence, P1, P3, P10, P12, 13	2.4.2	External systems, P17
1.3.3	Continuous integration, P3, P5, 6, P10, 11, P16, P22, P25	1.2.9	Fast execution, P1, P6, P14, P20, 21
1.2.3	Coverage, P5, P7, P10, 11, P20	1.3.5	Fault location, P6, P12
4.2.9	Customer involvement, P3, P5, P14, 15, P17	4.2.8	Feedback, P2, P6, P11, P15, P19
2.3.3	Demand driven, P9, P18	4.2.7	Filing bug reports, P9, P11, P16, P19
1.1.3	Developer testing, P5, P18, 19		

- 2.2.7 GUI based, P3, P10
- 4.2.6 GUI community, P17
- 1.3.7 GUI maintainability, P7, P10, P14, 15, P18, P21, P23
- 1.3.8 GUI non-tested, P1, 2, P13, P17, P21
- 1.3.6 GUI testing, P3, P5, P10, P23
- 1.3.2 Hardware integration, P5
- 2.1.4 Headless PDE, P6
- 1.1.4 Hybrid testing, P3, P14, P21
- 1.1.1 Issue tracker, P9, P13, P15, P17, P19
- 1.2.1 Key practice: Unit testing, P7, P15, P18, P20, P22, P25
- 1.2.6 Limited confidence, P3, P5, P7, P24
- 2.4.5 Limited versions, P8, P14, P16, P25
- 2.3.4 Manual combinations, P16, P19
- 4.2.1 Manual testing, P6, 7, P9, P13, P17
- 2.4.4 Manual versions, P7, P9
- 2.2.2 Modularization, P3, P5, P18
- 4.2.4 Multiple versions, P8, 9, P12, P16, 17, P19
- 2.3.5 No automated cross-tests, P9, P16, P19
- 2.4.3 No automated versions, P7, P9
- 1.2.4 No coverage, P3, P13
- 2.2.8 No eclipse integration, P15, P22
- 2.2.1 No influence, P2
- 4.4.4 Opening closed software, P19
- 4.2.3 Operating systems, P12, P17, P19
- 3.1.3 Ownership, P6, 7
- 2.1.2 PDE as integration test, P6, P21
- 2.1.3 PDE as unit test, P1, P6, P14, P21
- 3.1.8 PDE integration tooling, P21
- 2.3.2 Play nicely, P6
- 2.3.1 Plug-in independence, P9, P17, 18
- 4.3.4 Plug-in symbiosis, P11, P16, P19, 20
- 2.2.5 Plug-in testability, P5, P16, P19
- 3.1.4 Plug-in testing knowledge, P4, P22, 23
- 1.2.2 Preference, P1, P6, P18
- 4.3.5 Providing patches, P6, P11, 12, P19
- 2.2.4 Registration untested, P16
- 4.3.2 Release train, P8, P13
- 4.4.1 Releases, P6, 7, P9, P15, 16, P19
- 1.1.2 Requirement source, P2, 3, P5, 6, P12, P16, 17

- 3.1.1
 - Responsibility, P8, 9
- 4.1.1
 - Self-hosting, P17
- 3.1.5
 - Set-up build system, P4, P15–17, P21, 22
- 4.2.2
 - Software usage, P9, P15, 16, P19
- 1.3.1
 - Test automation, P3, P5, P10–12, P18, P24, 25
- 4.4.3
 - Test request, P16
- 1.1.5
 - Tester status, P3, P18
- 2.4.8
 - Unfeasible, P12, P16
- 1.2.8
 - Unit non-tested, P20
- 1.2.7
 - Unit testability, P14, P20
- 1.3.4
 - Unit vs integration testing, P1, P6, P14, P20, P22
- 2.3.6
 - Unpredictable, P9, P16
- 2.4.7
 - Update rarely, P9, P14
- 2.1.1
 - Workbench dependencies, P1, P6, P14, P21, 22

Zusammenfassung

Unterstützung von Softwareentwicklern beim Verstehen von Softwaretests

Software ist allgegenwärtig und ein Leben ohne sie ist nicht mehr vorstellbar. Neben alltäglichen Verwaltungsprogrammen gibt es auch Software die es uns ermöglicht zu fliegen, Krankheiten zu diagnostizieren und Operationen sowie Banktransaktionen durchzuführen. Durch den umfassenden Einsatz von Software sind auch die Konsequenzen von fehlerhafter Software weitreichend.

Eine wichtige Maßnahme, um Fehler in Software zu erkennen und die Qualität einer Software zu bestimmen, ist das Softwaretesten. Hierbei wird die Software ausgeführt und die Ergebnisse dieser Ausführung werden mit den Erwartungen verglichen. Stimmen Erwartung und Ausführungsergebnisse nicht überein, wurde ein Fehler identifiziert. Das manuelle Ausführen von Testaktivitäten durch einen Menschen, zum Beispiel das Durchspielen aller möglichen Szenarien einer Software, ist mühsam, fehlerreich und zeitintensiv. Deswegen wird dieser Prozess heutzutage immer öfter automatisiert. Hierbei wird wiederum Software erzeugt, auch Testsystem genannt, deren Aufgabe die Überprüfung der Korrektheit der eigentlichen Software ist.

Im Laufe der letzten Jahre wurden nicht nur Softwaresysteme komplexer, sondern auch deren Testsysteme. Ein Großteil des Softwareentwicklungsaufwandes wird für das Testen der Software benötigt, und ein beträchtlicher Teil des Softwaresystems ist das Testsystem. Die meisten heutigen Softwaresysteme haben Testsysteme mit mehreren Millionen Zeilen von Testcode. Diese Menge an Testcode kann sehr schnell unübersichtlich und unverständlich werden, wodurch das Verstehen und somit auch die Wartung dieser Testsysteme einen erheblichen Aufwand verursachen. Speziell modulare und dynamische Systeme erschweren das Softwaretesten, da mehrere unterschiedliche Parteien (z.B.: Firmen oder Projektteams) an deren Entwicklung beteiligt sind. Dennoch sind diese Systeme heute weitverbreitet, da sie den Vorteil haben, auch noch nach Inbetriebnahme einfach, zum Beispiel vom Endanwender, verändert und erweitert werden zu können. Beispiele solcher Systeme sind der Internet Explorer oder der Mozilla Browser, aber

auch Programme wie Microsoft Word erlauben Benutzern die Erweiterung des Basissystems.

Um die Arbeitsweisen und die Probleme von Entwicklern während des Softwaretesten von solchen Systemen genau zu erfassen, haben wir eine gründliche Untersuchung durchgeführt, in welcher wir professionelle Softwareentwickler von mehr als 20 verschiedenen Firmen befragten. Diese Studie zeigte, dass speziell das Integrationstesten, also das Testen, ob mehrere Komponenten miteinander korrekt funktionieren, in modularen und dynamischen Systemen schwierig und zeitaufwendig ist. Deshalb wird diese Aufgabe zum Teil auf die Softwarebenutzer ausgelagert. Die Interviews zeigten außerdem, dass Softwareentwickler Schwierigkeiten haben, einen Überblick über alle Softwaretests zu erlangen oder beizubehalten, relevante Tests oder nicht getestete Funktionalität zu identifizieren, sowie Tests zu warten. Daraufhin haben wir es uns zum Ziel gesetzt, unterstützende Techniken und Werkzeuge für Softwareentwickler zu entwerfen.

In dieser Doktorarbeit entwickelten wir vier Techniken, die durch Abstraktion verschiedene Sichten auf komplexe Systeme und deren Testsystemen ermöglichen, welche relevante Informationen hervorheben und irrelevante Information ausblenden. Die entwickelten Techniken gehören der Familie der Reverse Engineering Techniken an, welche es erlauben, aus einem bestehenden, fertigen System durch Untersuchung der Strukturen, Zustände und Verhaltensweisen, die Konstruktionselemente zu extrahieren. Um die Techniken umfassend zu evaluieren und deren Anwendbarkeit, Genauigkeit und Skalierbarkeit zu erfassen, haben wir diese Techniken in Softwarewerkzeuge implementiert und mit deren Hilfe verschiedene industrielle Systeme analysiert. Außerdem haben wir durch Benutzerstudien die Meinung von Softwareentwicklern über die Anwendbarkeit und Nützlichkeit dieser Werkzeuge erfasst. Wir haben uns einer breiten Palette von Untersuchungsmethoden bedient, um unsere Techniken und Werkzeuge empirisch zu validieren. Interviews, Umfragen, Grounded Theory, Fallstudien und Data-Mining gehören zum Methodensortiment. In den meisten Studien kommt eine Kombination mehrerer Untersuchungsmethoden zum Einsatz (auch bekannt als Mixed-Method-Vorgehensweise), da diese eine Triangulation der Ergebnisse ermöglicht, d.h. Schwächen einer Vorgehensweise können mit Stärken der jeweils anderen ausgeglichen werden. Ein besonderes Augenmerk haben wir auch auf das Miteinbeziehen von professionellen Softwareentwicklern gelegt, sowie auf die Analyse von industriell eingesetzten Softwaresystemen.

In dieser Arbeit zeigen wir die Praktiken und Probleme beim Testen von modularen und dynamischen Systemen auf, und stellen vier Techniken und Werkzeuge zur Unterstützung von Softwareentwicklern vor. Unter anderem belegen die durchgeführten Studien, dass die entwickelten Werkzeuge hilfreich sind, (1) um Entwickler während des Integrationstestens und des Verstehens und Wartens von Tests zu unterstützen, (2) Entwicklern, die nicht vertraut mit dem Testsystem sind, den Einstieg in die Arbeit zu erleichtern, und (3) Schwachstellen in Testsystemen zu erkennen und zu beheben. Ein verbessertes Verständnis von Tests führt zu besseren Testsystemen, welche wiederum zu besserer und fehlerfreierer Software führen.

Samenvatting

Het ondersteunen van ontwikkelaars bij het begrijpen van softwaretests

Software is alomtegenwoordig, en een leven zonder software is niet meer voorstelbaar. Naast alledaagse administratieve programma's, bestaat er programmatuur die ons in staat stelt te vliegen, ziektes te diagnosticeren, en operaties zoals bancaire transacties door te voeren. Door deze brede inzet van software zijn echter ook de gevolgen van foutieve software vergaand.

Een belangrijke maatregel om fouten in software op te sporen en de kwaliteit van software te bepalen is het testen van software. Hierbij wordt software uitgevoerd, en worden de uitkomsten ervan met verwachtingen vergeleken. Komen de verwachting en de uitkomsten niet overeen, dan is er een fout gevonden. Het handmatig uitvoeren van testactiviteiten door de mens, bijvoorbeeld het aflopen van alle mogelijke scenario's van een programma, is moeizaam, foutgevoelig, en tijdrovend. Daarom wordt dit proces tegenwoordig steeds vaker geautomatiseerd. Hierbij wordt een tweede programma ontwikkeld (het zogenaamde testsysteem), dat als doel heeft correctheid van het eigenlijke softwaresysteem vast te stellen.

In de loop der jaren zijn niet alleen softwaresystemen zelf complexer geworden, maar ook de bijbehorende testsystemen. Een belangrijk deel van de inspanning van softwareontwikkeling komt voor rekening van het testen, en een aanzienlijk onderdeel van het softwaresysteem is het testsysteem. De huidige software systemen hebben doorgaans testsystemen van meerdere miljoenen regels testcode. Deze hoeveelheid testcode kan snel onoverzichtelijk en ondoorgrondelijk worden, waardoor het begrijpen van en daarmee ook het onderhoud aan deze systemen een aanzienlijke inspanning vergen. Met name modulaire en dynamische systemen bemoeilijken het testen, omdat meerdere partijen (bijvoorbeeld bedrijven of projectteams) bij de ontwikkeling betrokken zijn. Toch zijn deze systemen tegenwoordig wijdverbreid, omdat zij als voordeel hebben dat zij ook na ingebruikname gemakkelijk, bijvoorbeeld door een eindgebruiker, aangepast en uitgebreid kunnen worden. Voorbeelden van dergelijke systemen zijn browsers zoals Internet Explorer of Mozilla, of programma's zoals Microsoft Word die het gebruikers mogelijk

maken het basissysteem uit te breiden.

Om de werkwijze en problemen van ontwikkelaars tijdens het testen van dergelijke systemen precies te doorgronden, hebben we een uitgebreid onderzoek uitgevoerd, waarin we professionele softwareontwikkelaars van meer dan 20 verschillende bedrijven hebben ondervraagd. Deze studie toonde aan dat met name integratietesten (d.w.z. het testen of meerdere componenten correct samenwerken), in modulaire en dynamische systemen moeilijk en tijdrovend is. Om die reden wordt deze taak gedeeltelijk doorgeschoven naar de softwaregebruikers. De interviews lieten bovendien zien dat softwareontwikkelaars moeite hebben om het overzicht over alle softwaretests te verkrijgen en te behouden, om relevante tests of niet geteste functionaliteit te identificeren, alsmede om tests te onderhouden. Om dit te adresseren hebben wij ons ten doel gesteld ondersteunende technieken en gereedschappen voor softwareontwikkelaars te ontwerpen.

In dit proefschrift hebben we vier technieken ontwikkeld, die via abstractie verschillende perspectieven op complexe systemen en hun testsystemen bieden. Middels deze perspectieven kan relevante informatie benadrukt en irrelevante informatie verborgen worden. De ontwikkelde technieken vallen in de categorie “reverse engineering”, waarmee het mogelijk is om uit een bestaand, afgerond systeem middels een analyse van structuren, toestanden en gedrag, bouwelementen van het systeem te extraheren. Ten einde deze technieken grondig te evalueren, en hun toepasbaarheid, nauwkeurigheid, en schaalbaarheid te toetsen, hebben wij deze technieken in softwaregereedschap geïmplementeerd, en daarmee diverse industriële systemen geanalyseerd. Bovendien hebben we door gebruikersstudies het oordeel van ontwikkelaars over de toepasbaarheid en het nut van dit gereedschap getoetst.

Wij hebben een breed scala aan onderzoeksmethoden toegepast om onze technieken en ons gereedschap empirisch te valideren. Interviews, vragenlijsten, Grounded Theory, case studies, en data mining horen tot de ons assortiment van methoden. In de meeste van onze studies passen we een combinatie van verschillende onderzoeksmethoden toe (ook bekend als de “mixed-methods”-aanpak), omdat dit triangulatie van de resultaten mogelijk maakt, d.w.z. dat de beperkingen van de ene methode verholpen kunnen worden door de sterke punten van de andere. Hierbij hebben we in het bijzonder aandacht besteed aan het betrekken van professionele softwareontwikkelaars in ons onderzoek, alsmede aan de analyse van in de industrie toegepaste systemen.

In dit proefschrift maken we de praktijk van het testen van modulaire en dynamische systemen inzichtelijk, en laten we zien wat de bijbehorende problemen zijn. Om deze te adresseren presenteren we vier technieken en bijbehorend gereedschap ter ondersteuning van softwareontwikkelaars. De uitgevoerde studies laten onder meer zien dat het ontwikkelde gereedschap (1) ontwikkelaars ondersteunt tijdens het integratietesten en tijdens het doorgronden en onderhouden van tests; (2) het ontwikkelaars die niet vertrouwd zijn met het testsysteem makkelijker maakt met een systeem aan de slag te gaan; en (3) helpt bij het identificeren en oplossen van zwakke plekken in testsystemen. Een beter begrip van tests leidt tot betere testsystemen, wat vervolgens leidt tot betere software met minder fouten.



Curriculum Vitae

Michaela Simona Greiler was born on August 30th 1983 in Klagenfurt, Austria.

EDUCATION

December 2008 – April 2013

PhD student at Delft University of Technology, the Netherlands, in the Software Engineering Research Group under supervision of Prof. dr. Arie van Deursen.

April 2012 – August 2012

Visiting Researcher at the University of Victoria, Canada, in the Computer Human Interaction & Software Engineering Lab - CHISEL group under the supervision of Prof. dr. Margaret-Anne Storey.

October 2006 – March 2008

Master of Computer Science at the Alpen-Adria University of Klagenfurt, Austria. Graduated with honours. Thesis: “Secure Resource Sharing in ad hoc Networks”.

August 2007 – January 2008

Semester abroad at the University of Westminster, London, United Kingdom, in the Grid Computing Research Group.

October 2002 – October 2006

Bachelor of Computer Science at the Alpen-Adria University of Klagenfurt, Austria. Focus on software usability, interactive systems, and economics.

1997 – 2002

BR Gymnasium Viktring, Austria. Focus on art. Graduated with distinction.

1994 – 1997

BR Gymnasium Lerchenfeld, Austria. Focus on sport.

WORK EXPERIENCE

December 2008 – April 2013

Researcher in the Software Engineering Research Group, Delft University of Technology, The Netherlands.

April 2008 – February 2009

Lecturer and researcher in the System Security Research Group, Alpen-Adria University of Klagenfurt, Austria.

February 2007 – October 2007

Software Engineer at topmind Web Development, Klagenfurt, Austria.

July 2006 – January 2007

Internship at Flextronics International, Althofen, Austria.

October 2004 – September 2007

Information Technologies Coach at Volkshochschule Völkermarkt, Austria.

January 2004 – June 2006

Software Engineer at topmind Web Development, Klagenfurt, Austria.

Titles in the IPA Dissertation Series since 2007

- H.A. de Jong.** *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01
- N.K. Kavaldjiev.** *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02
- M. van Veelen.** *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03
- T.D. Vu.** *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04
- L. Brandán Briones.** *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05
- I. Loeb.** *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06
- M.W.A. Streppel.** *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07
- N. Trčka.** *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08
- R. Brinkman.** *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09
- A. van Weelden.** *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10
- J.A.R. Noppen.** *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11
- R. Boumen.** *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12
- A.J. Wijs.** *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13
- C.F.J. Lange.** *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14
- T. van der Storm.** *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15
- B.S. Graaf.** *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16
- A.H.J. Mathijssen.** *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17
- D. Jarnikov.** *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18
- M. A. Abam.** *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19
- W. Pieters.** *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01
- A.L. de Groot.** *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02
- M. Bruntink.** *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03
- A.M. Marin.** *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04
- N.C.W.M. Braspenning.** *Model-based Integration and Testing of High-tech Multidisciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05
- M. Bravenboer.** *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation*

of Language Conglomerates. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for*

Clean. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of

Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

- J.F.J. Laros.** *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27
- C.J. Boogerd.** *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01
- M.R. Neuhäuser.** *Model Checking Non-deterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02
- J. Endrullis.** *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03
- T. Staijen.** *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04
- Y. Wang.** *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05
- J.K. Berendsen.** *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06
- A. Nugroho.** *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07
- A. Silva.** *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08
- J.S. de Bruin.** *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09
- D. Costa.** *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10
- M.M. Jaghoori.** *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11
- R. Bakhshi.** *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01
- B.J. Arnoldus.** *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02
- E. Zambon.** *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03
- L. Astefanoaei.** *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04
- J. Proença.** *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05
- A. Morali.** *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06
- M. van der Bijl.** *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07
- C. Krause.** *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08
- M.E. Andrés.** *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09
- M. Atif.** *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10
- P.J.A. van Tilburg.** *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11
- Z. Protic.** *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12
- S. Georgievska.** *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

- S. Malakuti.** *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14
- M. Raffelsieper.** *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15
- C.P. Tsirogiannis.** *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16
- Y.-J. Moon.** *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17
- R. Middelkoop.** *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18
- M.F. van Amstel.** *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19
- A.N. Tamalet.** *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20
- H.J.S. Basten.** *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21
- M. Izadi.** *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22
- L.C.L. Kats.** *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23
- S. Kemper.** *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24
- J. Wang.** *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25
- A. Khosravi.** *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01
- A. Middelkoop.** *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02
- Z. Hemel.** *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03
- T. Dimkov.** *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04
- S. Sedghi.** *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05
- F. Heidarian Dehkordi.** *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06
- K. Verbeek.** *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07
- D.E. Nadales Agut.** *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08
- H. Rahmani.** *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09
- S.D. Vermolen.** *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10
- L.J.P. Engelen.** *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11
- F.P.M. Stappers.** *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12
- W. Heijstek.** *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13
- C. Kop.** *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

