

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

Michaela Greiler, Arie van Deursen
and Margaret-Anne Storey

Report TUD-SERG-2011-010

TUD-SERG-2011-010

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:

<http://www.se.ewi.tudelft.nl/techreports/>

For more information about the Software Engineering Research Group:

<http://www.se.ewi.tudelft.nl/>

© copyright 2011, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

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

Michaela Greiler, Arie van Deursen
Delft University of Technology
{m.s.greiler||arie.vanDeursen}@tudelft.nl

Margaret-Anne Storey
University of Victoria, BC, Canada
mstorey@uvic.ca

Abstract—Testing plug-in-based systems is challenging due to complex interactions among many different plug-ins, and variations in version and configuration. 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.

Keywords—Eclipse; grounded theory; plug-in architectures; open source software development

I. INTRODUCTION

Plug-in architectures permit the composition of a wide variety of tailored products by combining, configuring, and extending a set of plug-ins [4], [14]. Many successful plug-in architectures are emerging, such as Mozilla’s Add-on infrastructure¹ used in the Firefox browser, Apache’s Maven build manager,² the WordPress extension mechanism,³ and the Eclipse⁴ plug-in platform.

Testing component-based systems in general [19], [22], [27], 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?

¹<https://developer.mozilla.org/en-US/addons>

²<http://maven.apache.org>

³<http://wordpress.org/extend/plugins>

⁴<http://www.eclipse.org>

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 [23]. 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 [6], [10].

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* [1], [5] 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 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 II, we sketch the challenges involved in plug-in testing. Then, in Section III, we layout the experimental design and the steps we conducted as part of our study. In Sections IV–VII we present the key findings of our study, including the test

practices used, the barriers faced, and the compensation strategies adopted. In Sections VIII–IX, we reflect on our findings, addressing implications as well as limitations of our research. We conclude with a survey of related work (Section X), and a summary of our key findings (Section XI).

II. PLUG-IN SYSTEMS: CAPABILITIES AND CHALLENGES

Plug-in-based systems rely on plug-in components to extend a base system [14], [23], [24]. As argued by Marquardt [14], 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 [19], [22], [27]. 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.*”⁵ There are many resources on the Internet stating incompatible plug-in combinations.⁶ Still, incompatibility of plug-in combinations is an open issue.⁷

These same challenges also occur with Eclipse where combinations of plug-ins or versions can be incompatible.⁸ 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.*”⁹ 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*”¹⁰ commonly appear.

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

III. 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.

A. The Eclipse Plug-In Architecture

As the subject of our study, we selected the Eclipse plug-in framework¹¹ along with its community of practitioners. We selected Eclipse for a number of reasons.

First, Eclipse provides a sophisticated plug-in mechanism based on OSGi¹² and to that is enhanced with the Eclipse-specific extension mechanism. It is used to build a large variety of different applications,¹³ 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 [10]) and books emphasizing the test-driven development of plug-ins [6]. 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 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.

B. 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?

⁵<http://wordpress.org/news/2009/10/plugin-compatibility-beta>

⁶For example, plug-ins incompatible with Onswipe <http://wordpress.org/support/topic/plugin-onswipe-list-of-incompatible-plugins-so-far>

⁷<http://www.wpmos.com/wordpress-plugin-compatibility-procedure>

⁸To mention only a few bugs on Bugzilla: 355759, 292783, 196164

⁹Bug Identifier: 268207

¹⁰<http://sourceforge.net/apps/mediawiki/qcmlyln>

¹¹<http://www.eclipse.org>

¹²<http://www.osgi.org>

¹³http://en.wikipedia.org/wiki/List_of_Eclipse-based_software

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

C. 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 [11, 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 [8], but increasingly popular in software engineering research [1]. 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.

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,¹⁴ 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 [11].

D. Participant Selection

For the interviews, we carefully selected knowledgeable professionals who could provide relevant information on

Table I
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

testing practices. We contacted them by participating in Eclipse conferences and workshops, through blogging, and via Twitter. 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 [11, 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 I.¹⁵

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.

E. 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 technical report [11], 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.

¹⁴<http://www.eclipsecon.org/2011/sessions/?page=sessions&id=2207>

¹⁵Please note that for reasons of confidentiality not all companies and projects participating at the interviews are listed.

IV. 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.

A. 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.

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.

B. 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%).

C. 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.”*

D. 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 V, 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 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.

V. 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.

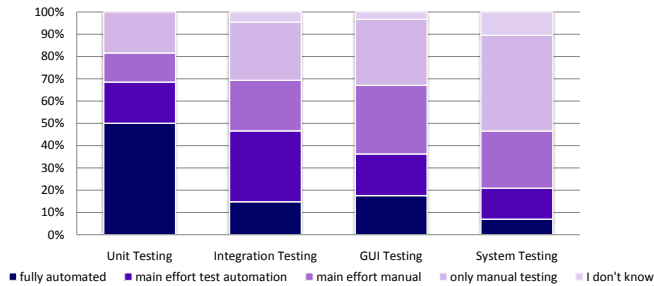


Figure 1. Test automation for each test practice

A. 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?

B. 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.

C. 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”*¹⁶ 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.¹⁷

¹⁶<http://eclipse.org/indigo/planning/EclipseSimultaneousRelease.php>

¹⁷Bug Identifier: 280598 and 213988

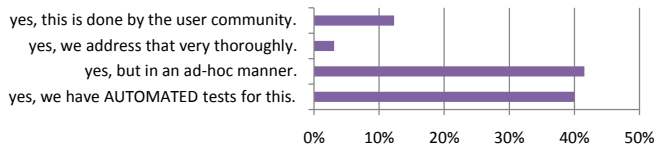


Figure 2. Cross-Product Integration Testing

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 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 VII.

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).

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?

D. 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?

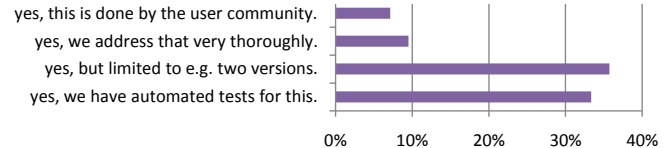


Figure 3. Testing versions of plug-in dependencies.

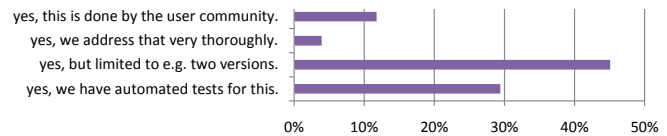


Figure 4. Testing Eclipse platform versions.

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 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 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 4).

VI. 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 [8].

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 [6], 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).

VII. 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*.

A. 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 [12], and recognized for helping in managing and testing configurations of plug-ins, including combinations and different versions.¹⁸

B. 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*

¹⁸<http://dev.eclipse.org/newslists/news.eclipse.platform/msg24424.html>

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.”

C. 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 train¹⁹ 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.”

D. 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 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.”

¹⁹http://wiki.eclipse.org/Indigo/Simultaneous_Release_Plan

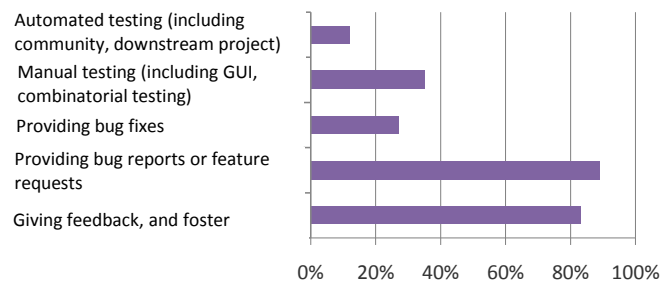


Figure 5. User involvement during testing

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 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.

VIII. 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.

A. 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*,²⁰ 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

²⁰E.g., WordPress introduced a crowd-sourced “compatibility checker” plug-in for their plug-in directory <http://wordpress.org/news/2009/10/plugin-compatibility-beta>

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 believe 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 [10], 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.* for component-based systems [15].

B. 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).

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.

IX. CREDIBILITY AND LIMITATIONS

Assessing the validity of explorative qualitative research is a challenging task [18], [9]. With that in mind, we discuss the credibility and limitations of our research findings.

A. Credibility

One of the risks of grounded theory is that the resulting findings do not *fit* with the data or the participants [2]. 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 believe 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 [11] 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.

B. 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 VII 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 VI, 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.

C. 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 [13], [28]. 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.

X. RELATED WORK

A few surveys have been conducted in order to reveal software testing practices [17], [7]. 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. [16] analyze the Apache web server and the Mozilla browser in order to quantify aspects of OSS development (e.g. reported by Raymond [21]). Raja and Tretter [20] 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 [26]. Krogh et al. developed an inductive theory on how and why people join an existing open source software community [25].

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 [3].

XI. 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 [11].

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.

REFERENCES

- [1] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, pages 1–27, 2011.
- [2] Antony Bryant and Kathy Charmaz, editors. *The SAGE Handbook of Grounded Theory*. SAGE, 2007.
- [3] Isis Cabral, Myra B. Cohen, and Gregg Rothermel. 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, 2010. Springer-Verlag.
- [4] Robert Chatley, Susan Eisenbach, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Predictable dynamic plugin systems. In *7th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 129–143. Springer-Verlag, 2004.
- [5] Juliet M. Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13:3–21, 1990.
- [6] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.
- [7] Vahid Garousi and Tan Varma. 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, November 2010.
- [8] Barney Glaser and Anselm Strauss. *The discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, 1967.
- [9] Nahid Golafshani. Understanding reliability and validity in qualitative research. *The Qualitative Report*, 8(4):597–606, 2003.
- [10] Michaela Greiler, Hans-Gerhard Gross, and Arie van Deursen. Understanding plug-in test suites from an extensibility perspective. In *Proceedings 17th Working Conference on Reverse Engineering*, pages 67–76. IEEE, 2010.
- [11] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. What Eclipse users think and do about testing: A grounded theory. Technical Report SERG-2011-010, Delft University of Technology, 2011. To appear.
- [12] Warren Harrison. Eating your own dog food. *IEEE Softw.*, 23:5–7, May 2006.
- [13] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. 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, 2010.
- [14] Klaus Marquardt. Patterns for plug-ins. In *Proceedings 4th European Conference on Pattern Languages of Programs (EuroPLoP)*, page 37pp, Bad Issee, Germany, 1999.
- [15] Atif Memon, Adam Porter, and Alan Sussman. 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, 2010. ACM.
- [16] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11:309–346, July 2002.
- [17] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen. 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, 2004. IEEE Computer Society.
- [18] Anthony J. Onwuegbuzie and Nancy L. Leech. Validity and qualitative research: An oxymoron? *Quality & Quantity*, 41(2):233–249, May 2007.
- [19] Klaus Pohl and Andreas Metzger. Software product line testing. *Commun. ACM*, 49:78–81, December 2006.
- [20] Uzma Raja and Marietta J. Tretter. Antecedents of open source software defects: A data mining approach to model formulation, validation and testing. *Inf. Technol. and Management*, 10:235–251, December 2009.
- [21] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
- [22] J. Rehmand, F. Jabeen, A. Bertolino, and A. Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, 2007.
- [23] Sherry Shavor, Jim D’Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer’s Guide to Eclipse*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.
- [24] Markus Voelter. Pluggable component: A pattern for interactive system configuration. In Paul Dyson and Martine Devos, 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, 2001.
- [25] Georg von Krogh, Sebastian Spaeth, and Karim R. Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, 2003.
- [26] Joel West and O’mahony Siobhán. The role of participation architecture in growing sponsored open source communities. *Industry & Innovation*, 15(2):145–168, 2008.
- [27] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.
- [28] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 2011.

A. Experimental Design

A.1 Motivation and Goal

Plug-in-based systems rely on plug-in components to extend a base system [6–8]. As argued by Marquardt [6], 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.

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 how systems assembled from such plug-ins are to be tested.

A.2 The Eclipse Plug-in Architecture

As the subject of our study we selected the Eclipse plug-in framework¹ along with community of practitioners. We selected Eclipse for a number of reasons.

Firstly, Eclipse provides a sophisticated plug-in mechanism based on OSGi,² enhanced with the Eclipse-specific extension mechanism. It is used to build a large variety of different applications³ ranging from widely used collection of development environments to dedicated products built using the Rich Client Platform (RCP). Many of those plug-in-based products are large scale, complex, and industry-strength.

Secondly, we choose Eclipse because of the large community of Eclipse professionals involved in development of applications based on the Eclipse plug-in framework. As an example, approximately 1000 developers meet at the annual EclipseCon event alone.

The third main reason is the positive attitude of the Eclipse community towards testing, as exemplified by the presence of substantial test suites (see, e.g., our analysis of the Mylyn and eGit test suites [5]) and books emphasizing test-driven development of plug-ins [3]. Moreover, Eclipse has explicit support for testing plug-ins, through its *Plug-in Development Environment* (PDE) and the corresponding PDE-tests.

Further, the Eclipse framework itself and many project building upon it are open source, making it easy to inspect projects e.g., access to Bugzilla, the code, but also to share findings with other researchers. Besides open source development, the Eclipse platform is used for closed source commercial development, making it possible to compare open and closed source testing approaches.

A.3 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 interview guidelines from the beginning on. During our interviews, we observed that many professionals explain how limited testing is compensated, which led to the last research questions.

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?

A.4 Research Method

The experimental design covers three phases, as shown in Figure 1, comprising literature analysis, actual interviews, and an evaluation phase.

A.4.1 Plug-in Specific Literature Analysis

The literature analysis aims at identifying existing documentation that can help to answer our research questions.

¹<http://www.eclipse.org>

²<http://www.osgi.org>

³http://en.wikipedia.org/wiki/List_of_Eclipse-based_software

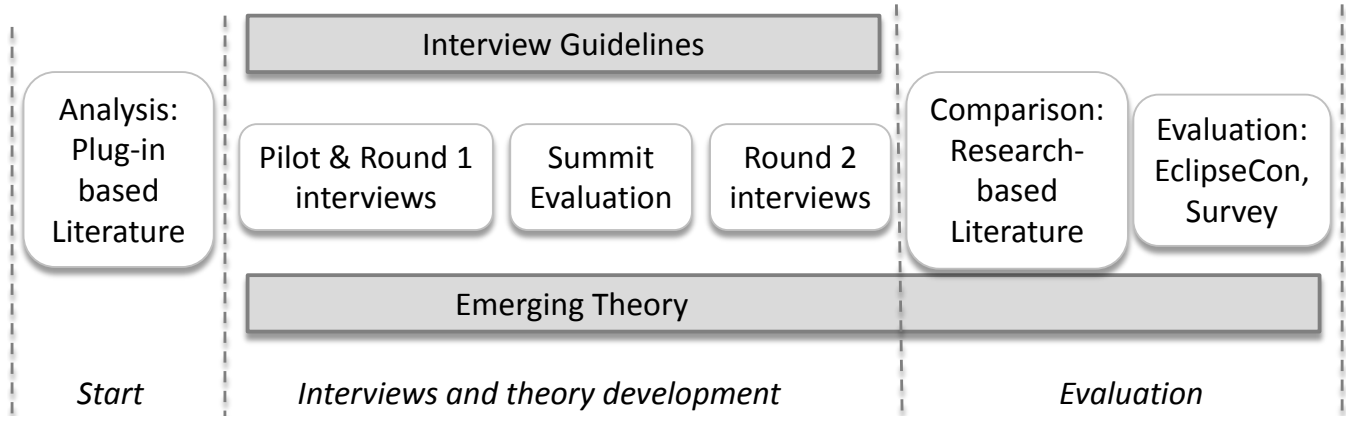


Figure 1. Phases of the Study

We studied over 200 resources about testing of plug-in systems in general and the Eclipse plug-in architecture in particular. Relevant sources include *eclipsepedia*,⁴ *eclipse resource* articles,⁵ *stackoverflow*⁶ and accepted presentations at the two main Eclipse conferences between 2003-2011.⁷ To finalize our literature selection we blogged about our findings and asked for feedback and potential missing resources.⁸

Most of the articles are concerned with either the set-up of the test environment, including test execution systems, or the use of test frameworks facilitating mocking, GUI testing, or set-up of infrastructure (including provisioning of the OSGi framework). Thus, these articles offer little help in answering our research questions, which is why we decided to engage in interviewing Eclipse experts instead.

A.4.2 Interviews and Theory Development

In order to truly understand which test techniques Eclipse developers adopt, why they do this, and which impediments they experience, we adopt an explorative research method: *Grounded Theory* (GT) [4]. This is a systematic, inductive and comparative approach to grow a theory iteratively from data, increasingly used in the field of software engineering [1]. In contrast to the hypothetico-deductive method, where the researcher has predefined hypothesis at the beginning of the investigation, grounded theory is *explorative*, aimed at discovering new perspectives and insights.

The process per interview is shown in Figure 2. It starts with an open interview of 1–2 hours, usually conducted over Skype, which we recorded and subsequently transcribed. We preferred to use the mother tongue of the interviewees, in cases it was either German or English. To structure the interviews, we compose a *guideline*, which can be adjusted after each interview, as our insight in the testing processes increases.

To analyze the transcripts we use *coding*: breaking up the interviews into smaller coherent units (sentences or paragraphs), and adding *codes* to these (identifiers representing a key characteristic such as “mocking”, “performance”, or “singleton state”). Codes can be organized hierarchically into *concepts* (such as “infrastructure”, or “testing impediments”), which in turn can be grouped into *categories*. The process of data analysis consists of a *constant comparison* of text units, codes, concepts and categories.

The resulting concepts and categories are connected via *memos*: narratives explaining, for example, which factors play a role when a developer indicates his team does not have an explicit integration testing phase. While analyzing each transcript, the elements of the theory (codes, concepts, memos, etc., and their connections) are revisited, possibly leading to theory refinement.

When we notice that interviewees progressively give answers that are similar to earlier ones, a state of *saturation* is reached: we consider the topic sufficiently clear, and adjust the interview guidelines to focus on the testing practices that require further elaboration.

As shown in Figure 1, we group the interviews into two rounds. The first round is broad in nature helping to get a first impression of the main testing processes. A first evaluation of these results was conducted through a poster presentation at the

⁴<http://wiki.eclipse.org>

⁵<http://www.eclipse.org/resources>

⁶<http://stackoverflow.com>

⁷EclipseCon (USA), and Eclipse Summit (Europe)

⁸<http://the-eclipse-study.blogspot.com/2010/11/testing-eclipse-rcp-and-plugin-in.html>

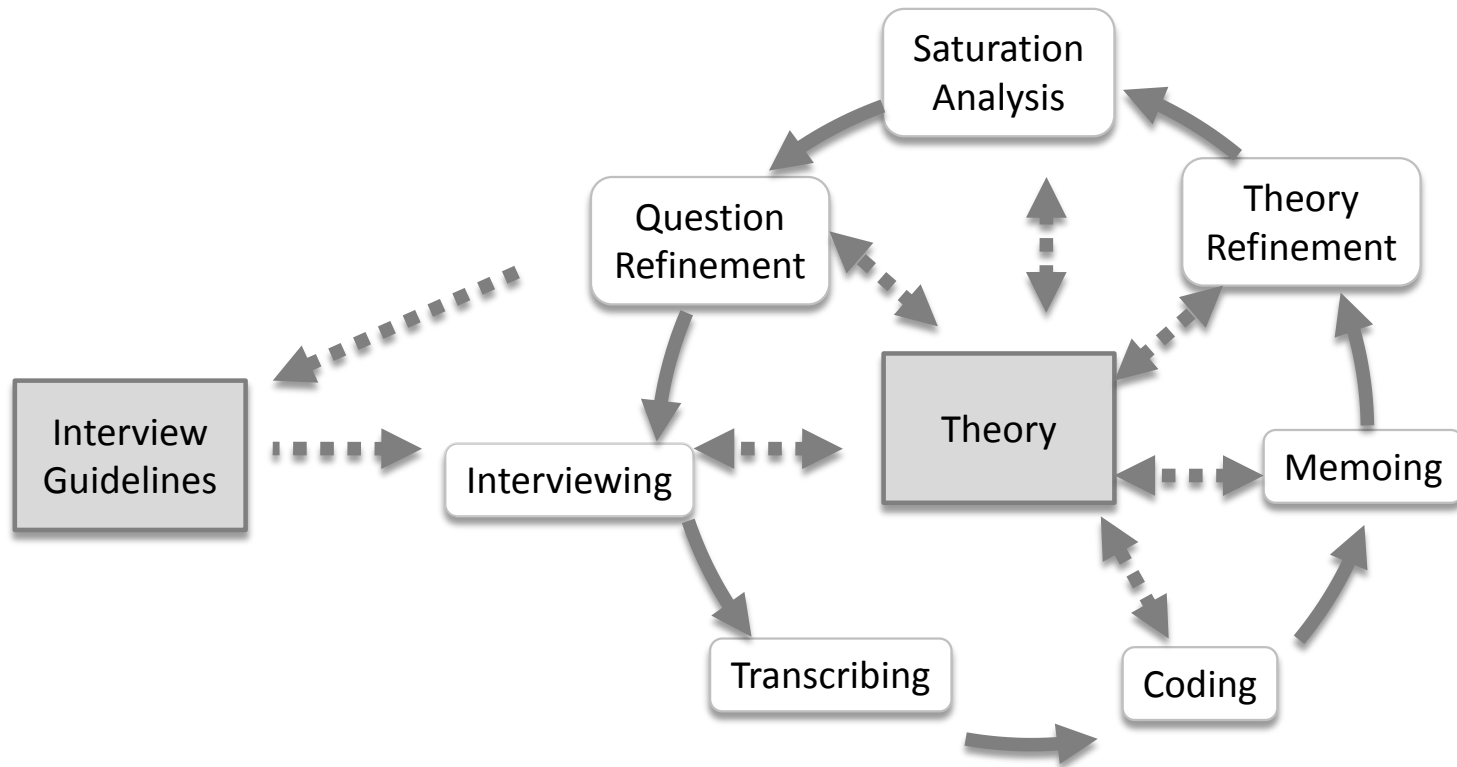


Figure 2. The interviewing and coding process followed for each interview

Eclipse Summit Europe.⁹ The second round of interviews is more specialized, focusing on specific testing issues such as the actual problems experienced for integration testing and the involvement of the user community.

A.4.3 Evaluation

Only after all interviews and the data analysis has been finished, we started a thorough literature study on studies of related research fields to compare our findings. See Section X for more details on the related work.

The final step in our study was a large-scaled evaluation at the biggest Eclipse event, the EclipseCon,¹⁰ with approximately 1000 participants. At this event we firstly presented our findings to a broad audience during a 40 minutes extended talk, where we also actively requested and discussed audience feedback. To engage in an even more in-depth discussion with the community we also organized a dedicated discussion session¹¹ (called BOF), in which more than 20 participants discussed with us their testing experiences and the usefulness and completeness of our theory.

In addition, we prepared a survey to challenge and possibly refuse our theory and to get a broader perspective on our results, which was completed by 151 practitioners. The structure of the survey followed directly from the resulting theory, and addresses the “testing practices”, “plug-in integration testing”, “adoption factors”, and “compensation strategies”.¹²

A.5 Participant Selection

Interview Phase. For the interviews we adopted *theoretical sampling* to identify suitable participants [2]. Thus, we carefully selected knowledgeable Eclipse professionals, called Eclipsers sequentially, who could provide relevant information on testing

⁹<http://www.eclipsecon.org/summiteurope2010/sessions/?page=sessions&id=1956>

¹⁰<http://www.eclipsecon.org/2011/sessions/?page=sessions&id=2207>

¹¹<http://www.eclipsecon.org/2011/sessions/?page=sessions&id=2474>

¹²The survey is available at http://swel.tudelft.nl/twiki/pub/MichaelaGreiler/WebHome/Survey_PrintVersion.pdf

Domain	Project and/or Company
IDEs, Eclipse Distribution	Yoxos, EclipseSource
SOA	Mangrove, SOA, Inria
GUI Testing Tool	GUIDancer, Brexex
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 1. Domains, projects and/or companies involved in the interviews

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 2. Participants involved (P: participants, CR: code repository (closed or open), TS: team size)

practices. We got into contact with them by participating in the Eclipse Testing Day,¹³ through a blog we maintained about the study,¹⁴ via Twitter,¹⁵ and via a poster presentation at the Eclipse Summit Europe.¹⁶

Eventually, this resulted in 25 participants (identified as P1–P25 in this paper), whose characteristics are provided in Table 2. They all have substantial experience in developing and/or testing Eclipse plug-ins or RCP products. A summary of the projects and application domains the participants worked on is given in Table 1.¹⁷

Survey Phase. In the survey phase we aimed at reaching not only the experts, but the full Eclipse community. To that end, we setup an on line survey, and announced via mailing lists, Twitter, and, most importantly, during a presentation at EclipseCon 2011 attended by over 100 people.

References

- [1] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, pages 1–27, 2011.
- [2] Antony Bryant and Kathy Charmaz, editors. *The SAGE Handbook of Grounded Theory*. SAGE, 2007.
- [3] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.
- [4] Barney Glaser and Anselm Strauss. *The discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, 1967.
- [5] Michaela Greiler, Hans-Gerhard Gross, and Arie van Deursen. Understanding plug-in test suites from an extensibility perspective. In *Proceedings 17th Working Conference on Reverse Engineering*, pages 67–76. IEEE, 2010.
- [6] Klaus Marquardt. Patterns for plug-ins. In *Proceedings 4th European Conference on Pattern Languages of Programs (EuroPLoP)*, page 37pp, Bad Issee, Germany, 1999.
- [7] Sherry Shavor, Jim D’Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer’s Guide to Eclipse*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.
- [8] Markus Voelter. Pluggable component: A pattern for interactive system configuration. In Paul Dyson and Martine Devos, 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, 2001.

¹³<http://wiki.eclipse.org/EclipseTestingDay2010>

¹⁴<http://the-eclipse-study.blogspot.com>

¹⁵<http://twitter.com/mgreiler>

¹⁶<http://www.eclipsecon.org/summiteurope2010/sessions/?page=sessions&id=1956>

¹⁷Please note that for reasons of confidentiality not all companies and projects interviewed are listed.

B. 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.

B.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.

B.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.

B.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.

B.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.

C. 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	GUI non-tested	<i>"We don't do functional testing since 80% of the code is already tested with unit tests"</i>
1.2.5	Confidence	<i>"When you design unit tests in a good way, then refactoring is easier."</i>
1.2.9	Fast execution	<i>"The good thing about unit tests is that they are fast, whereby PDE tests are slow."</i>
3.1.6	Execution time	
2.1.3	PDE as unit test	<i>"Legacy code can be problematic when unit testing, therefore sometimes PDE unit tests are misused as mini integration tests."</i>
3.1.7	Eclipse testability	
2.1.1	Workbench dependencies	<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	PDE as unit test	
3.1.7	Eclipse testability	
1.3.4	Unit vs integration testing	<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	Preference	

Participant P2

2.2.1	No influence	<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	Feedback	<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	Requirement source	<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	GUI non-tested	<i>"We don't have GUI tests, because we don't have much user interaction."</i>

Participant P3

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 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	Tester status	
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 eFitnessse."</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	Eclipse testability Ownership	<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	End user requirements Ownership	<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	Ownership	<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	GUI maintainability	<i>"We tried GUI testing for a while but it was too much work."</i>
4.4.2	Communication	<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	Communication	<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	Compatibility	<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	Communication Compatibility	<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	Manual testing Releases	<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	Coverage Key practice: Unit testing	<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	Limited confidence	<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 4.4.1 2.4.4	No automated versions Releases Manual versions	<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>

Participant P8

4.4.2	Communication	<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>
4.2.4 3.1.1 4.3.2 2.4.6 2.4.5	Multiple versions Responsibility Release train Assert compatibility Limited versions	<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>
4.2.4 3.1.1 2.4.6 2.4.5	Multiple versions Responsibility Assert compatibility Limited versions	<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>

Participant P9

1.1.1	Issue tracker	<i>"We have even an open repository, we want it to be easy to participate."</i>
4.2.4 4.4.1 2.4.3 2.4.4	Multiple versions Releases No automated versions Manual versions	<i>"To test those versions, we do it manually, but we also have some users that do that with the unstable versions."</i>
4.4.1 4.2.7 4.2.2	Releases Filing bug reports Software usage	<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>
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>
4.2.2 4.2.1 2.3.5	Software usage Manual testing No automated cross-tests	<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>
2.3.1 2.3.3 2.3.7 2.3.6	Plug-in independence Demand driven Combination issues Unpredictable	<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>
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>

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>
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>
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>

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	Issue tracker	<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	Extension points Plug-in testability Registration untested	<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	Ecosystem	<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	Manual combinations No automated cross-tests Unpredictable Unfeasible	<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	Set-up build system	<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>
4.4.2	Communication	<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>
4.4.1 4.2.2 4.4.3	Releases Software usage Test request	<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	Multiple versions Software usage Filing bug reports	<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>
4.3.4	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

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>

4.2.3	Operating systems	“The community helps to test the system for different operating systems, and versions. They are very active with that.”
4.2.4	Multiple versions	
4.1.1	Self-hosting	“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.”
1.3.8	GUI non-tested	“We do not test the GUI. The problem is that all tools are essentially useless.”
3.1.5	Set-up build system	“In addition to the long execution time, it is a hassle to get [the GUI tests] run on the build sever.”
1.3.8	GUI non-tested	“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.”
4.2.6 4.2.1	GUI community Manual testing	“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.”
2.3.1	Plug-in independence	“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.”
1.1.1 1.1.2 4.2.9	Issue tracker Requirement source Customer involvement	“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.”
2.4.2	External systems	“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.”
2.4.2	External systems	“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.”

Participant P18

1.2.1	Key practice: Unit testing	“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.”
1.2.2	Preference	“Unit testing, that’s where you find the most bugs.”
2.2.2	Modularization	“I believe OSGi is very helpful for creating clear structures. The test strategy is not changed, but there are structural changes.”
1.1.3	Developer testing	“Every developer is also a tester. That’s also due to test driven development.”
3.1.6	Execution time	“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.”
1.1.5 1.1.3	Tester status Developer testing	“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.”
2.3.3 2.3.1	Demand driven Plug-in independence	“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.”
1.3.1	Test automation	“We automate everything. That’s our main principle. Tests that are not automated, are not tests for us.”
1.3.7	GUI maintainability	“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.”

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>
4.3.5 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>
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>
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>
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>
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>
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>
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>

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	
4.3.3 2.3.7	Ecosystem Combination issues	<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>
1.3.4 1.2.3 1.2.1	Unit vs integration testing Coverage Key practice: Unit testing	<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	Unit vs integration 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	Test automation	<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

- 1.1.1 Issue tracker, P9, P13, P15, P17, P19
- 1.1.2 Requirement source, P2, 3, P5, 6, P12, P16, 17
- 1.1.3 Developer testing, P5, P18, 19
- 1.1.4 Hybrid testing, P3, P14, P21
- 1.1.5 Tester status, P3, P18
- 1.2.1 Key practice: Unit testing, P7, P15, P18, P20, P22, P25
- 1.2.2 Preference, P1, P6, P18
- 1.2.3 Coverage, P5, P7, P10, 11, P20
- 1.2.4 No coverage, P3, P13
- 1.2.5 Confidence, P1, P3, P10, P12, 13
- 1.2.6 Limited confidence, P3, P5, P7, P24
- 1.2.7 Unit testability, P14, P20
- 1.2.8 Unit non-tested, P20
- 1.2.9 Fast execution, P1, P6, P14, P20, 21
- 1.3.1 Test automation, P3, P5, P10–12, P18, P24, 25
- 1.3.2 Hardware integration, P5
- 1.3.3 Continuous integration, P3, P5, 6, P10, 11, P16, P22, P25
- 1.3.4 Unit vs integration testing, P1, P6, P14, P20, P22
- 1.3.5 Fault location, P6, P12
- 1.3.6 GUI testing, P3, P5, P10, P23
- 1.3.7 GUI maintainability, P7, P10, P14, 15, P18, P21, P23
- 1.3.8 GUI non-tested, P1, 2, P13, P17, P21
- 2.1.1 Workbench dependencies, P1, P6, P14, P21, 22
- 2.1.2 PDE as integration test, P6, P21
- 2.1.3 PDE as unit test, P1, P6, P14, P21
- 2.1.4 Headless PDE, P6
- 2.2.1 No influence, P2
- 2.2.2 Modularization, P3, P5, P18
- 2.2.3 Extension points, P13, P16, P19, P22
- 2.2.4 Registration untested, P16
- 2.2.5 Plug-in testability, P5, P16, P19
- 2.2.6 Eco-system integration, P14
- 2.2.7 GUI based, P3, P10
- 2.2.8 No eclipse integration, P15, P22
- 2.3.1 Plug-in independence, P9, P17, 18
- 2.3.2 Play nicely, P6
- 2.3.3 Demand driven, P9, P18
- 2.3.4 Manual combinations, P16, P19
- 2.3.5 No automated cross-tests, P9, P16, P19
- 2.3.6 Unpredictable, P9, P16
- 2.3.7 Combination issues, P6, P9, P20
- 2.4.1 Build system, P16, P22
- 2.4.2 External systems, P17
- 2.4.3 No automated versions, P7, P9
- 2.4.4 Manual versions, P7, P9
- 2.4.5 Limited versions, P8, P14, P16, P25
- 2.4.6 Assert compatibility, P8, P13
- 2.4.7 Update rarely, P9, P14
- 2.4.8 Unfeasible, P12, P16
- 3.1.1 Responsibility, P8, 9
- 3.1.2 End user requirements, P7
- 3.1.3

- Ownership, P6, 7
- 3.1.4 Plug-in testing knowledge, P4, P22, 23
- 3.1.5 Set-up build system, P4, P15–17, P21, 22
- 3.1.6 Execution time, P1, P5, 6, P14, P17, 18, P20–22
- 3.1.7 Eclipse testability, P1, P5–7, P14, P23
- 3.1.8 PDE integration tooling, P21
- 4.1.1 Self-hosting, P17
- 4.2.1 Manual testing, P6, 7, P9, P13, P17
- 4.2.2 Software usage, P9, P15, 16, P19
- 4.2.3 Operating systems, P12, P17, P19
- 4.2.4 Multiple versions, P8, 9, P12, P16, 17, P19
- 4.2.5 Compatibility, P7, P9, P19
- 4.2.6 GUI community, P17
- 4.2.7 Filing bug reports, P9, P11, P16, P19
- 4.2.8 Feedback, P2, P6, P11, P15, P19
- 4.2.9 Customer involvement, P3, P5, P14, 15, P17
- 4.3.1 Automated testing, P13, P20
- 4.3.2 Release train, P8, P13
- 4.3.3 Ecosystem, P16, P20
- 4.3.4 Plug-in symbiosis, P11, P16, P19, 20
- 4.3.5 Providing patches, P6, P11, 12, P19
- 4.4.1 Releases, P6, 7, P9, P15, 16, P19
- 4.4.2 Communication, P4, P7, 8, P12, 13, P16, P19
- 4.4.3 Test request, P16
- 4.4.4 Opening closed software, P19

D. Survey

The following pages contain the questionnaire as distributed among over 150 Eclipse developers.

Eclipse Testing Study

1. Background

Thank you for participating in this survey. We would be grateful if you could take the time to answer some questions about your test practices for Eclipse-based applications.

All of your individual responses will be treated as confidential. Anonymised information may be used in academic papers or other publications.

When filling in this survey, please relate the answers always to the current or most recent Eclipse-based software project you have been part of.

By participating at this survey you can make a chance to win a ultra-cool mini solar car!



★ 1. My team develops...

- ☐ ...open source software.
- ☐ ...closed source software.
- ☐ ...open and closed source software.
- ☐ Other (please specify)

★ 2. My team develops...

(Multiple answers allowed.)

- ☐ RCP applications.
- ☐ Eclipse plug-ins.
- ☐ OSGi-based, non-Eclipse applications.
- ☐ Other Java applications.
- ☐ Other (please specify)

★ 3. Do you have a separate test team?

- ☐ yes, we have a separate test team.
- ☐ no, we do not have a separate test team.
- ☐ Other (please specify)

Eclipse Testing Study

2. Test Activities

When filling in this survey, please relate the answers always to the current or most recent Eclipse-based software project you have been part of.

Unit Testing: typically comprises a relative small executable like a method, class, or several related classes

Integration Testing: testing with the intent of finding bugs in component (plug-in) interactions

System Testing: tests execute the entire system

★ 4. Please estimate the relative effort spent on each test technique.

(The total effort spent should not exceed ~100%.)

	90-100%	75-90%	50-25%	25-10%	<10%	0%	I don't know
Unit Testing	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>
Integration Testing	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>
GUI Testing	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>
System Testing	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>

★ 5. Please indicate the level of test automation?

(If you do not practice a technique please check "not applicable".)

	fully automated	main effort test automation	main effort manual	only manual testing	I don't know	not applicable
Unit Testing	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>
Integration Testing	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>
GUI Testing	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>
System Testing	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>	<input type="text" value="jñ"/>

Eclipse Testing Study

3. Test Activities

Unit Testing: typically comprises a relative small executable like a method, class, or several related classes

Integration Testing: testing with the intent of finding bugs in component (plug-in) interactions

System Testing: tests execute the entire system

★ 6. How important do you personally think are the following test techniques?

	unimportant	less important	quite important	important	very important	I don't know
Unit Testing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Integration Testing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
GUI Testing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
System Testing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

★ 7. How experienced are you with the following test techniques?

	not experienced	quite experienced	experienced	very experienced
Unit Testing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Integration Testing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
GUI Testing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
System Testing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

★ 8. Have you received any training related to software testing?

(like at University, or company internal trainings etc.)

☐ yes

☐ no

9. If you had training, please indicate by keywords which training you followed.

For example: company internal course on test-driven development

5

6

Eclipse Testing Study

4. Integration Testing

When filling in this survey, please relate the answers always to the current or most recent Eclipse-based software project you have been part of.

★ 10. **Is cross-feature integration testing performed in your project?**

(Cross-feature integration is for example the integration of your plug-in with a third party plug-in like EcEmma or Mylyn. Multiple answers allowed.)

- ☐ yes, we have AUTOMATED tests for this.
- ☐ yes, we have MANUAL tests for this.
- ☐ yes, but in an ad-hoc manner.
- ☐ yes, we address that very thoroughly.
- ☐ yes, this is done by the user community.
- ☐ no, we do not test that.

★ 11. **Do you test compatibility of your plug-in with several different Eclipse platform versions?**

(Multiple answers allowed.)

- ☐ yes, we have AUTOMATED tests for this.
- ☐ yes, we have MANUAL tests for this.
- ☐ yes, but testing is limited (e.g., to the current and the latest version of the platform).
- ☐ yes, we address that very thoroughly.
- ☐ yes, this is done by the user community.
- ☐ no, we do not test that.

★ 12. **Do you test compatibility of your plug-in with different versions of plug-ins and libraries you depend on?**

(Multiple answers allowed.)

- ☐ yes, we have AUTOMATED tests for this.
- ☐ yes, we have MANUAL tests for this.
- ☐ yes, but not for all versions of the version ranges (e.g., done for two versions of each dependency).
- ☐ yes, we address that very thoroughly.
- ☐ yes, this is done by the user community.
- ☐ no, we do not test that.

Eclipse Testing Study

5. Organizational Problems

When filling in this survey, please relate the answers always to the current or most recent Eclipse-based software project you have been part of.

13. Please indicate for each test technique you actually have used if you have experienced the following organizational problems.

(Please leave the entire column of the test practices you have never used blank.)

	Unit Testing	Integration Testing	System Testing	GUI Testing
Lack of time for this test practice	€	€	€	€
Test technique is perceived as less important	€	€	€	€
No recognizable benefits of this test practice	€	€	€	€
Unclear who is/feels responsible for performing this test practice	€	€	€	€
Practice too time consuming	€	€	€	€
Unclear or unknown design documents or end user requirements	€	€	€	€
Restricted controllability of foreign plug-ins	€	€	€	€
Unclear who is/feels responsible for overall quality	€	€	€	€
Testing less appreciated than development activities	€	€	€	€

Eclipse Testing Study

6. Technical Problems

14. Please indicate for each test technique you actually have used if you have experienced the following technical problems.

(Please leave the entire column of the test practices you have never use blank.)

	Unit Testing	Integration Testing	System Testing	GUI Testing
Lack of knowledge or expertise	€	€	€	€
Difficult to set-up test execution environment	€	€	€	€
Hard to test highly coupled, or legacy code	€	€	€	€
Hard to test code tightly coupled to Eclipse	€	€	€	€
High maintenance effort	€	€	€	€
Immature tooling or missing test infrastructure	€	€	€	€
Long Test Execution Time	€	€	€	€

15. Are there other organizational or technical problems you experienced we have not mentioned? Please specify them and also the test practices they apply to.

Example: missing dummy devices for integration testing

5

6

Eclipse Testing Study

7. User Involvement

When filling in this survey, please relate the answers always to the current or most recent Eclipse-based software project you have been part of.

16. How open are your software development activities to users?

- ☐ We have an open issue tracking system.
- ☐ We have a publicly accessible software repository.
- ☐ We have mailing lists, and/or newsgroups anyone can join and post.
- ☐ We inform users by e.g., blog posts, newsletters, Twitter...
- ☐ We are completely closed.
- ☐ Other (please specify)

17. The users of our software...

- ☐ ...represent an open community, everybody can join.
- ☐ ...represent a closed community (e.g., a specific costumer)
- ☐ Other (please specify)

18. In which activities are the users involved?

(Multiple answers allowed.)

- ☐ Giving feedback, and foster discussions.
- ☐ Providing bug reports or feature requests.
- ☐ Providing bug fixes.
- ☐ Manual testing.
- ☐ Manual GUI testing.
- ☐ Manual combinatorial testing (e.g., different OS, Eclipse versions, or plug-in combinations).
- ☐ Automated testing.
- ☐ Automated testing done by downstream projects.
- ☐ Automated system testing done by the community involved in the release train.
- ☐ Other (please specify)

Eclipse Testing Study

8. Background And Prizes

* 19. In which country do you live?

* 20. Project team size

☐ 1-5

☐ 6-10

☐ 11-20

☐ >20

* 21. Professional role

☐ Developer

☐ Solely Tester

☐ Project Lead / Manager

☐ Test Manager

☐ Other (please specify)

22. Do you have any remarks or questions? Please feel free to specify them.

23. If you want to be considered for winning a ultra-cool mini solar car, please leave your email.

We will not use your email for advertisement neither will we distribute it to third parties.



Eclipse Testing Study

9. Thank you for participating @ the Eclipse Testing Study

If you are interested in the results of this study please say so in an email to eclipsestudy@gmail.com. Then, we send you the report with the results. Or visit <http://the-eclipse-study.blogspot.com>.

This survey is being carried out by Michaela Greiler and Arie van Deursen on behalf of the Delft University of Technology.

E. Survey Responses





The following pages contain the survey responses, as generated by the online questionnaire tool SurveyMonkey.¹⁸

¹⁸ www.surveymonkey.com

Eclipse Testing Study








1. My team develops...




		Response Percent	Response Count
...open source software.		21.2%	32
...closed source software.		47.0%	71
...open and closed source software.		31.1%	47
Other (please specify)		0.7%	1
answered question			151
skipped question			0

2. My team develops...

(Multiple answers allowed.)

		Response Percent	Response Count
RCP applications.		51.0%	77
Eclipse plug-ins.		63.6%	96
OSGi-based, non-Eclipse applications.		15.9%	24
Other Java applications.		41.7%	63
Other (please specify)		9.3%	14
answered question			151
skipped question			0

3. Do you have a separate test team?

		Response Percent	Response Count
yes, we have a separate test team.		34.4%	52
no, we do not have a separate test team.		63.6%	96
Other (please specify)		2.0%	3
answered question			151
skipped question			0

4. Please estimate the relative effort spent on each test technique.

(The total effort spent should not exceed ~100%.)

	90-100%	75-90%	50-25%	25-10%	<10%	0%	I don't know	Response Count
Unit Testing	5.6% (7)	7.3% (9)	23.4% (29)	27.4% (34)	26.6% (33)	6.5% (8)	3.2% (4)	124
Integration Testing	2.6% (3)	3.4% (4)	27.4% (32)	32.5% (38)	17.1% (20)	10.3% (12)	6.8% (8)	117
GUI Testing	4.0% (5)	6.5% (8)	20.2% (25)	29.8% (37)	27.4% (34)	8.9% (11)	3.2% (4)	124
System Testing	4.9% (6)	5.7% (7)	20.5% (25)	19.7% (24)	24.6% (30)	13.1% (16)	11.5% (14)	122
answered question								126
skipped question								25

5. Please indicate the level of test automation?

(If you do not practice a technique please check "not applicable".)

	fully automated	main effort test automation	main effort manual	only manual testing	I don't know	not applicable	Response Count
Unit Testing	47.6% (60)	17.5% (22)	11.9% (15)	15.1% (19)	0.0% (0)	7.9% (10)	126
Integration Testing	16.1% (19)	25.4% (30)	22.9% (27)	21.2% (25)	4.2% (5)	10.2% (12)	118
GUI Testing	15.8% (19)	19.2% (23)	27.5% (33)	28.3% (34)	2.5% (3)	6.7% (8)	120
System Testing	6.0% (7)	12.8% (15)	22.2% (26)	36.8% (43)	9.4% (11)	12.8% (15)	117
answered question							126
skipped question							25

6. How important do you personally think are the following test techniques?

	unimportant	less important	quite important	important	very important	I don't know	Response Count
Unit Testing	0.8% (1)	8.5% (10)	15.3% (18)	13.6% (16)	61.9% (73)	0.0% (0)	118
Integration Testing	0.0% (0)	2.6% (3)	14.8% (17)	25.2% (29)	55.7% (64)	1.7% (2)	115
GUI Testing	0.0% (0)	9.6% (11)	16.5% (19)	33.9% (39)	40.0% (46)	0.0% (0)	115
System Testing	0.0% (0)	7.9% (9)	12.3% (14)	29.8% (34)	42.1% (48)	7.9% (9)	114
answered question							118
skipped question							33

7. How experienced are you with the following test techniques?

	not experienced	quite experienced	experienced	very experienced	Response Count
Unit Testing	5.9% (7)	22.9% (27)	33.1% (39)	38.1% (45)	118
Integration Testing	12.9% (15)	37.1% (43)	27.6% (32)	22.4% (26)	116
GUI Testing	22.4% (26)	28.4% (33)	31.9% (37)	17.2% (20)	116
System Testing	26.5% (30)	38.1% (43)	22.1% (25)	13.3% (15)	113
answered question					118
skipped question					33

8. Have you received any training related to software testing? (like at University, or company internal trainings etc.)







		Response Percent	Response Count
yes		31.4%	37
no		68.6%	81
answered question			118
skipped question			33

9. If you had training, please indicate by keywords which training you followed. For example: company internal course on test-driven development







	Response Count
	34
answered question	34
skipped question	117

10. Is cross-feature integration testing performed in your project?







(Cross-feature integration is for example the integration of your plug-in with a third party plug-in like EcIEmma or Mylyn. Multiple answers allowed.)

		Response Percent	Response Count
yes, we have AUTOMATED tests for this.		23.0%	26
yes, we have MANUAL tests for this.		19.5%	22
yes, but in an ad-hoc manner.		23.9%	27
yes, we address that very thoroughly.		1.8%	2
yes, this is done by the user community.		7.1%	8
no, we do not test that.		42.5%	48
answered question			113
skipped question			38

11. Do you test compatibility of your plug-in with several different Eclipse platform versions?
 (Multiple answers allowed.)

		Response Percent	Response Count
yes, we have AUTOMATED tests for this.		13.3%	15
yes, we have MANUAL tests for this.		15.9%	18
yes, but testing is limited (e.g., to the current and the latest version of the platform).		20.4%	23
yes, we address that very thoroughly.		1.8%	2
yes, this is done by the user community.		5.3%	6
no, we do not test that.		54.9%	62
answered question			113
skipped question			38

12. Do you test compatibility of your plug-in with different versions of plug-ins and libraries you depend on?
(Multiple answers allowed.)

		Response Percent	Response Count
yes, we have AUTOMATED tests for this.		12.4%	14
yes, we have MANUAL tests for this.		14.2%	16
yes, but not for all versions of the version ranges (e.g., done for two versions of each dependency).		13.3%	15
yes, we address that very thoroughly.		3.5%	4
yes, this is done by the user community.		2.7%	3
no, we do not test that.		62.8%	71
answered question			113
skipped question			38

13. Please indicate for each test technique you actually have used if you have experienced the following organizational problems.

(Please leave the entire column of the test practices you have never used blank.)

	Unit Testing	Integration Testing	System Testing	GUI Testing	Response Count
Lack of time for this test practice	55.4% (36)	52.3% (34)	50.8% (33)	55.4% (36)	65
Test technique is perceived as less important	32.7% (17)	46.2% (24)	32.7% (17)	38.5% (20)	52
No recognizable benefits of this test practice	25.0% (8)	25.0% (8)	40.6% (13)	34.4% (11)	32
Unclear who is/feels responsible for performing this test practice	20.0% (9)	51.1% (23)	57.8% (26)	46.7% (21)	45
Practice too time consuming	39.1% (25)	32.8% (21)	40.6% (26)	67.2% (43)	64
Unclear or unknown design documents or end user requirements	31.0% (13)	54.8% (23)	61.9% (26)	57.1% (24)	42
Restricted controllability of foreign plug-ins	28.6% (6)	47.6% (10)	42.9% (9)	28.6% (6)	21
Unclear who is/feels responsible for overall quality	19.4% (6)	41.9% (13)	58.1% (18)	58.1% (18)	31
Testing less appreciated than development activities	67.3% (35)	63.5% (33)	55.8% (29)	67.3% (35)	52
answered question					82
skipped question					69

14. Please indicate for each test technique you actually have used if you have experienced the following technical problems.

(Please leave the entire column of the test practices you have never use blank.)







	Unit Testing	Integration Testing	System Testing	GUI Testing	Response Count
Lack of knowledge or expertise	37.7% (20)	34.0% (18)	50.9% (27)	60.4% (32)	53
Difficult to set-up test execution environment	24.6% (17)	52.2% (36)	50.7% (35)	60.9% (42)	69
Hard to test highly coupled, or legacy code	49.1% (26)	52.8% (28)	45.3% (24)	41.5% (22)	53
Hard to test code tightly coupled to Eclipse	39.5% (17)	53.5% (23)	30.2% (13)	44.2% (19)	43
High maintenance effort	34.8% (23)	43.9% (29)	40.9% (27)	62.1% (41)	66
Immature tooling or missing test infrastructure	22.8% (13)	57.9% (33)	61.4% (35)	66.7% (38)	57
Long Test Execution Time	23.0% (14)	41.0% (25)	49.2% (30)	50.8% (31)	61
answered question					85
skipped question					66

15. Are there other organizational or technical problems you experienced we have not mentioned? Please specify them and also the test practices they apply to.




Example: missing dummy devices for integration testing

	Response Count
	11
answered question	11
skipped question	140




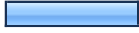






16. How open are your software development activities to users?

		Response Percent	Response Count
We have an open issue tracking system.		64.0%	57
We have a publicly accessible software repository.		38.2%	34
We have mailing lists, and/or newsgroups anyone can join and post.		40.4%	36
We inform users by e.g., blog posts, newsletters, Twitter...		36.0%	32
We are completely closed.		27.0%	24
Other (please specify)		4.5%	4
	answered question		89
	skipped question		62

17. The users of our software...

		Response Percent	Response Count
...represent an open community, everybody can join.		43.8%	39
...represent a closed community (e.g., a specific costumer)		52.8%	47
Other (please specify)		3.4%	3
answered question			89
skipped question			62





18. In which activities are the users involved?
(Multiple answers allowed.)

		Response Percent	Response Count
Giving feedback, and foster discussions.		81.8%	72
Providing bug reports or feature requests.		85.2%	75
Providing bug fixes.		25.0%	22
Manual testing.		21.6%	19
Manual GUI testing.		27.3%	24
Manual combinatorial testing (e.g., different OS, Eclipse versions, or plug-in combinations).		12.5%	11
Automated testing.		6.8%	6
Automated testing done by downstream projects.		3.4%	3
Automated system testing done by the community involved in the release train.		5.7%	5
Other (please specify)		6.8%	6
answered question			88
skipped question			63






19. In which country do you live?

	Response Count
	92
answered question	92
skipped question	59

20. Project team size

		Response Percent	Response Count
1-5		32.6%	30
6-10		30.4%	28
11-20		19.6%	18
>20		17.4%	16
answered question			92
skipped question			59

21. Professional role

		Response Percent	Response Count
Developer		64.1%	59
Solely Tester		3.3%	3
Project Lead / Manager		22.8%	21
Test Manager		2.2%	2
Other (please specify)		7.6%	7
answered question			92
skipped question			59

22. Do you have any remarks or questions? Please feel free to specify them.

	Response Count
	11
answered question	11
skipped question	140

Page 8, Q19. **In which country do you live?**

1	Brazil	Mar 21, 2011 8:02 AM
2	France	Mar 21, 2011 8:42 AM
3	Russia	Mar 21, 2011 10:29 AM
4	Canada	Mar 21, 2011 10:32 AM
5	USA	Mar 21, 2011 10:33 AM
6	US	Mar 21, 2011 10:38 AM
7	Poland	Mar 21, 2011 10:39 AM
8	Bulgaria	Mar 21, 2011 10:48 AM
9	Russia	Mar 21, 2011 10:59 AM
10	Usa	Mar 21, 2011 11:04 AM
11	Switzerland	Mar 21, 2011 11:55 AM
12	Switzerland	Mar 21, 2011 11:58 AM
13	India	Mar 21, 2011 12:18 PM
14	Germany	Mar 21, 2011 12:58 PM
15	italy	Mar 21, 2011 1:07 PM
16	Poland	Mar 21, 2011 1:10 PM
17	Germany	Mar 21, 2011 1:27 PM
18	France	Mar 21, 2011 1:47 PM
19	France	Mar 21, 2011 5:24 PM
20	France	Mar 21, 2011 5:42 PM
21	Germany	Mar 22, 2011 12:08 AM
22	France	Mar 22, 2011 12:59 AM
23	Sweden	Mar 22, 2011 4:00 AM
24	france	Mar 22, 2011 5:24 AM
25	Germany	Mar 22, 2011 6:19 AM
26	Germany	Mar 22, 2011 7:57 AM
27	Switzerland	Mar 22, 2011 9:52 AM
28	Bulgaria	Mar 22, 2011 4:31 PM
29	germany	Mar 22, 2011 7:49 PM

Page 8, Q19. In which country do you live?

30	USA	Mar 22, 2011 9:08 PM
31	us	Mar 22, 2011 9:09 PM
32	USA	Mar 22, 2011 10:16 PM
33	Brazil	Mar 23, 2011 5:50 AM
34	Brasil	Mar 23, 2011 5:59 AM
35	Brazil	Mar 23, 2011 6:16 AM
36	romania	Mar 23, 2011 7:01 AM
37	Brazil	Mar 23, 2011 7:04 AM
38	switzerland	Mar 23, 2011 7:49 AM
39	Canada	Mar 23, 2011 8:15 AM
40	USA	Mar 23, 2011 8:41 AM
41	Brazil	Mar 23, 2011 9:01 AM
42	Luxembourg	Mar 23, 2011 9:12 AM
43	Canada	Mar 23, 2011 9:24 AM
44	U.S.A.	Mar 23, 2011 9:47 AM
45	Hungary	Mar 23, 2011 10:20 AM
46	USA	Mar 23, 2011 10:36 AM
47	Brazil	Mar 23, 2011 10:44 AM
48	USA	Mar 23, 2011 11:10 AM
49	Germany	Mar 23, 2011 11:34 AM
50	Canada	Mar 23, 2011 12:09 PM
51	United States	Mar 23, 2011 12:28 PM
52	Russia	Mar 23, 2011 2:29 PM
53	South Korea	Mar 23, 2011 2:47 PM
54	BC, Canada	Mar 23, 2011 5:02 PM
55	India	Mar 23, 2011 9:10 PM
56	Germany	Mar 23, 2011 11:44 PM
57	USA	Mar 24, 2011 6:53 AM
58	USA	Mar 24, 2011 1:17 PM

Page 8, Q19. **In which country do you live?**

59	Canada	Mar 24, 2011 4:36 PM
60	USA	Mar 24, 2011 5:35 PM
61	rft	Mar 25, 2011 5:07 AM
62	United Kingdom	Mar 25, 2011 6:10 AM
63	U.S.	Mar 25, 2011 11:44 AM
64	CANADA	Mar 26, 2011 8:58 AM
65	INDIA	Mar 26, 2011 10:30 AM
66	UK	Mar 27, 2011 12:13 PM
67	Switzerland	Mar 27, 2011 10:17 PM
68	Bulgaria	Mar 28, 2011 1:03 AM
69	sweden	Mar 28, 2011 2:31 AM
70	Germany	Mar 28, 2011 2:34 AM
71	Germany	Mar 28, 2011 2:42 AM
72	Germany	Mar 28, 2011 2:43 AM
73	Germany	Mar 28, 2011 4:16 AM
74	Germany	Mar 28, 2011 7:35 AM
75	Germany	Mar 28, 2011 9:23 AM
76	USA	Mar 29, 2011 9:01 AM
77	United States	Mar 30, 2011 9:15 AM
78	Switzerland	Mar 30, 2011 6:36 PM
79	Russia	Mar 31, 2011 12:10 AM
80	Germany	Mar 31, 2011 2:49 AM
81	Verigy	Apr 1, 2011 4:44 AM
82	Hungary	Apr 3, 2011 12:18 PM
83	Canada	Apr 4, 2011 7:00 AM
84	France	Apr 4, 2011 7:02 AM
85	Canada	Apr 4, 2011 10:13 AM
86	Denmark	Apr 4, 2011 12:02 PM
87	India	Apr 4, 2011 3:05 PM

Page 8, Q19. In which country do you live?

88	USA	Apr 5, 2011 5:20 AM
89	Canada	Apr 5, 2011 11:41 AM
90	US	Apr 6, 2011 3:39 AM
91	Germany	Apr 7, 2011 7:22 AM
92	Canada	Apr 7, 2011 1:37 PM

Page 8, Q21. Professional role

1	Trainer	Mar 21, 2011 5:24 PM
2	release engineer	Mar 22, 2011 5:24 AM
3	CTO	Mar 22, 2011 9:52 AM
4	exec	Mar 22, 2011 9:09 PM
5	Software Engineering Specialists : help to improve software processes, including the test part.	Mar 23, 2011 9:12 AM
6	Architect / Consultant / Coach	Apr 4, 2011 3:05 PM
7	release engineer	Apr 5, 2011 11:41 AM

