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 infrastructure[1] used in the Firefox browser, Apache's Maven build manager,[2] the WordPress extension mechanism,[3] and the Eclipse[4] 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

978-1-4673-1067-3/12/$31.00

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 Marqaurdt [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 framework[11] along with its community of practitioners. We selected Eclipse for a number of reasons.

First, Eclipse provides a sophisticated plug-in mechanism based on OSGi[12] 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

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

[7] http://www.wpmods.com/wordpress-plugin-compability-procedure

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

[9] Bug Identifier: 268207

[10] http://sourceforge.net/apps/mediawiki/qcmylyn

[11] http://www.eclipse.org

[12] http://www.osgi.org

[13] http://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 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]

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

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

[15] Please 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.

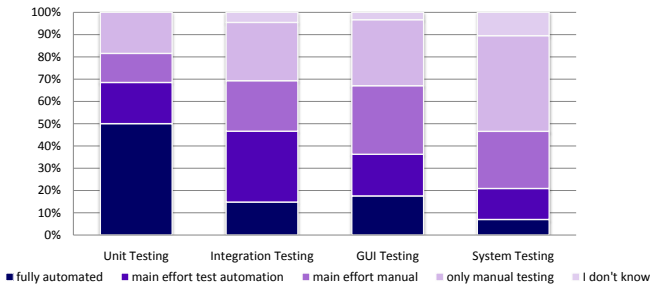

Figure 1. Test automation for each test practice

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
[17]Bug Identifier: 280598 and 213988

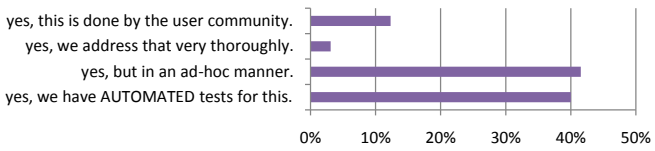

Figure 2. Cross-Product Integration Testing

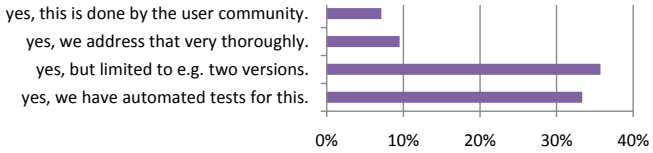

Figure 3. Testing versions of plug-in dependencies.

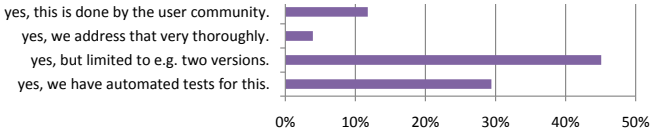

Figure 4. Testing Eclipse platform versions.

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?

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 train[19] 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."*

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

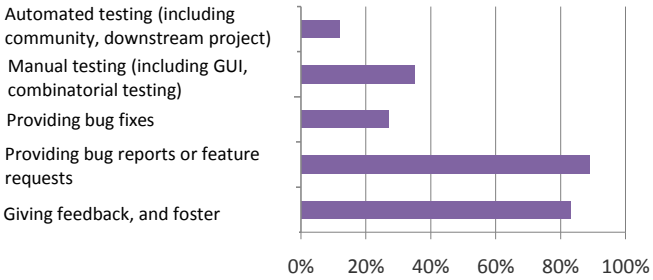

Figure 5. User involvement during testing

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

[20]E.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 preconceived 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 Eclipsers 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 Irsee, 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.