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

Michaela Greiler and Arie van Deursen

Delft University of Technology
{m.s.greiler|arie.vandeursen}@tudelft.nl

Abstract. Software architectures such as plug-in and service-oriented architectures enable developers to build extensible software products, whose functionality can be enriched by adding or configuring components. A well-known example of such an architecture is Eclipse, best known for its use to create a series of extensible IDEs. Although such architectures give users and developers a great deal of flexibility to create new products, the complexity of the built systems increases. In order to manage this complexity developers use extensive automated test suites. Unfortunately, current testing tools offer little insight in which of the many possible combinations of components and components configurations are actually tested. The goal of this paper is to remedy this problem.
To that end, we interview 25 professional developers on the problems they experience in test suite understanding for plug-in architectures. The findings have been incorporated in five architectural views that provide an extensibility perspective on plug-in-based systems and their test suites. The views combine static and dynamic information on plug-in dependencies, extension initialization, extension and service usage, and the test suites. The views have been implemented in ETSE, the Eclipse Plug-in Test Suite Exploration tool. We evaluate the proposed views by analyzing eGit, Mylyn, and a Mylyn connector.

1 Introduction

Plug-in architectures are widely used for complex systems such as browsers, development environments, or embedded systems, since they support modularization, product extensibility, and run time product adaptation and configuration [4, 25, 27]. A well-known example of such an architecture is Eclipse[1] which has been used for building a variety of extensible products, including a range of development environments for different languages [36].

The size and complexity of software products based on plug-ins can be substantial. To deal with this, software developers rely on extensive automated test suites. For example, in their book *Contributing to Eclipse*, Gamma and Beck emphasize test-driven development of Eclipse plug-ins [16]. Likewise, the Eclipse developer web site[2] describes the structure of the unit and user interface tests that come with Eclipse.

A consequence of systematic automated testing is the *test suite understanding problem*: Developers working with such well-tested plug-in-based architectures, face the

[1] http://www.eclipse.org
[2] http://wiki.eclipse.org/Eclipse/Testing

problem of understanding a sizable code base along with a substantial test suite. As an example, the Mylyn[3] plug-in for Eclipse comes with approximately 50,000 lines of test code. Developers responsible for modifying Mylyn, must also adjust the Mylyn test suite.

To address the test suite understanding problem, researchers have identified *test smells* pointing to problematic test code, *test refactorings* for improving them, and have proposed visualizations of test execution [6, 11, 29, 33]. Most of the existing work, however, focuses on the *unit* level. While this is an essential first step, for plug-in-based architectures it will not reveal how plug-ins are loaded, initialized, and executed dynamically. As an example, just starting Eclipse loads close to one hundred plug-ins. Since these plug-ins do have interactions, looking at plug-ins in isolation yields insufficient insight in the way the dynamic plug-in configuration is exercised in test suites.

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

First of all, in order to get insight in the nature of this problem, we interview 25 senior professionals from the Eclipse community on their testing practices. This study was set up as a *Grounded Theory* study [1, 5]. The outcomes include a number of challenges professional developers face when confronted with complex test suites for the plug-ins they are working on.

Subsequently, to address these challenges, we propose a series of *architectural views* [10] that can help engineers understand plug-in interactions. These views are tailored towards the plug-in architecture of the Eclipse ecosystem. Thus, they support not only regular plug-ins as software composition mechanism, but also dynamic *extension-points*, through which a plug-in can permit other plug-ins to extend its functionality. Furthermore, they address the OSGi module system Eclipse is based on, as well as its service platform,[4] which offers an additional extensibility mechanism based on *services*.

The five views that we propose to offer insight in these extension mechanisms are the *Plug-in Modularization*, the *Extension Initialization*, the *Extension Usage*, the *Service Usage*, and the *Test Suite Modularization* views. They will be discussed in full detail in Section 4. To construct these views, we deploy a mixture of static and dynamic analysis.

To evaluate the applicability of these views, we discuss their application to three open source Eclipse plug-ins (each built from various plug-ins). We analyze the eGit plug-in system[5] permitting the use of the `git` versioning system within Eclipse, the substantial collection of plug-ins that comprises the Mylyn plug-in for work item management, and the Mylyn connector for the issue tracking system Trac.[6]

The paper is structured as follows. Section 2 provides the necessary background material on plug-in architectures. In Section 3, we present the findings of the interviews, which reveal the need for support during test suite understanding. Section 4 describes our approach, and covers the reconstructed architectural views. Section 5 discusses the

[3] `http://www.eclipse.org/mylyn`
[4] `http://www.osgi.org`
[5] `http://www.eclipse.org/egit`
[6] `http://wiki.eclipse.org/Mylyn_Trac_Connector`

architecture of our tool suite for reconstructing these views, after which we evaluate the views based on three case studies in Section 6. We reflect on the case study findings in Section 7, after which we conclude with a summary of related work, contributions, and areas for future research.

This paper is a substantially revised and extended version of an earlier paper [20]. The major changes include the addition of the analysis of information needs (Section 3), the addition of the service usage and test suite modularization views (Section 4), and a new case study based on Trac (Section 6).

2 Background: Modularization in Eclipse

Plug-in based dynamic modularization systems are widely used to create adaptive and configurable systems [4, 25, 27]. For Java, a well known example is OSGi,[7] which offers a service registry, life cycle management, and dynamic updating.

The Eclipse plug-in architecture[8] is built on top of OSGi, through the Equinox[9] implementation of the OSGi standard. Eclipse groups classes and packages into units, the so called plug-ins. Plug-in applications, like the well known Eclipse development environment, are composed from constituent plug-ins coming from different developers. We call the collection of all plug-ins forming a common application, including the plug-in architecture itself, a software ecosystem. A plug-in consists of code and meta data file, the manifest. The manifest describes, among others, the required and provided dependencies between plug-ins, and the plug-in version and author.

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

A next level of configurability is provided by means of the *extension* mechanism, illustrated in Figure 1. Plug-in A offers an extension-point, which is exploited by B to extend A's functionality. As an example, A could define a user-visible menu, and B would add an entry with an action to this menu.

An extension may be an executable extension contributing executable code to be invoked by the extended plug-in, a data extension, contributing static information such as help files, or a combination of both [36]. For executable extensions, a common idiom is to define a Java interface that the actual extension should implement, as shown in Figure 1.

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

Last but not least, the Eclipse platform also uses OSGi *services* to allow loosely coupled interactions. OSGi services are objects of classes that implement one or more

[7] `http://www.osgi.org`

[8] `http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.htm`

[9] `http://www.eclipse.org/equinox`

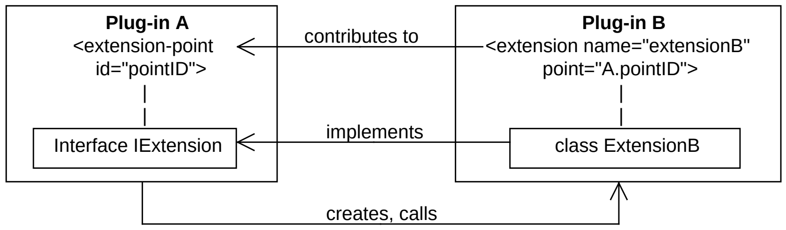


Fig. 1. The Eclipse plug-in extension mechanism

interfaces [37]. These services are registered in the service registry under their interface names. Other services can discover them by querying the service registry, e.g., for the specific interface name. The registry returns a reference which acts as pointer to the requested service object. The two main mechanisms to provide and acquire services are either programmatically via a call to the service registry, or via a dependency injection mechanism (i.e., declarative services).

Even though at the moment, extension-points and extensions are still the dominant extension mechanism, OSGi services are becoming more and more important in the Eclipse architecture. Especially the next Eclipse platform version, codename e4,[10] bets on services to solve the problem of tight coupling within the current Eclipse architecture. The e4 platform introduces a new programming model defining how plug-ins communicate beyond the extension mechanism. The introduced service programming models rely on three distinct parties, namely the service providers, service consumers, and a service broker. Using those, e4 defines a set of core services covering the main platform functionality.

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

3 Information Needs

In order to identify the information needs that developers have when working with plug-in test suites, we interviewed 25 Eclipse practitioners (henceforth 'Eclipsers'). The information needs that emerged from these interviews are described in Section 3.3.

These interviews were conducted in the context of a much larger study, aimed at identifying test practices adopted in plug-in architectures. The general findings of that

[10] `http://www.eclipse.org/e4`

study are published elsewhere [19], and are only briefly summarized in the present paper (Section 3.2).

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

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

Table 1. Domains, projects and/or companies involved in the interviews

3.1 Set-up Interviews

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

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

The study followed a *Grounded Theory* design, a research method from the social sciences aimed at distilling theories from documents, interviews, and other qualitative data [3, 5, 18]. Grounded theory is increasingly used in software engineering research [1], for example in the area of API documentation [8], reviewing in open source projects [32], and spreadsheet programming [22].

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)

3.2 Summary: Eclipse Testing Practices

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

On the other hand, interviewees express their belief that integration tests are especially essential for plug-in based systems. They report on technical and organizational barriers for performing integration, GUI, and system testing practices. The main challenges are long test execution times, immature test tooling or missing test infrastructure, high maintenance effort, as well as limited time for test design and execution [19].

Our interview results for Eclipse testing practices are also supported by literature. Gamma and Beck [16] provide best practices for testing Eclipse, and, thus, for plug-in-based architectures, in general. Their book emphasizes test-first development of plug-ins. It does not focus on integration testing of plug-in systems. Guidelines for testing Eclipse[11] emphasize unit testing as well as user interface testing for which capture-and-playback tools are used.

The literature addressing OSGi testing focuses on the provisioning of the infrastructure required during the set-up of integration tests [35]. We have not been able to find test strategies for OSGi targeting integration testing of dynamic modularization systems in general, or plug-in systems in particular. Literature many Eclipsers are aware of and mentioned in the interviews is for example the book “Clean Code” by Martin [26], which propagates the importance of clean and structured test code.

3.3 Test Suite Understanding Needs

During the interviews we asked the participants how easy or difficult the task of understanding test suites is, and which information would facilitate the task. Participants described two main scenarios to interact with the test code (i.e. understanding individual test cases and test suites), each implying different information needs. In the following sections, we will discuss the identified information needs and present excerpts of the interviews. A summary of all nine identified information needs (referred to as $N_1 - N_9$) is presented in Table 3. We will use those identifiers in the remaining of the paper to refer to the information needs.

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

[11] `http://wiki.eclipse.org/Eclipse/Testing`

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

But also the format of a test case must be well structured to facilitate understanding of test code, as P17 reports: *"Tests have to be written similar to a work specification, like click here, execute that and check the result. And it should not be like 300 lines of test code. Then, nobody understands what's going on. More like a step by step description, and it's important to have good test names."* P18 explains: *"We have a standardized form for test cases, and also naming conventions, that helps a lot. We also write tests in form of Given-When-Then, like described in the book 'Clean Code'[12] [26]".*

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

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

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

Understanding Test Suites

Challenges. The second scenario involves comprehending the complete test suite in order to be able to assess quality and coverage of test scenarios. To master this task, developers need to understand which part of the system under test are not covered (N_4),

[12] This style is originally from domain-driven design [12].

which can be challenging as P14 explains: *"What one specific test does, that's quite easy to understand. What's difficult is to see where the blank spots on the map are."*

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 3. Distilled information needs

4 Models for Understanding Plug-in Test Suites

The interviews just presented demonstrate that Eclipse developers indeed face a test suite understanding problem. This problem can be partially addressed by regular support for program comprehension (such as dynamic analysis, [7] software architecture reconstruction [10], or reengineering patterns [9]) as well as methods aimed at understanding unit test suites [6, 11, 29, 33]. Furthermore, guidelines on how to set up (JUnit) test suites by, e.g., Martin [26], Feathers [13] or Freeman and Pryce [14], will help to avoid and resolve overly complex test suites.

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

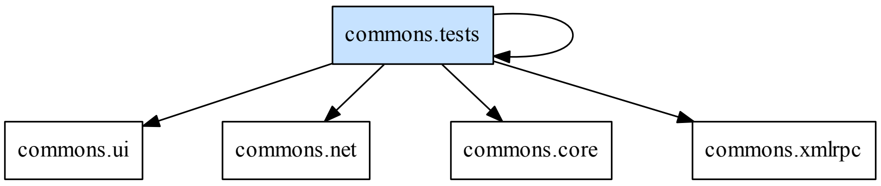
The goal of the first view, the *Plug-in Modularization View* is to provide such structural and organizational awareness with respect to the code-dependencies of plug-ins. Equipped with this basic structural knowledge, the second step is the analysis of the extension relations between plug-ins and the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

In this section we present these views, state their goal, and formulate the information needs they address. In terms of the Symphony software architecture reconstruction process [10], for each view we distinguish a *source model* corresponding to the raw data we collect, a *target model* reflecting the view that we eventually need to derive, as well as mapping rules between them. In what follows we present a selection of the meta-models for the source and target models involved, as well as the transformation between them.

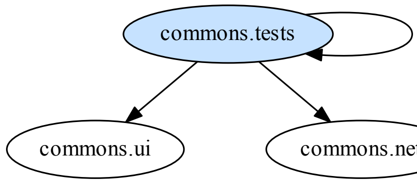
4.1 The Plug-in Modularization View

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

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



(a) Static dependencies



(b) Dynamic dependencies

Fig. 2. Static and dynamic dependencies of test-component "commons.tests" in Mylyn

Figure 2 illustrates this view for the test-component *commons.tests* of Mylyn showing its static (2(a)) and dynamic code-dependencies (2(b)). On the left we see that *commons.tests* statically depends on four other plug-ins. The dynamic representation on the right side, reveals that only two out of those four plug-ins are actually exercised in a test run. It does not explain why this is the case (reasons could be that the test suite requires manual involvement, or that a different launch configuration should be used), but it steers the investigation towards particular plug-ins.

4.2 Extension Initialization View

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

By means of this perspective, we will not only be able to tell which extensions and extension-points are tested in the current test suite (N_3), but we also gain insight in the system under test and its extensibility relations. For example, keeping track of which extensions are initialized during a test run is an information need expressed by P6 (N_9), which can be satisfied by this view. The meta model of this view is illustrated in Figure 3, by means of a UML class diagram.[13] The view contains plug-ins, the extensions and extension-points they provide, as well as test methods that initialize the extensions. Extension initialization is the process of activating an extension (i.e. loading its classes). This differs from using an extension which means invoking a method of its classes.

The view helps answering questions on extensions and the way they are tested at system, plug-in, and test method scope. The main focus of the view is revealing which

[13] Drawn using UMLet version 11.3 (see `http://www.umlet.com/`)

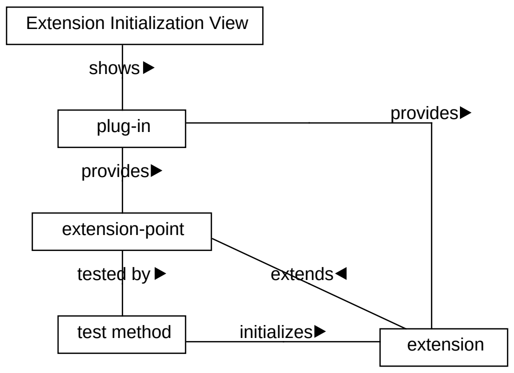


Fig. 3. Meta model of the Extension Initialization View

plug-ins in the system under test influence the functionality of each other through the extension mechanism, and which of those influencing relations are activated by a test suite.

System Scope. At system scope, the view gives insight in the extension relations present in the system under test, i.e., which plug-in contributes to the functionality of another plug-in. This is visualized in one graph, as shown in Figure 12 for one of our case studies. The graph presents the overall contributions of the systems, i.e., all extension-points and extensions within the system under test. In case plug-in A declares an extension-point and plug-in B provides an extension for it, the graph shows a link between the two nodes. The labels (fractions) on the links represent the number of statically declared extensions (denominator) one plug-in provides for the other, and the number of extensions that are actually used during a test run (numerator).

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

This can be visualized, e.g., by means of a graph. An example is given in Figure 13, again for our Mylyn case study. The graph presents all involved plug-ins as ellipse-shaped nodes. Extension-points are represented as rectangles. Relations between an extension-point and a plug-in providing an extension are presented as edges. Extensions that are actually used during the test run are filled with a color. Thus, Figure 13 shows that 5 extensions are actually used, but that extension *tasks.ui* is not used. The view can also be used to show all extensions declared by the system under test for extension-points inside and outside the system under test. This means the view shows how the system under test influences the ecosystem during a test run, as shown in Figure 15.

Test Method Scope. At method scope, the developer can observe which test methods have invoked the code of an extension-point responsible for loading extensions, and which extensions have been created for it. For example, from Figure 14, the developer knows that test method “testShadowsStructureBridge()” triggers extension-point “mylyn.context.core.bridges” to load all available extensions. In this way, a developer or tester can identify the location of the test-code for a particular extension-point.

Underlying Meta-Models This view is based on static meta data and dynamic trace information. The meta data comes from the mandatory XML file, and from the optional XML-schema file (see Section 2).

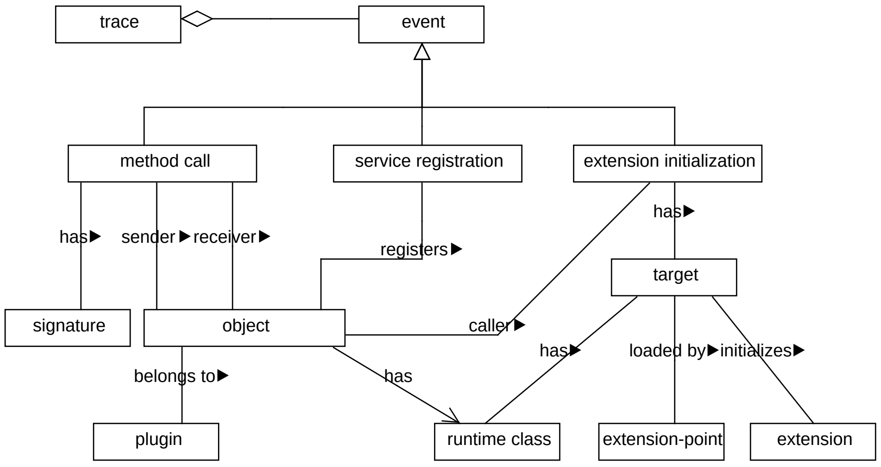


Fig. 4. Trace meta model

The trace used for this view comprises “extension initialization events” during the test run. The underlying trace data follows the meta model shown in Figure 4, which is also used to derive dynamic information for the other views. An “extension initialization event” is recorded before a method named “createExecutable()” is called. In the Eclipse Platform, this method is used to create the extension from a given class, passed as parameter. This also is the point we intercept to trace the caller of this method and the target-object, by means of an aspect.

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

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

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

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

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

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

4.3 Extension Usage View

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

The target meta model of the *Extension Usage View* is illustrated in Figure 5. In this view, extensions are referenced by their name. Extensions are furthermore related to the extension-points they target, and to the test methods exercising them. Recall from Figure 1 that extension-points can declare types (interfaces or classes) that are implemented by actual extension classes.

The *Extension Usage View* can be used at system, extension, and usage level. On *system scope*, we can gain detailed information about which of the declared extensions have been actually used during a test run, and how many of the test methods are associated with extension usages. Using an extension means to invoke a method of the extension class, overwritten or inherited by the type declared at the extension-point. For example, from Figure 16 we can see a list of extensions that have been used during the test run (left side).

Zooming in to the *extension scope*, the developer can see which test methods have used a given extension. For example, on the right side of Figure 16, we can see that the extension "mylyn.tasks.ui" has been used during the execution of four test methods. This information is helpful to spot the right piece of code responsible for the extension usage, e.g., to enhance or change it.

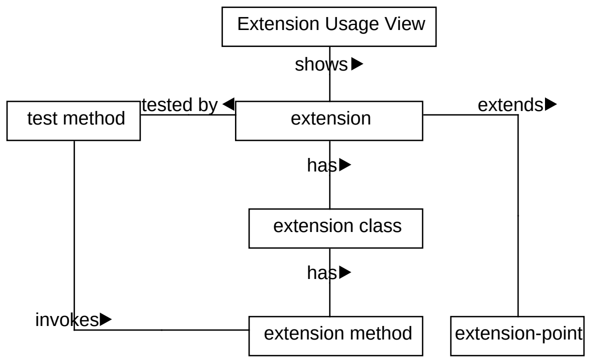


Fig. 5. Meta Model of the Extension Usage View

A refinement of this view to the *method scope* shows how the extension has been used during the test run, for example illustrated by the pop-up window in Figure 16. Here, all methods of an extension that have been called during testing are listed.

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

Underlying Meta-Models The execution trace used to construct the *Extension Usage View* is the same as the one used for the initialization view. It comprises detailed method calls of a test run, as shown in Figure 4.

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

Reconstructing the View To construct this view, we need in addition to the dynamic data discussed before, all the methods an extension can implement. Those methods can be used by an extension-point to invoke it. We will refer to this set as to the *extension method set*. Therefore, the extension-point has to define a specific base type (e.g. a class or an interface), which can be extended by an extension. To give a simple example, let us look at Listing 1.1. Here, the class *Extension* represents the base type of an extension. This class is defined within the plug-in providing the extension-point. Also within the plug-in defining the extension-point, the code is located which is in charge of invoking all loaded extensions, as illustrated by the method *invokeExtensions()* of class *ExtendsionUsage*. A plug-in which wants to provide an extension for this extension-point has to extend the base class, as done by classes *B* and *C*. Those classes can be part of another plug-in.

Listing 1.1. Extension Usage Example

```
abstract class Extension{
  void me();
}

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

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

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

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

First, in case the extension-point formally declares a base type for an extension, the algorithm uses this to derive recursively all methods defined by it and its super-types, i.e., interfaces and ancestors. This collection represents the extension method set. For our example in Listening 1.1, the method set comprises just method *me()*.

In the case no base type is provided, the algorithm collects all the classes a given *extension* declares from its meta data file. Starting from these types, the algorithm recursively derives all super-types of these classes. Note, however, that not all of them might be visible to the extension-point. For example, consider a class *A*, defined in plug-in *Pa*, that extends class *E*, defined in plug-in *Pe* and implements Interface *I* also defined in *Pa*. Since no declaration of a base class is provided, the algorithm has to decide whether *A* is based on *I* or *E*. This example is illustrated in Figure 6.

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

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

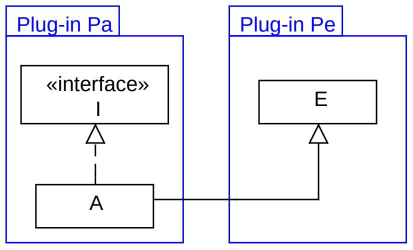


Fig. 6. Deriving an Extension Base Type

Applying this algorithm to the example of Listing 1.1 reveals that, in case class *B* is defined within another plug-in, method *mb()* will not be visible to the extension-point, and is therefore excluded from the extension method set. In case class *B* is defined within the plug-in defining also the extension-point the algorithm will declare class *B* as a potential extension class and include methods *me()* and *mb()* in the extension method set.

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

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

4.4 Service Usage View

The Eclipse plug-in architecture builds on top of OSGi. Especially in the new e4 version of the Eclipse platform, OSGi services are an important extensibility mechanism. Services are a means to decouple plug-in interactions, and allow interchangeability of service providers, which also improves testability of the system. The *Service Usage View* helps in understanding which services are used during test execution (N_3). It helps answering questions like “which plug-ins acquire or register which services?”, and “how are these services used during test execution?” The meta model of this view is illustrated in Figure 7. A concrete example of this view for the usage of the service “IProxyService” is given in Figure 17. From this view, it is apparent that this service was used during the execution of five test methods.

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

Underlying Meta-Models OSGi services can be registered and obtained either programmatically (see Listing 1.2 for some examples) or by using dependency injection defining services in an XML-file (i.e., declarative services). To obtain the static data

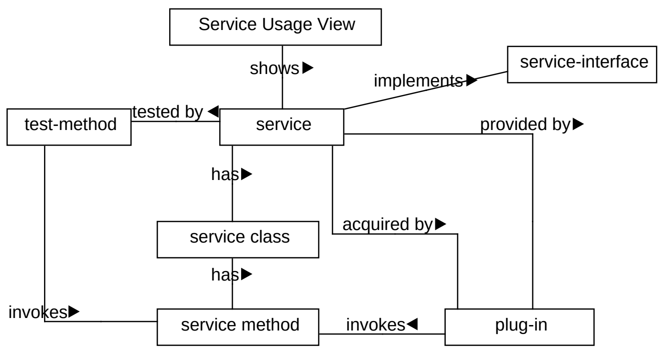


Fig. 7. Meta Model of the Service Usage View

representing service registration and acquisition, we analyze the plug-in's byte code for service registration or service acquisition, as well as the meta data for the use of declarative services.

The dynamic data required is provided by two execution traces. First, one trace covers method call events, as described in Section 4.3. Second, service registration events, as illustrated in Listing 1.2, are traced.

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

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

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

4.5 The Test Suite Modularization View

For complex Eclipse plug-ins, it is common practice to assemble individual test cases into different test suites. Examples are suites for specific versions of required plug-ins or external services, fast test cases run during development, more time consuming test

Listing 1.2. Excerpt programmatic service registration and acquisition

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

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

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

cases depending on the user interface (workbench) or network access, and so on. These suites are often assembled programmatically, and sometimes involve the creation of different fixtures in order to run the same test cases under different circumstances.

The *Test Suite Modularization View* aims at clarifying how test cases are grouped into (nested) test suites at run time. It maps assembled test suites to (1) the test plug-ins that contribute test cases; and (2) plug-ins under test. It helps answering questions like "which plug-ins contribute tests to this particular test suite?", "which plug-ins are tested by this test suite?", and "which extensions, extension-points and/or services get addressed by this test suite?". The meta model of this view is illustrated in Figure 8. This view helps the developer to choose the right test suite to execute, to understand which entities are tested by a particular test suite, or to assemble a new, customized test suite addressing the right plug-ins of the system, and satisfies information needs N_3, N_7 and N_8 expressed in Section 3.3.

Underlying Meta-Models This view is based on static meta data and dynamic trace information. The meta data comes from the plug-in manifest files of the plug-ins, the mandatory XML file for extension and extension-point definition, from the optional XML-schema file (see Section 2), the XML-definitions for declarative services, as well as from the analysis of the byte code for service registration or acquisition.

The dynamic data comes from two traces. First, a trace comprising method calls during the test run, and second, the trace comprising "service registration events" as illustrated by the trace meta model in Figure 4.

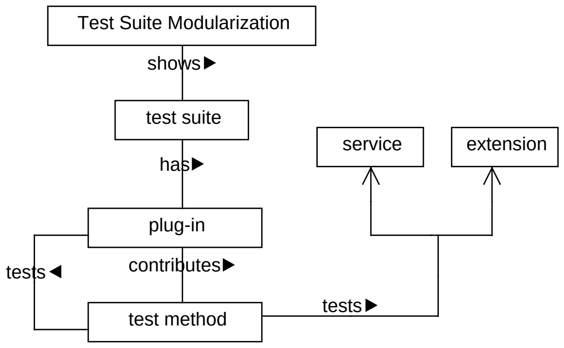


Fig. 8. Meta Model of the Test Suite Modularization View

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

5 Implementation and Tool Architecture

We implemented the reconstruction and presentation of our views in ETSE,[14] the *Eclipse Test Suite Exploration Tool*. It is implemented in Java, offers an API to construct the views in question, and a graphical user interface which is implemented as Eclipse extension, which integrates the tool in the Eclipse IDE.

ETSE consists of three logical modules: a module dedicated to information gathering, a module responsible for knowledge inference and a module responsible for the presentation of the views, as shown in Figure 9.

Module: Information Gathering. This module is responsible for gathering static meta data as well as for gathering execution traces during test execution. To analyze the static Java code we use the Byte Code Engineering Library,[15] which inspects and manipulates the binary Java class files. Meta data, including the OSGi manifest, the service definitions and the plugin.xml files, is collected and analyzed. The user can instruct ETSE which test suite and which system under test should be examined by using the "Configuration tab" provided by ETSE's user interface. To trace the execution of the test run, we use aspect-oriented programming, in particular the AspectJ[16] framework. Because

[14] ETSE is available at `http://swerl.tudelft.nl/bin/view/Main/ETSE`
[15] `http://jakarta.apache.org/bcel`
[16] `http://www.eclipse.org/aspectj`

we do not want to intercept plain Java applications, but Equinox, the Eclipse implementation of OSGi, we are using the Equinox aspects framework.[17] This provides load time-weaving of advices, i.e., when a class is loaded by OSGi. There are four main groups of aspects that can be differentiated: the aspect used for weaving into the initialization of the extensions, the aspect used to trace method calls, the aspect used to trace plug-in starts and stops, and the aspect used to trace registration and acquisition of OSGi services. All the analyzed data is finally stored as source views, in a format similar to the meta model illustrated in Figure 4, in a repository.

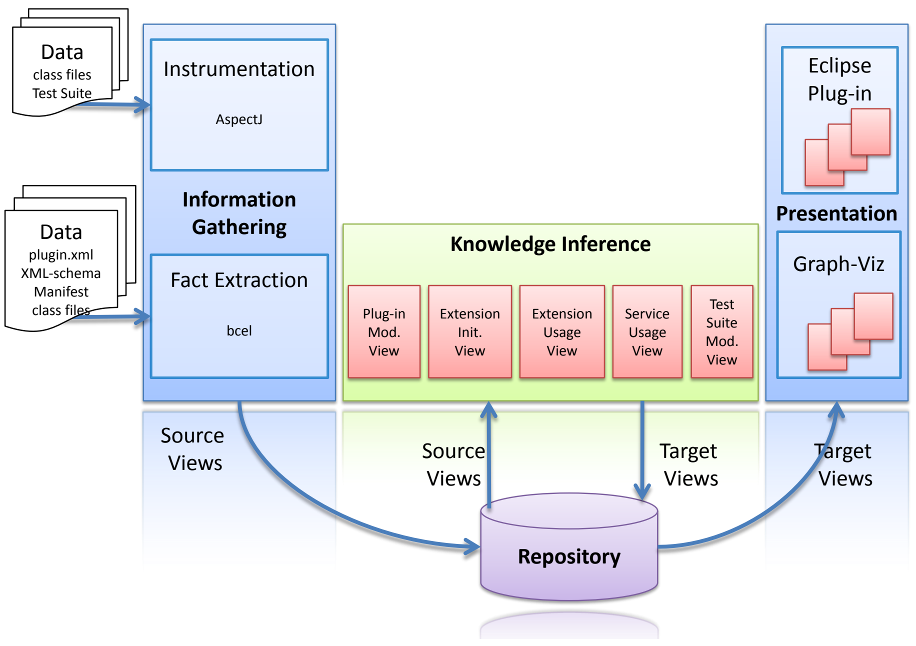


Fig. 9. ETSE Architecture

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

Module: Presentation The presentation module is used to visually represent the target views to the user of ETSE. Target views can be saved as comma separated value

[17] `http://www.eclipse.org/equinox/incubator/aspects/equinox-aspects-quick-start.php`

files, which a user can later visualize ad libitum. Also ETSE can visualize those files for the user. First, ETSE allows users to store the target views in the dot-graph format, which then can be visualized by Graphviz,[18] a graph visualization package. Furthermore, ETSE is integrated in the Eclipse IDE, where it provides a graphical user interface allowing the user to interact easier with the tool. Within this paper we show visualizations based on dot-graphs, as well as visualizations rendered by ETSE's user interface within Eclipse. A screenshot of the ETSE Eclipse integration showing the *Extension Initialization View* is provided in Figure 10. Users can for example navigate between views, or define different level of detail for each view, as detailed below.

Navigation between scopes ETSE presents each *architectural view* in a separate tab within an Eclipse "view". The user can easily switch between the *architectural views* by activating the desired tab. Within each tab, the user can navigate between the several scopes discussed in this paper (i.e., system, plug-in, extension, service or method scope). For example, in the *Extension Initialization View*, the user can switch between the plug-in or the system scope by activating a radio button. In the *Extension* and *Service Usage View*, the user will first see a list of all the extensions respectively service that have been used during a particular test run on the left side (i.e. system scope). By selecting an extension/service from the list, all test methods which have triggered a use of that particular extension/service are shown on the right side (i.e. extension resp. service scope). The user can further zoom in on method scope by clicking on a particular test method. This will cause a pop-up window to appear and to show which methods of the selected extension/service have been used during execution of the selected test method. All scopes of this view are illustrated in Figure 16. Further, the user can also request to see the source code of the test method by left-clicking on the test method. Then, the Java class comprising the test method is opened and visualized within the editor.

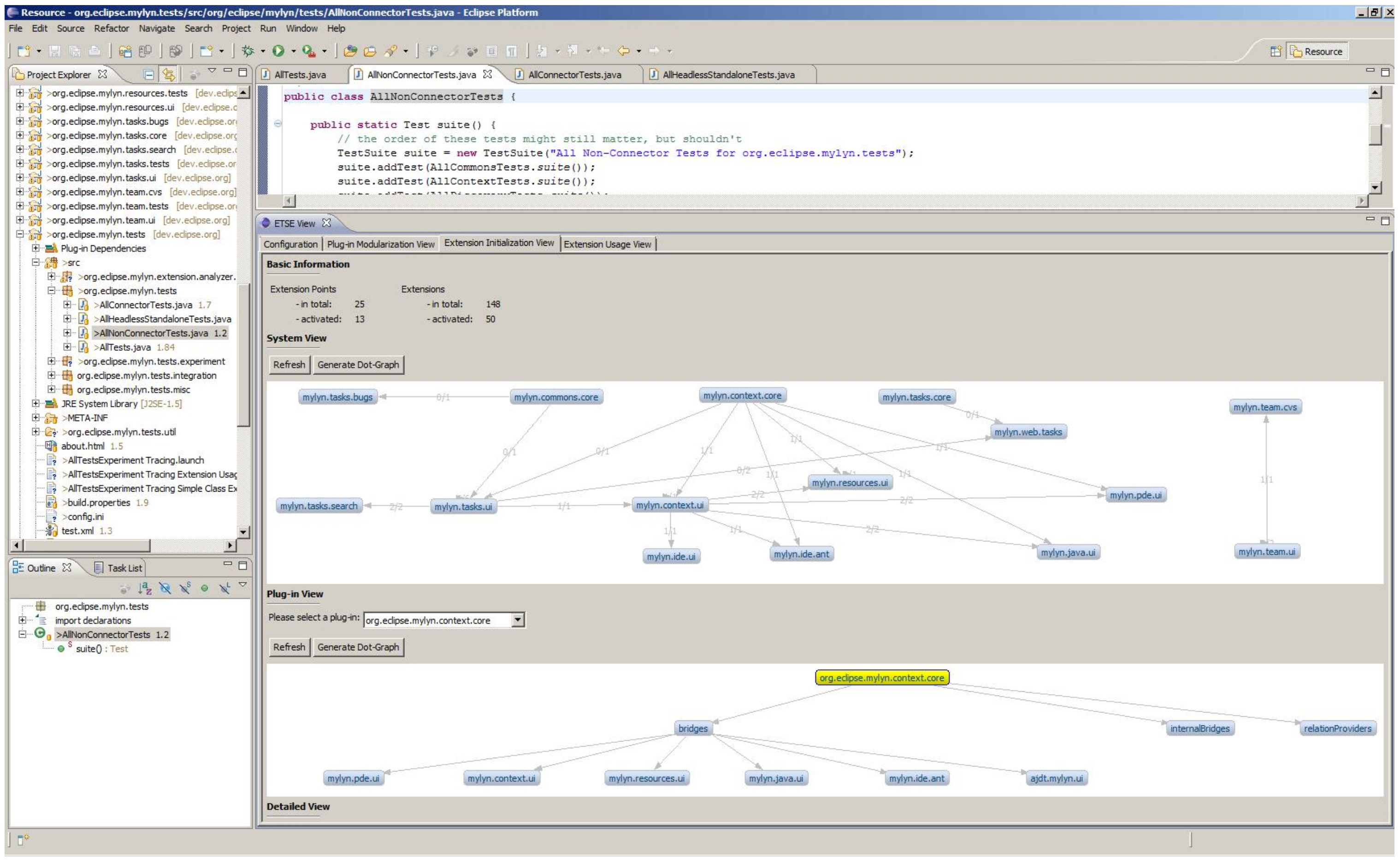
6 Evaluation

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

RQ1: *Which information about the test suite and system under test can be obtained by the proposed views and to which extent does the information provided by the tool address the information needs identified?*
RQ2: *To what extent do the views scale to large systems?*
RQ3: *To what extent are the views a correct representation of the system under test?*

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

[18] http://www.graphviz.org


```java
public class AllNonConnectorTests {

    public static Test suite() {
        // the order of these tests might still matter, but shouldn't
        TestSuite suite = new TestSuite("All Non-Connector Tests for org.eclipse.mylyn.tests");
        suite.addTest(AllCommonsTests.suite());
        suite.addTest(AllContextTests.suite());
```

Basic Information

Extension Points		Extensions	
- in total:	25	- in total:	148
- activated:	13	- activated:	50

Fig. 10. Screenshot of the ETSE Eclipse integration showing the *Extension Initialization View*

6.1 The Subject Systems

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

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

The last subject system is a Mylyn connector for the issue tracking system Trac. We choose the Trac-Mylyn connector for two reasons: First, it is of 8,500 lines of code and 3,400 lines of test code, a quite small, but well-tested plug-in system that permits, in addition to the investigation by means of the views, manual inspection of the complete system. Second, we choose it because it is referred to in the Mylyn Contributor Reference[19] as the system to look at, in case a new connector for Mylyn should be developed. The Trac-Mylyn connector consists of three plug-ins and one test plug-in.

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

6.2 RQ1: Applicability and Information Needs

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

Answering RQ1: In summary, the five proposed views satisfy many of the information needs identified concerning test suite understanding. They can help to understand

[19] `http://wiki.eclipse.org/Mylyn/Integrator_Reference#Creating_connector_projects`

and investigate test code and the system under test from a top-down approach. The views help to understand what (i.e. plug-ins, extension-points, extensions and services, as well as their methods) has been tested (N_3), and what has been left out (N_4). They provide an overview of the test suites (N_5), highlight the integration with other plug-ins (N_6), shed light on the test organization (N_7) and the configuration of the test execution environment (N_9), and help to locate test code (N_8). On the other hand, the views are not suited to investigate the system from a bottom-up approach, i.e. start with a single test case. Information needs such as understanding source code (N_1) or the reasons behind tests (N_2) are not covered by these views. The relations between views and information needs are summarized in Table 4.

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

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

Table 4. Distilled information needs: Plug-in Modularization View (PMV), Extension Initialization View (EIV), Extension Usage View (EUV), Service Usage View (SUV), Test Suite Modularization View (TMV)

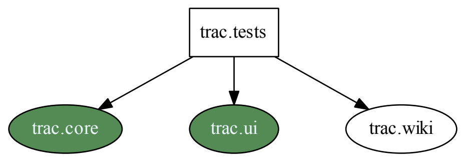


Fig. 11. *Plug-in Modularization View* for Trac

Plug-in Modularization View The *Plug-in Modularization View* aims at providing a first high-level overview of dependencies at the top-most level of plug-ins. An example of a plug-in modularization view was shown in Figure 2 for the Mylyn test component. It shows the four plug-ins the Mylyn test component statically depends on, as well as the two that are actually executed during testing. A similar view is shown in Figure 11, displaying which of the Trac plug-ins are involved in testing. These views highlight which plug-ins a test plug-in depends on statically and also which of those are actually invoked during a test run (N_3, N_4). This information can be valuable to understand the structure and organization of test as well as production code (N_7). Structure of test and production code plays an role during test suite understanding (see Section 3.3).

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

This view at system scope for Mylyn is illustrated in Figure 12. ETSE allows to removed common prefixes from the plug-in names to improve readability, as we did with "org.eclipse.mylyn". An edge between two plug-ins means that one plug-in declares an extension-point for which the other plug-in provides an extension. Only plug-ins with actual extension relations, which means that a plug-in exists providing an extension-point and another one using it, are shown, reducing the number of nodes to 15. From this representation of the system, it is apparent which plug-ins influence each other, and also which of those relations have been actually addressed by the test suite (N_3), and which have been left out (N_4). The view abstracts from the specific extension-points declared. The fraction on the edge states how many of the static declared extensions (denominator) are activated during a test run (numerator).

At plug-in scope, this view for plug-in *mylyn.context.core* is illustrated by Figure 13.[20] The plug-in provides three extension-points, namely *bridges*, *internalBridges* and *relationProviders*. The view shows that within Mylyn six plug-ins exist that use

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

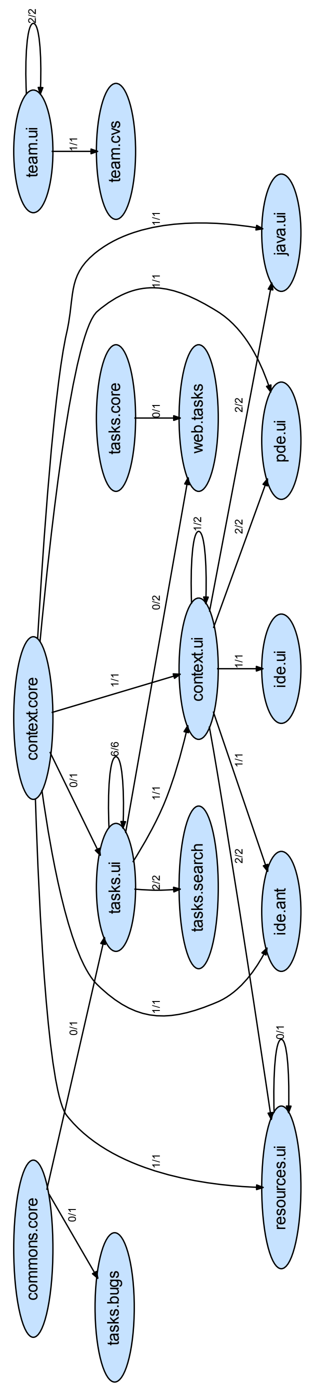


Fig. 12. Extension Initialization View on system scope showing static and dynamic dependencies based on extension-points

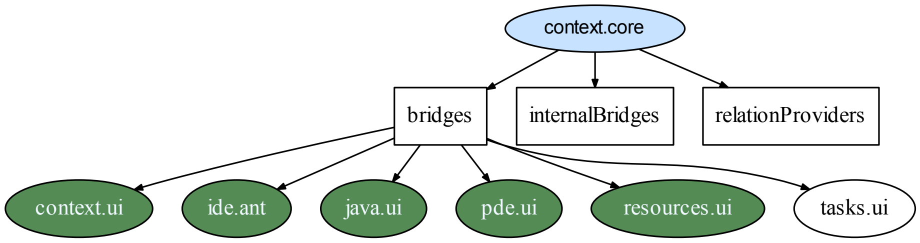
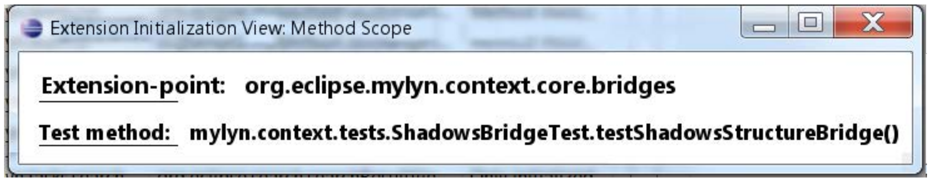


Fig. 13. Extension Initialization View on plug-in scope based on extension-points

extension-point *bridges* to influence the plug-in, represented by the six nodes connected to this extension-point. The coloring of five nodes indicates that only five of the relations are activated during the test run. The view does not give explanations, but points to one plug-in the developer might manually inspect and find an empty XML declaration for this extension.

The developer can also zoom at method scope, as illustrated by Figure 14. This view reveals which test method causes this extension-point to load its extensions, and can be used to locate test code (N_8).


Extension Initialization View: Method Scope

Extension-point: org.eclipse.mylyn.context.core.bridges

Test method: mylyn.context.tests.ShadowsBridgeTest.testShadowsStructureBridge()

Fig. 14. Extension-Initialization View on test method scope

The *Extension Initialization View* serves to show how plug-ins affect each other's behavior. The present view does not show how the system under test is influenced by its ecosystem, i.e., the Eclipse platform. Nevertheless, the borders defining the system under test can be chosen by the viewer, thus allowing the developer to include parts of Eclipse he or she is interested in. Also for smaller systems, like the Trac connector, this view is helpful, as it shows how the system enhances (extends) the ecosystem during the test run. For example, Figure 15 shows that plug-in "trac.ui" has four extensions enhancing the Eclipse ecosystem, out of which two extensions are initialized during this test run, and one extension is a data extension.

The *Extension Initialization View* visualizes information about the integration of plug-ins within the system under test (N_5, N_6), the coverage of these integration points during test execution (N_3, N_4), as well as about the configuration of the test environment (N_9). This covers several information needs about *plug-ins and extensions* raised in Section 3.3. For example, this view can answer the question of P6, who wants to know which plug-ins in the system under test can influence the functionality of other

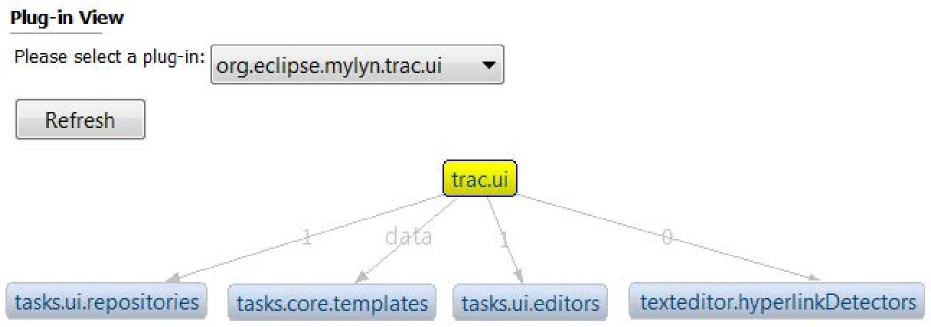


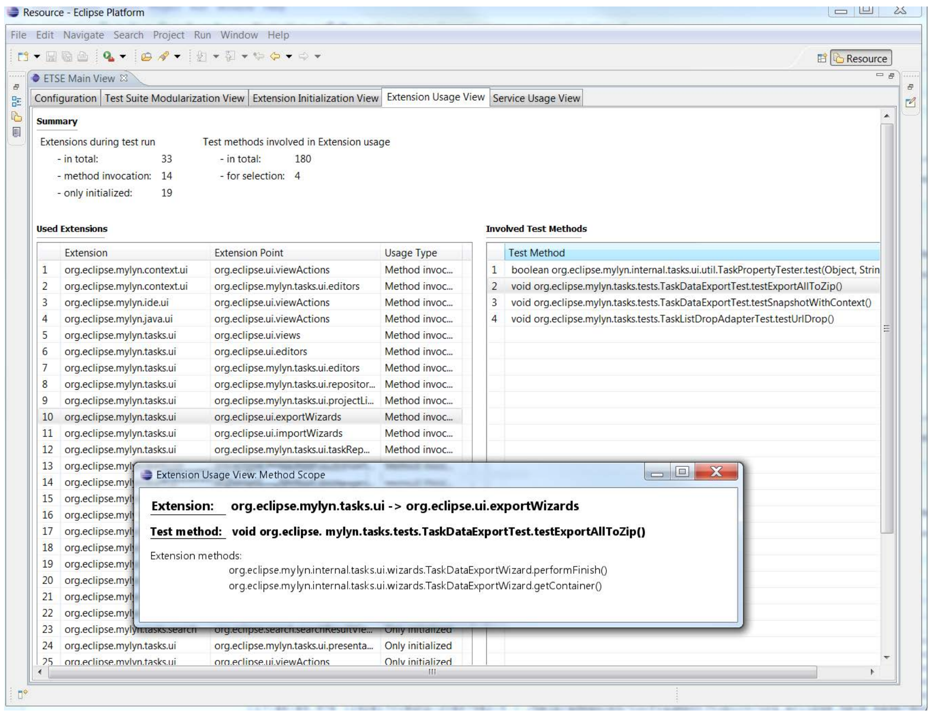
Fig. 15. Extension Initialization View for extensions of Trac on plug-in scope

plug-ins. On *method scope* this view also allows to locate test code addressing (N_8) a given extensions and thus facilitating understanding of *test organization and structure*.

Extension Usage View The Extension Usage View helps to understand how the test suite addresses integration with respect to extensions. It gives an overview of how many extensions have been used during the test run (N_5), and whether extensions have been only initialized or whether there has been a method invocation (N_3, N_4). For example, from Figure 16, the developer can see that in total 32 extensions have been used during this test run, whereby 18 extensions have been just initialized, and for 14 methods of these extensions have been invoked. 172 test methods have caused a usage of an extension during this test run. On a detailed level, this view allows to locate the test code related to an extension (N_8). In Figure 16, the developer clicks on an extension of interest, for example on extension of plug-in "mylyn.tasks.ui" for the extension-point *exportWizards*, and on the right hand side, the view shows all four test methods that trigger a method invocation of this extensions.

Further, the view sheds light on the structural testing approach followed by this test suite, e.g., how many of the methods of an extension have been used and which have been left out (N_3, N_4). The developer can see this information by double clicks on an extension of interest. For example in Figure 16, the developer can see from the pop-up window that method "performFinish()" and method "getContainer()" of the extension for *exportWizards*, declared in plug-in "mylyn.tasks.ui", have been invoked by the test method "testExportAllToZip()". Such structural information about the coverage of extensions and their methods by particular test suites can help the developer to identify "blank spots" (i.e. untested functionality) as expressed as an information need by developer P14 in Section 3.3.

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


Resource - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Resource
ETSE Main View
Configuration
Test Suite Modularization View
Extension Initialization View
Extension Usage View
Service Usage View

Summary

Extensions during test run
- in total: 33
- method invocation: 14
- only initialized: 19

Test methods involved in Extension usage
- in total: 180
- for selection: 4

Used Extensions

	Extension	Extension Point	Usage Type
1	org.eclipse.mylyn.context.ui	org.eclipse.ui.viewActions	Method invoc...
2	org.eclipse.mylyn.context.ui	org.eclipse.mylyn.tasks.ui.editors	Method invoc...
3	org.eclipse.mylyn.ide.ui	org.eclipse.ui.viewActions	Method invoc...
4	org.eclipse.mylyn.java.ui	org.eclipse.ui.viewActions	Method invoc...
5	org.eclipse.mylyn.tasks.ui	org.eclipse.ui.views	Method invoc...
6	org.eclipse.mylyn.tasks.ui	org.eclipse.ui.editors	Method invoc...
7	org.eclipse.mylyn.tasks.ui	org.eclipse.mylyn.tasks.ui.editors	Method invoc...
8	org.eclipse.mylyn.tasks.ui	org.eclipse.mylyn.tasks.ui.repositor...	Method invoc...
9	org.eclipse.mylyn.tasks.ui	org.eclipse.mylyn.tasks.ui.projectLi...	Method invoc...
10	org.eclipse.mylyn.tasks.ui	org.eclipse.ui.exportWizards	Method invoc...
11	org.eclipse.mylyn.tasks.ui	org.eclipse.ui.importWizards	Method invoc...
12	org.eclipse.mylyn.tasks.ui	org.eclipse.mylyn.tasks.ui.taskRep...	Method invoc...
13	org.eclipse.myl		
14	org.eclipse.myl		
15	org.eclipse.myl		
16	org.eclipse.myl		
17	org.eclipse.myl		
18	org.eclipse.myl		
19	org.eclipse.myl		
20	org.eclipse.myl		
21	org.eclipse.myl		
22	org.eclipse.myl		
23	org.eclipse.mylyn.tasks.search	org.eclipse.search.searchResultVie...	Only initialized
24	org.eclipse.mylyn.tasks.ui	org.eclipse.mylyn.tasks.ui.presenta...	Only initialized
25	org.eclipse.mylyn.tasks.ui	org.eclipse.ui.viewActions	Only initialized

Involved Test Methods

	Test Method
1	boolean org.eclipse.mylyn.internal.tasks.ui.util.TaskPropertyTester.test(Object, Strin
2	void org.eclipse.mylyn.tasks.tests.TaskDataExportTest.testExportAllToZip()
3	void org.eclipse.mylyn.tasks.tests.TaskDataExportTest.testSnapshotWithContext()
4	void org.eclipse.mylyn.tasks.tests.TaskListDropAdapterTest.testUrlDrop()

Extension Usage View: Method Scope

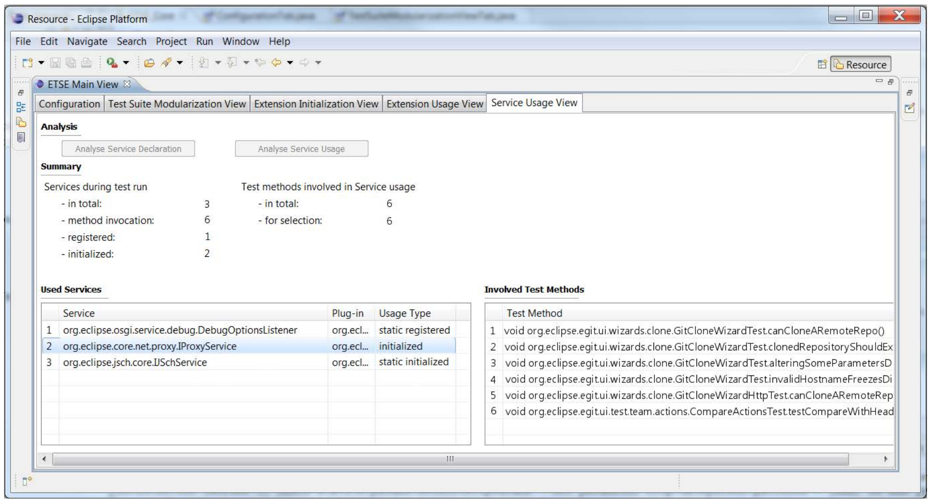
Extension: **org.eclipse.mylyn.tasks.ui -> org.eclipse.ui.exportWizards**

Test method: **void org.eclipse. mylyn.tasks.tests.TaskDataExportTest.testExportAllToZip()**

Extension methods:
org.eclipse.mylyn.internal.tasks.ui.wizards.TaskDataExportWizard.performFinish()
org.eclipse.mylyn.internal.tasks.ui.wizards.TaskDataExportWizard.getContainer()

Fig. 16. Extension Usage View showing the system scope (left side), the extension scope (right side) and the method scope (pop-up) for Mylyn

In the Service Usage View, illustrated in Figure 17, the developer sees that only the service *"IProxyService"* is used during the test run, in six different test methods. If the developers zooms in further, the view reveals which methods of the service have been invoked during this test run (N_3) and which ones have not (N_4).


Resource - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Resource
ETSE Main View
Configuration
Test Suite Modularization View
Extension Initialization View
Extension Usage View
Service Usage View
Analysis
Analyse Service Declaration
Analyse Service Usage

Summary

Services during test run
- in total: 3
- method invocation: 6
- registered: 1
- initialized: 2

Test methods involved in Service usage
- in total: 6
- for selection: 6

Used Services

	Service	Plug-in	Usage Type
1	org.eclipse.osgi.service.debug.DebugOptionsListener	org.ecl...	static registered
2	org.eclipse.core.net.proxy.IProxyService	org.ecl...	initialized
3	org.eclipse.jsch.core.IJSchService	org.ecl...	static initialized

Involved Test Methods

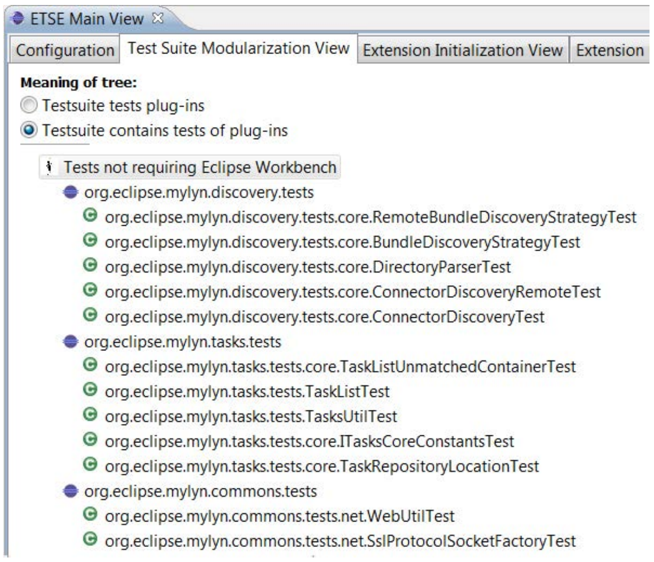
	Test Method
1	void org.eclipse.egit.ui.wizards.clone.GitCloneWizardTest.canCloneARemoteRepo()
2	void org.eclipse.egit.ui.wizards.clone.GitCloneWizardTest.clonedRepositoryShouldEx
3	void org.eclipse.egit.ui.wizards.clone.GitCloneWizardTest.alteringSomeParametersD
4	void org.eclipse.egit.ui.wizards.clone.GitCloneWizardTest.invalidHostnameFreezesDi
5	void org.eclipse.egit.ui.wizards.clone.GitCloneWizardHttpTest.canCloneARemoteRep
6	void org.eclipse.egit.ui.test.team.actions.CompareActionsTest.testCompareWithHead

Fig. 17. Service Usage View on system and service scope for eGit

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

The plug-in systems investigated in this study still make limited use of services, mainly because they are designed to work with Eclipse 3.5-3.7. In future, we anticipate that the usage of services within Eclipse will increase drastically, especially with introduction of the new version of Eclipse, e4. In e4, the main platform API, known as the "twenty things"[21] are provided by services. More and more Eclipse plug-in projects get ready for the transition to the new technologies in e4. At the moment, developers are not concerned with services in association with test suite understanding, as they only use few or none of them. On the other hand, they are well aware of their future importance and the evolution towards adoption. When adoption raises, structural information about the coverage of services and their methods by particular test suites will be as valuable to identify untested functionality, or scenarios, as this information for extensions.

[21] `http://www.eclipse.org/e4/resources/e4-whitepaper.php`


ETSE Main View

Configuration | Test Suite Modularization View | Extension Initialization View | Extension

Meaning of tree:

○ Testsuite tests plug-ins

◉ Testsuite contains tests of plug-ins

```
Tests not requiring Eclipse Workbench
  org.eclipse.mylyn.discovery.tests
    org.eclipse.mylyn.discovery.tests.core.RemoteBundleDiscoveryStrategyTest
    org.eclipse.mylyn.discovery.tests.core.BundleDiscoveryStrategyTest
    org.eclipse.mylyn.discovery.tests.core.DirectoryParserTest
    org.eclipse.mylyn.discovery.tests.core.ConnectorDiscoveryRemoteTest
    org.eclipse.mylyn.discovery.tests.core.ConnectorDiscoveryTest
  org.eclipse.mylyn.tasks.tests
    org.eclipse.mylyn.tasks.tests.core.TaskListUnmatchedContainerTest
    org.eclipse.mylyn.tasks.tests.TaskListTest
    org.eclipse.mylyn.tasks.tests.TasksUtilTest
    org.eclipse.mylyn.tasks.tests.core.ITasksCoreConstantsTest
    org.eclipse.mylyn.tasks.tests.core.TaskRepositoryLocationTest
  org.eclipse.mylyn.commons.tests
    org.eclipse.mylyn.commons.tests.net.WebUtilTest
    org.eclipse.mylyn.commons.tests.net.SslProtocolSocketFactoryTest
```

Fig. 18. Test Suite Modularization View showing which plug-ins contribute tests

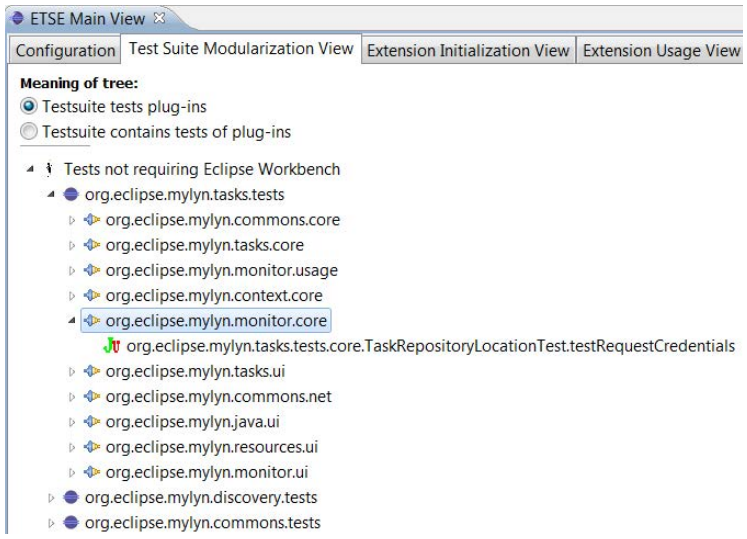
Test Suite Modularization View The *Test Suite Modularization View* helps to understand what is tested by a certain test suite (N_3), and which plug-ins are contributing tests (N_7). From Figure 18, the developer sees that test plug-in "mylyn.discovery.tests" contributes five test cases, and that two other plug-ins namely "mylyn.tasks.tests" and "mylyn.common.tests" contribute test cases to this test suite. On a more detailed level, the view shows which plug-ins are tested by those test plug-ins, and also which test cases are using those plug-ins (N_8).

In Figure 19, this view shows the three test plug-ins, and reveals (when opening those plug-ins) which plug-ins under test have been addressed by test cases from this test plug-in. For example, test plug-in "mylyn.tasks.tests" addresses ten plug-ins under test. When expanding one of those plug-ins, e.g., "mylyn.monitor.core" we see which test cases, in this case only one called "testRequestCredentials", addresses this plug-in.

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

The test suite modularization view, on the other hand, addresses the correlation between test suite, plug-ins contributing test cases, and the plug-ins that have been tested in this test run. The developer can tell, e.g., from Figure 19, that in this test suite the plug-in “mylyn.tasks.tests” contributed test cases “testRequestCredential”, and that during the execution of this test case, plug-in “mylyn.monitor.core” has been tested.

This view helps to cover information needs related to the inspection of coverage of *plug-ins and extensions* (N_3, N_4) and to *test organization* (N_7, N_8), as identified in Section 3.3. For example, the test suite modularization view allows to inspect the test run in terms of entities (i.e. plug-ins, extensions or services) covered within each test suite, and also helps to understand test organization as it reveals which (test) plug-ins contribute tests.



ETSE Main View

Configuration | Test Suite Modularization View | Extension Initialization View | Extension Usage View

Meaning of tree:

[✓ Testsuite tests plug-ins]
Testsuite contains tests of plug-ins

```
Tests not requiring Eclipse Workbench
  org.eclipse.mylyn.tasks.tests
    org.eclipse.mylyn.commons.core
    org.eclipse.mylyn.tasks.core
    org.eclipse.mylyn.monitor.usage
    org.eclipse.mylyn.context.core
    org.eclipse.mylyn.monitor.core
      org.eclipse.mylyn.tasks.tests.core.TaskRepositoryLocationTest.testRequestCredentials
    org.eclipse.mylyn.tasks.ui
    org.eclipse.mylyn.commons.net
    org.eclipse.mylyn.java.ui
    org.eclipse.mylyn.resources.ui
    org.eclipse.mylyn.monitor.ui
  org.eclipse.mylyn.discovery.tests
  org.eclipse.mylyn.commons.tests
```

Fig. 19. Test Suite Modularization View showing which plug-ins are tested

6.3 RQ2: Scalability

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

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

Understandability Our views are based on visualizations of graphs, trees, and tabular data. We manage the inherent limitations of these representations by allowing users to filter data (restricting the view to, e.g., particular extensions) and by offering views at different levels of abstraction (e.g., system, plug-in, method scope).

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

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

The *Extension Initialization View* for Mylyn displays 15 plug-ins related to extension initialization, a number which still yields comprehensible graphs. Likewise, eGit provides 53 extensions, of which 16 are initialized during the test run. If the view is applied to all extension-points of the full Eclipse workbench, the graph may comprise hundreds of nodes, which is less likely to be understandable. Such a use of this view, however, would not be the typical usage scenario: the views are intended to shed light on the extension relations of particular clusters of plug-ins, such as those of Mylyn. Note that the understandability of the view at plug-in scope depends on the number of extension-points defined per plug-in, and not on the overall size of the system under test. This means that for a small system like eGit, the view can be as helpful as for a large-scale system. In both subject systems, the views have an average of 2 and a maximum of 10 extension-points defined per plug-in (based only on plug-ins providing extension-points).

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

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

6.4 RQ3: Accuracy

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

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

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

The accuracy of the *Extension Usage View* is mainly influenced by the classification of the classes to be either visible or invisible for the extension-point. A classification error might occur, if the extension-point does not provide a base-class in its XML schema file. When in doubt, the algorithm behaves conservatively and classifies all types, that are extended by the class of the extension as related. This means no extension usages are missed, but it leads to a wrong classification in case the extension class does not only extend the type declared by the extension-point, but additional classes (e.g. classes only visible to this plug-in). Then, methods of these additional classes are added to the *extension method* set. This can cause the *Extension Usage View* to show more methods of an extension as being used than those that actually occur, and the viewer has to reduce them manually (i.e. identify the right base class of the extension).

This false classification is reduced, by considering that if the extension-point is not declared in the plug-in that provides the extension, and a type extended by the extension is defined within this plug-in, this type cannot be visible to the extension-point, and can be excluded. Until the extension-point is required to indicate a type, we cannot eliminate potential misclassification. In Mylyn, all extension-points provide an XML schema-file. To get an impression for the likelihood of a misclassification we manually inspected all 29 extension classes declared for an extension-point within Mylyn, i.e. representing the system under test. None of those would have led to a misclassification. In addition, we inspected 9 extension classes declared for extension-points declared outside the subject system (but in the ecosystem) to see their potential classification error. Of these, only one class would have caused a misclassification. This class is part of the extension for the extension-point *org.eclipse.ui.handlers*. Here plug-in *mylyn.tasks.bugs* provides an extension based on the class "NewTaskFromMarkerHandler". This class extends class "MarkerViewHandler"[22] which extends "AbstractHandler"[23]. Our algorithm would identify "MarkerViewHandler" already as a potential extension class, whereas the base type of the extension is defined as "AbstractHandler". This leads to the inclusion of methods defined in "MarkerViewHandler" in the analysis.

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

[22] This class is in the package org.eclipse.ui.views.markers
[23] This class is in the package org.eclipse.core.commands

7 Discussion and Future Work

7.1 User Study

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

First feedback from Eclipsers was obtained via a a presentation of our findings about test suite understanding, as well as the five architectural views, to approximately 70 practitioners during the *Eclipse Testing Day*.[24] The overall responses were positive, and encouraged us to proceed.

We consider a full user study as beyond the scope of the present paper. However, to get some initial insight we took the following steps already. In particular, we asked three developers, who have been participants in the grounded theory study described in Section 3, for their feedback on ETSE. We demonstrated the tool and outlined the meaning of each architectural view. All three participants reacted very positively and expressed that the tool gives them a new perspective on plug-in test suites. The developers found the visualization of the degree of integration testing of system under test, illustrated by the *Extension Initialization View*, very useful.

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

7.2 Limitations

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

7.3 Recommendations

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

[24] `http://wiki.eclipse.org/Eclipse_Testing_Day_2011`

Set-Up and Tear-Down While executing a test suite with the Eclipse plug-in test runner, the framework is only started once. Also plug-ins and extensions are created on demand and are not automatically stopped after a test execution of one method. This means that the execution of a test method can change the state of the system, and therefore possibly change the outcome of following tests. For example, a test method that creates an extension, might also need to activate the plug-in providing this extension. In the case, the extension would be used also by a subsequent test method, this test method would not have to activate the plug-in anymore. We believe that there is not enough awareness for the implications of such dependencies between tests. The test runner should allow users to configure the set-up and tear-down of the execution environment, in this case Eclipse. We anticipate that such information would be useful to integrate with ETSE views as well.

7.4 Threats to validity

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

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

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

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

8 Related Work

A recent survey on the use of dynamic analysis for program understanding purposes is provided by Cornelissen *et al.* [7]. One of the findings of this survey is that very few studies exist addressing dynamically reconfigurable systems – a gap that we try to bridge with our paper.

In the area of test suite analysis and understanding, van Deursen *et al.* [11] proposed a series of JUnit test *smells* (pointing to hard to understand test cases) as well as a number of refactorings to remedy them. Later, this work was substantially elaborated by Meszaros into an extensive book on xUnit patterns [29]. Van Rompaey *et al.* propose a formalization of a series of test smells, as well as metrics to support their detection [33].

They also propose heuristics to connect a test class to its corresponding class-under-test – which we also use in our approach. Gälli *et al.* present a taxonomy of (Smalltalk) unit tests, in which they distinguish tests based on, for example, the number of test methods per method-under test, and whether or not exceptions are taken into account [15].

In order to support the understanding of test suites, Cornelissen *et al.* investigate the automated extraction of sequence diagrams from test executions [6]. Zaidman *et al.* investigate implicit connections between production code and test code, by analyzing their co-evolution in version repositories [39]. While these studies provide important starting points, none of them approaches test suite understanding from an integration or extensibility point of view.

Koochakzadeh et al. present a graph-based test coverage visualization tool [24], whose usefulness is evaluated in [17]. The tool, which is now part of the CodeCover test coverage tool for Eclipse, allows to view the test coverage between two artifacts on different scopes (i.e. test package, class and method in xUnit). The views of this tool differ strongly from our views as their focus is visualization of purely "traditional" coverage information with no connection to the plug-in characteristics of systems under test. Further, our approach also analyzes the static meta data information of the plug-in system to gain information on potential integration possibilities and their coverage within the test execution. Visualization of those connections helps also to facilitate comprehension of the system under test, and its test suite.

A substantial body of research has been conducted in the area of integration testing [2, 23, 30]. Closest to the Eclipse extension mechanism are test strategies addressing polymorphism, such as the *all-receiver classes* adequacy criterion [34].

Most integration testing approaches are model-based, and explain how models, e.g., UML state machines, can be used to derive test cases systematically [21, 31]. In the Eclipse setting, it is not common to have models of plug-ins and their extension-points available a priori. As we saw, however, our views can be reverse engineered from static dependency declarations as well as from run time plug-in interactions. As such, they can help developers compare actual plug-in interactions with declared dependencies.

The Eclipse plug-in architecture and the related Eclipse IDE are well studied subject systems in the research community. For example, Wermelinger et al. analyze the evolution of Eclipse plug-ins [38], and Mens et al. investigate whether software evolution metrics are supported within the Eclipse context [28]. Both studies analyze the evolution of Eclipse, whereas our study performs a static and dynamic analysis to study extensibility relations between plug-ins.

The grounded theory methodology originates from the social sciences, and has nowadays gained popularity in the software engineering research field [8, 22, 32]. Also for our study, grounded theory was beneficial as it is suitable, in particular, for explorative, human-centered research areas.

9 Concluding Remarks

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

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

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

References

1. Adolph, S., Hall, W., Kruchten, P.: Using grounded theory to study the experience of software development. Empirical Software Engineering pp. 1–27 (2011)
2. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional (1999)
3. Bryant, A., Charmaz, K. (eds.): The SAGE Handbook of Grounded Theory. SAGE (2007)
4. Chatley, R., Eisenbach, S., Kramer, J., Magee, J., Uchitel, S.: Predictable dynamic plugin systems. In: 7th International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 129–143. Springer-Verlag (2004)
5. Corbin, J.M., Strauss, A.: Grounded theory research: Procedures, canons, and evaluative criteria. Qualitative Sociology **13**, 3–21 (1990)
6. Cornelissen, B., van Deursen, A., Moonen, L., Zaidman, A.: Visualizing testsuites to aid in software understanding. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR’07), pp. 213–222. IEEE Computer Society (2007)
7. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R.: A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering **35**(5), 684–702 (2009)
8. Dagenais, B., Robillard, M.P.: Creating and evolving developer documentation: understanding the decisions of open source contributors. In: Proceedings Foundations of Sofatware Engineering (FSE), pp. 127–136. ACM SIGSOFT (2010)
9. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-oriented reengineering patterns. Morgan Kaufmann (2003)
10. van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C.: Symphony: View-driven software architecture reconstruction. In: Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA’04), pp. 122–134. IEEE Computer Society Press (2004)
11. van Deursen, A., Moonen, L., van Den Bergh, A., Kok, G.: Refactoring test code. In: G. Succi, M. Marchesi, D. Wells, L. Williams (eds.) Extreme Programming Perspectives, pp. 141–152. Addison Wesley (2002)

12. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software, 1. a. edn. Addison-Wesley Professional (2003)
13. Feathers, M.: Working Effectively with Legacy Code. Prentice Hall (2004)
14. Freeman, S., Pryce, N.: Growing Object-Oriented Software, Guided by Tests. Addison-Wesley (2010)
15. Gaelli, M., Lanza, M., Nierstrasz, O.: Towards a taxonomy of SUnit tests. In: 13th International European Smalltalk Conference (ESUG 2005), pp. 1–22 (2005)
16. Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plugins. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2003)
17. Garousi, V., Koochakzadeh, N.: An empirical evaluation to study benefits of visual versus textual test coverage information. In: Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques, TAIC PART'10, pp. 189–193. Springer-Verlag, Berlin, Heidelberg (2010)
18. Glaser, B., Strauss, A.: The discovery of Grounded Theory: Strategies for Qualitative Research. Aldine Transaction (1967)
19. Greiler, M., van Deursen, A., Storey, M.A.: Test confessions: a study of testing practices for plug-in systems. In: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, pp. 244–254. IEEE Press, Piscataway, NJ, USA (2012)
20. Greiler, M., Groß, H.G., van Deursen, A.: Understanding plug-in test suites from an extensibility perspective. In: Proceedings 17th Working Conference on Reverse Engineering (WCRE), pp. 67–76. IEEE Computer Society (2010)
21. Hartmann, J., Imoberdorf, C., Meisinger, M.: UML-Based integration testing. In: International Symposium on Software Testing and Analysis, pp. 60–70. ACM (2000)
22. Hermans, F., Pinzger, M., van Deursen, A.: Supporting professional spreadsheet users by generating leveled dataflow diagrams. In: H. Gall, N. Medvidovic (eds.) Proceedings 33rd International Conference on Software Engineering (ICSE 2011). ACM (2011)
23. Jorgensen, P.C., Erickson, C.: Object-oriented integration testing. Communications of the ACM **37**(9), 30 (1994)
24. Koochakzadeh, N., Garousi, V.: Tecrevis: a tool for test coverage and test redundancy visualization. In: Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques, TAIC PART'10, pp. 129–136. Springer-Verlag, Berlin, Heidelberg (2010)
25. Marquardt, K.: Patterns for plug-ins. In: Proceedings 4th European Conference on Pattern Languages of Programs (EuroPLoP), p. 37pp. Bad Irsee, Germany (1999)
26. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship, 1 edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2008)
27. Mayer, J., Melzer, I., Schweiggert, F.: Lightweight plug-in-based application development. In: International Conference NetObjectDays, NODe 2002, pp. 87–102. Springer-Verlag (2003)
28. Mens, T., Fernández-Ramil, J., Degrandsart, S.: The evolution of eclipse. In: Proceedings 24th IEEE International Conference on Software Maintenance (ICSM), pp. 386–395. IEEE (2008)
29. Meszaros, G.: xUnit Test Patterns: Refactoring Test Code. Addison-Wesley (2007)
30. Pezzè, M., Young, M.: Software Testing and Analysis. Wiley (2008)
31. Reis, S., Metzger, A., Pohl, K.: Integration testing in software product line engineering:a model-based technique. Lecture Notes In Computer Science pp. 321–335 (2007)
32. Rigby, P.C., Storey, M.A.: Understanding Broadcast Based Peer Review on Open Source Software Projects. In: ICSE '11: Proceedings of the 33rd International Conference on Software Engineering. ACM (2011)

33. van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M.: On the detection of test smells: A metrics-based approach for general fixture and eager test. IEEE Transactions on Software Engineering **33**(12), 800–817 (2007)
34. Rountev, A., Milanova, A., Ryder, B.: Fragment class analysis for testing of polymorphism in Java software. IEEE Transactions on Software Engineering **30**(6), 372–387 (2004)
35. Rubio, D.: Testing with Spring and OSGi, chap. 9, pp. 331–359. Apress, Berkeley, CA (2009)
36. Shavor, S., D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P.: The Java Developer's Guide to Eclipse. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2005)
37. The OSGi Alliance: OSGi Service Platform Core Specification; Release 4, Version 4.3. `http://www.osgi.org` (2011)
38. Wermelinger, M., Yu, Y.: Analyzing the evolution of eclipse plugins. In: Proceedings of the 2008 international working conference on Mining software repositories, MSR '08, pp. 133–136. ACM, New York, NY, USA (2008)
39. Zaidman, A., van Rompaey, B., Demeyer, S., van Deursen, A.: Mining software repositories to study co-evolution of production & test code. In: Proceedings 1st International Conference on Software Testing Verification and Validation (ICST), pp. 220–229. IEEE Computer Society (2008)