Delft University of Technology
Software Engineering Research Group
Technical Report Series

Understanding Plug-in Test Suites from an Extensibility Perspective

Michaela Greiler, Hans-Gerhard Gross and Arie van Deursen

Report TUD-SERG-2010-028

TUD-SERG-2010-028

Published, produced and distributed by:

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

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the Working Conference on Reverse Engineering (WCRE), 2010, IEEE Computer Society.

Understanding Plug-in Test Suites from an Extensibility Perspective

Michaela Greiler
Delft University of Technology
The Netherlands
m.s.greiler@tudelft.nl

Hans-Gerhard Gross
Delft University of Technology
The Netherlands
h.g.gross@tudelft.nl

Arie van Deursen*
Delft University of Technology
The Netherlands
arie.vandeursen@tudelft.nl

Abstract

Plug-in architectures enable developers to build extensible software products. Such products are assembled from plug-ins, and their functionality can be enriched by adding or configuring plug-ins. The plug-ins themselves consist also of multiple plug-ins, and offer dedicated points through which their functionality can be influenced. A well-known example of such an architecture is Eclipse, best known for its use to create a series of extensible IDEs.

In order to test systems built from plug-ins developers use extensive automated test suites. Unfortunately, current testing tools offer little insight in which of the many possible combinations of plug-ins and plug-in configurations are actually tested.

To remedy this problem, we propose three 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, and extension usage. The views are implemented in ETSE, the Eclipse Plug-in Test Suite Exploration tool. We evaluate the proposed views by analyzing eGit and Mylyn, two open source Eclipse plug-ins.

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 [2, 10, 11]. 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 [17].

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 [7]. 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 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 [3, 12, 19, 20]. Most of the existing work, however, focuses on the *unit* level. While this is an essential first step, for plug-in-based architectures it is insufficient, since 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.

Based on this, we propose to look at test suites from an *extensibility* perspective, focusing on the way in which plug-ins are used dynamically to extend system functionality. Thus, the central research question of this paper is: *How can we support developers in understanding complex test suites for plug-in-based architectures from an extensibility perspective?*

To address this question, we propose three *architectural views* [18] that can help engineers understand plug-in in-

*Work done while at the *Computer Human Interaction and Software Engineering Lab (CHISEL)*, Department of Computer Science, University of Victoria, Canada.

[1] `http://www.eclipse.org`

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

[3] `http://www.eclipse.org/mylyn`

teractions. The views we propose are tailored towards the plug-in architecture of the Eclipse ecosystem. Eclipse is of particular interest, since it not only offers 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.

To offer insight in these extension mechanisms, we propose three views: the Plug-in *Modularization* View, the Extension *Initialization* View, and the Extension *Usage* View, which will be discussed in Section 3. To construct these views, we follow the Symphony software architecture reconstruction approach [18], and deploy a mixture of static and dynamic analysis.

To evaluate the usefulness of these views, we discuss their application to two open source Eclipse plug-ins: the fairly small eGit plug-in permitting the use of the `git` versioning system within Eclipse, and the substantial collection of plug-ins that comprises the Mylyn plug-in for work item management.

The paper is structured as follows. Section 2 provides the necessary background material on plug-in architectures. Section 3 describes our approach, and covers the three architectural views. Section 4 discusses our tool suite for reconstructing these views, after which Section 5 describes how the views helped to understand two case studies. We reflect on the case study findings in Section 6, after which we conclude with a summary of related work, contributions, and areas for future research.

2 Background

2.1 Eclipse Modularization

Plug-in based dynamic modularization systems are widely used to create adaptive and configurable systems [2]. A well known example is OSGi[4], which provides a dynamic modularization platform for Java.

The Eclipse plug-in architecture[5] is based on the Equinox[6] implementation of OSGi. 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 a specific meta data file, the manifest. The manifest describes, among others, the dependencies between plug-ins.

Plug-ins represent the basic extensibility feature of Eclipse, allowing dynamic loading of new functionality.

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

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

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

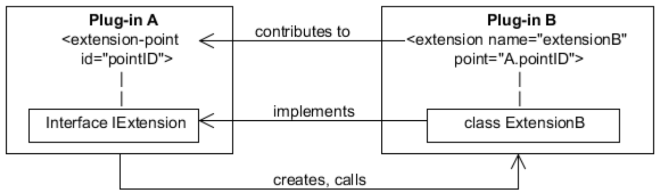


Figure 1. The Eclipse plug-in extension mechanism

Plug-in P can invoke functionality from other plug-ins P_i. At compile time, this requires the availability of the constituent plug-in's Java interfaces, giving rise to *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 [17]. 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 syntactical 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.

2.2 Eclipse Testing Practices

Gamma and Beck [7] 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[7] 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 [16]. We have not been able to find test strategies targeting integration testing of dynamic modularization systems in general, or plug-in systems in particular.

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

A substantial body of research has been conducted in the area of integration testing [1, 9, 13]. Closest to the Eclipse extension mechanism are test strategies addressing polymorphism, such as the *all-receiver classes* adequacy criterion [15].

Most integration testing approaches are model-based, and explain how, e.g., UML state machines can be used to derive test cases systematically [8, 14]. In the Eclipse setting, it is not common to have models of plug-ins and their extension-points available a priori. As we will see, 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.

3 Models for Test Suite Understanding

When facing the problem of understanding a large test suite, a first step for an engineer is to look at the documentation, to gather basic static and dynamic information on e.g. the size in lines of code, and to assess the test coverage and timing through test execution. These activities are similar to what newcomers do with regular code in a “first contact” setting, as described by DeMeyer *et al.* [5].

While this provides an initial sense of the scope and setup of the test suite, it does not yield insight in the internal structure and organization of the test suite. Currently, the only way towards deeper insight is the (manual) inspection of the code. The goal of the first view, the *Plug-in Modularization View*, therefore, 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*.

Finally, the *Extension Usage View* completes the picture by providing the developer with insight in the way the test suite exercises the actual methods involved in the extensions.

In this section we present these views, state their goal, and formulate the information needs they address. To reconstruct the views, we follow the Symphony architecture reconstruction method [18]. Thus, we distinguish *source models* corresponding to the raw data we collect, *target models* 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.

3.1 The Plug-in Modularization View

The Plug-in Modularization View provides 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?”.

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.

Figure 2 illustrates this view. It shows test-component *commons.tests* of Mylyn and its static (on the left) and dynamic code-dependencies (on the right). 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.

3.2 Extension Initialization View

The plug-in modularization view 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 following 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, but we also gain insights in the system-under test and its extensibility relations. The meta model of this view is illustrated in Figure 3.

The view helps answering questions on extensions and the way they are tested at system, plug-in, and test-method level, as discussed below.

System Scope. At system scope, the view gives insights 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 7. 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 label of the link represents the number of statically declared extensions one plug-in provides for the other, and the number of extensions that are actually used during a test run.

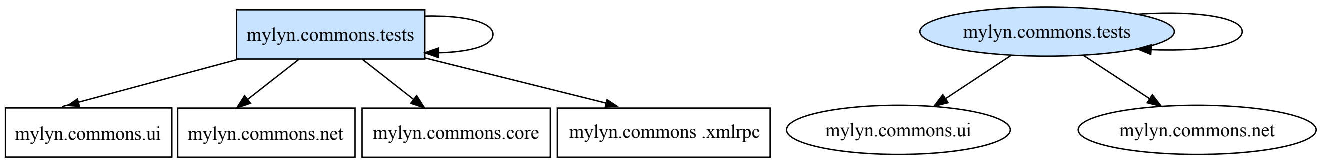


Figure 2. Static and Dynamic Dependencies of Test-Component "mylyn.commons.tests"

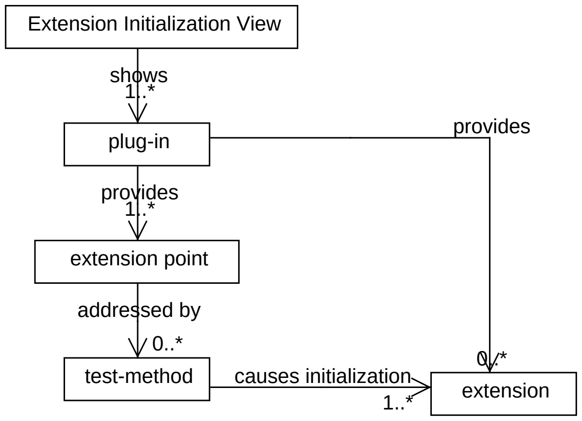


Figure 3. Meta model of the Extension Initialization View

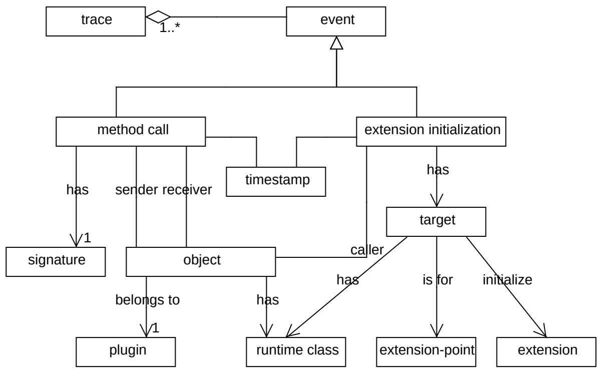


Figure 4. Trace meta model

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

This can be visualized e.g., by means of a graph. An example is given in Figure 6. 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.

The view can also be used to show all extensions declared by the system-under test, and those that have been activated during a test run.

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. In this way, a developer or tester can identify the location of the test-code for a particular extension-point.

3.2.1 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).

Listing 1. Extension Initialization Aspect

```
trace call createExecutableExtension(..) with target(Object o);

before createExecutableExtension(){
  write(o.getExtensionPoint(), o.getContributor, time)
}
```

The trace used for this view comprises "extension initialization events" during the test run, as illustrated by the the trace meta model in Figure 4. 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. The pseudo-code of such an aspect is given in Listing 1.

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.

3.2.2 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 data does expose information about the usage of an extension. To take advantage of that, the Extension Usage View is introduced in the following.

3.3 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. The question it addresses is "which extensions have been actually used during the test run, and when and how have they been used?"

The 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 are target at, 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.

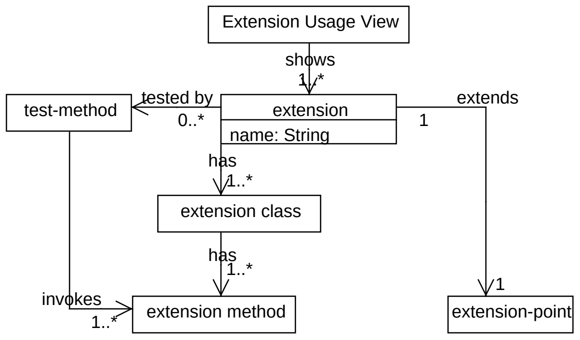


Figure 5. Meta Model: Extension Usage View

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.

Zooming in to the *extension scope*, the developer can see which test-methods have used a given extension. This information is helpful to spot the right piece of code responsible for the extension usage, e.g., to enhance or change it.

A refinement of this view to the *usage scope* shows how the extension has been used during the test run. All methods of an extension that have been called during testing are listed. The view also visualizes which of those methods have been redefined by the particular extension. With this view, the tester gains knowledge about which integrations of extensions have been tested, and can locate test code responsible for the usage of an extension.

3.3.1 Underlying Meta-Models

The execution trace used to construct the Extension Usage View comprises detailed method calls of a test run, as illustrated by the meta model 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.

In order to see actual usage of extensions, dynamic information is required. As an example, consider Listing 2 illustrating an simplified example of an invocation of an extension. Class Extension defines the base class of the extension. Class B and C are extensions, extending the base class. The ExtensionUsage represents the code in the extension-point using an extension. In the trace an invocation of an

Listing 2. Extension Usage Example

```
class Extension{
  void me1() {}
}

class B extends Extension {
  void mb1() {}
  void me1() {}
}

class C extends Extension {
  void mc1() {}
}

class ExtensionUsage{
  void invokeExtension(Extension [] extensions){
    for(Extension e : extensions)
      e.me1();
  }}
```

extension is visible as a call to "Extension.me1()". This implies, that the runtime-class for every extension invocation has to be known, in order to distinguish the extensions from each other, e.g., B from C in the example.

3.3.2 Reconstructing the View

To construct this view, we need in addition to the dynamic data discussed before, the method set of an extension that can be used by an extension-point to invoke it. We will refer to this set as to the extension method set. As Eclipse does not force an extension-point to declare formally the type an extension has to extend, we might have to derive our extension method set based on a heuristic.

First, in case the extension-point declares a base class 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.

In the case, no base class is provided, the algorithm collects all the classes an extension declares in its meta data file. Starting from these classes, 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*. 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 optimistic 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.

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 Implementation

We implemented the reconstruction and presentation of our views in ETSE[8], our "Eclipse Test Suite Exploration Tool". It is implemented in Java, and offers an API to construct the views in question.

To analyze the static Java code we use the Byte Code Engineering Library[9], which inspects and manipulates the binary Java class files. Meta data, including the OSGi manifest and the plugin.xml files, is collected and analyzed. To trace the execution of the test run, we use aspect-oriented programming, in particular the AspectJ[10] framework. We defined several aspects, addressing different join points to weave in our tracing advices. There are three main classes of aspects that can be differentiated: the aspect used for weaving into the initialization of the extensions, the aspect used to trace method calls, and the aspect used to trace plug-in starts and stops.

5 Evaluation of the Views

The evaluation is based on a case study including two subject systems whose test suites have been investigated by means of the proposed views. We defined the following research questions to evaluate whether the views meet the information needs of a developer (RQ1&2), and to estimate how scalable (RQ3) and accurate (RQ4) the views are:
RQ1: To which extent does the Extension Initialization View help to understand the influencing relations between the plug-ins under test, and how much does it help to understand which of those relations have been addressed in the test suite?
RQ2: To which extent does the Extension Usage View help to understand how the integration of extensions has been addressed by the test suite?
RQ3: How understandable and manageable are the views

[8]We are in the process of creating an ETSE distribution at `http://swerl.tudelft.nl/bin/view/Main/ETSE`
[9]`http://jakarta.apache.org/bcel`
[10]`http://www.eclipse.org/aspectj`

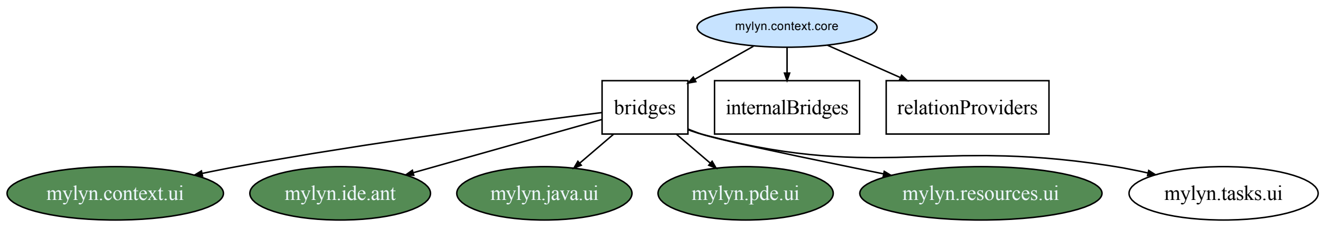


Figure 6. Static and Dynamic Dependencies based on Extension-Points: Plug-In Scope

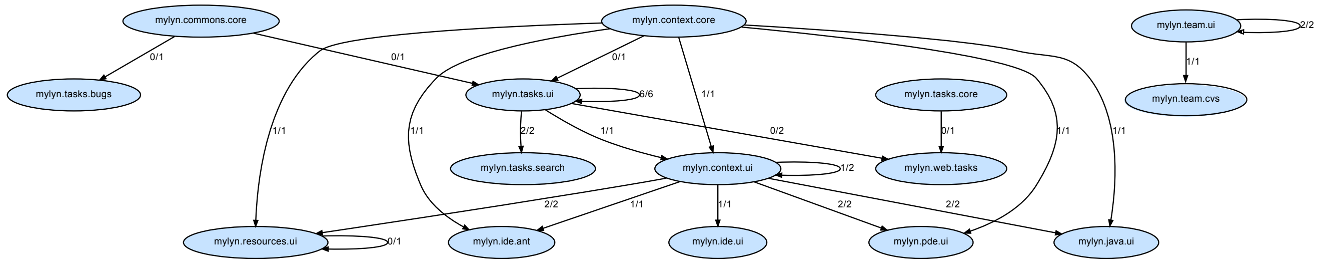


Figure 7. Static and Dynamic Dependencies based on Extension-Points: System Scope

for large-scale systems?
RQ4: How accurate are the views presenting the system-under test?

All research questions are addressed with respect to the different abstraction levels provided by the proposed views.

5.1 The Subject Systems

One experimental subject is eGit[11], a smaller plug-in system designed to integrate the source control management system Git into Eclipse. eGit is a good fit for our evaluation, mainly of its small size, that permits, in addition to the investigation by means of the views, to manually inspect the complete system. eGit consists of three main plug-ins, and two test suites. One test suite comprises the core tests, and the other the user-interface tests based on the *SWTBot* framework. The underlying source code has 28,300 lines of code, and the test suites comprise 1,700 lines of code.

The other study subject is Mylyn, a task management system for Eclipse. Mylyn has been chosen because it represents a large-scale plug-in system, and gives valuable insights to 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. It includes the selection of 27 plug-ins that make up the core contribution. Those 27 plug-ins come with 11 test-components. Additional contributions, like connectors, are excluded from this study. The source code 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.

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

5.2 Information Needs

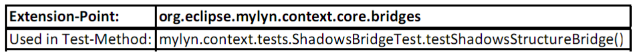
This section presents the evaluation results for investigating research questions one and two. We do so by going through the use of the views for Mylyn, followed by a reflection on the strengths and weaknesses of the views.

The Views in Practice The 27 plug-ins in Mylyn offer 25 extension-points to contribute functionality, and also declare 148 extensions to enhance its functionality and that of Eclipse.

The first question during this evaluation is whether the Extension Initialization View helps to understand how the 148 extensions are related to the 25 extension-points within the system-under test, and also which of those relations have been covered by the test suite.

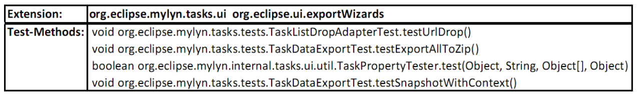
This view at system scope for Mylyn is illustrated in Figure 7. An edge between two plug-ins means that one plug-in declares an extension-point for which the other plug-in provides an extension. The view abstracts from the specific extension-points declared. However, the fraction on the edge states how many of the static declarations (bottom of fraction) are activated during a test run (top).

At plug-in scope, this view is illustrated by Figure 6 for plug-in *mylyn.context.core*. The plug-in provides three extension-points, namely *bridges*, *internalBridges* and *relationProviders*. The view shows that within Mylyn six plug-ins exist that use extension-point *bridges* to influence the plug-in, represented by the six nodes connected to this extension-point. The coloring of five nodes indicates that only five of the relations are activated during the test run. The view does not give explanations, but points to one plug-



Extension-Point:	org.eclipse.mylyn.context.core.bridges
Used in Test-Method:	mylyn.context.tests.ShadowsBridgeTest.testShadowsStructureBridge()

Figure 8. Extension-Initialization View



Extension:	org.eclipse.mylyn.tasks.ui org.eclipse.ui.exportWizards
Test-Methods:	void org.eclipse.mylyn.tasks.tests.TaskListDropAdapterTest.testUrlDrop()
	void org.eclipse.mylyn.tasks.tests.TaskDataExportTest.testExportAllToZip()
	boolean org.eclipse.mylyn.internal.tasks.ui.util.TaskPropertyTester.test(Object, String, Object[], Object)
	void org.eclipse.mylyn.tasks.tests.TaskDataExportTest.testSnapshotWithContext()

Figure 9. Extension Usage Profile

in the developer might manually inspect and find an empty XML declaration for this extension.

To understand which test-method causes this extension-point to load its extensions, the developer zooms at method scope, as illustrated by Figure 8. Subsequently, the Extension Usage View provides deeper insight in the actual usage of those extensions and reveals that none loaded by *bridges* is actually used during a test run.

Findings. The Extension Initialization View allows to understand the relations of the plug-ins under test, based on their contributions to each other. Especially for a large-scale system like Mylyn, the system scope view has proven useful to visualize and represent how plug-ins influence the behavior of each other, and to indicate which of those extension relations have been addressed by the test suite. On the other hand, the view does not show how the system-under test influences or is influenced by its ecosystem, i.e., Eclipse. Nevertheless, the borders defining the system-under test can be chosen by the viewer. On plug-in level, the view helps to understand which extension-points load extensions. On the other hand, the view does not indicate reasons why some relations are activated and others are not, as in Figure 6.

The Extension Usage View helps to understand how the test suite addresses integration with respect to extensions. On a detailed level, this view allows to locate the test code related to an extension, as Figure 9 shows for an extension of *exportWizards*. 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.

However, both of the views do not evaluate the test suite against coverage and test adequacy criteria. Despite that, they give the developer a valuable perception to judge the quality of the test suite. With respect to Mylyn and eGit, the views revealed that not all of the extension-points and extensions are tested. By means of the detailed views, we were able to locate source code and get insights in the testing approaches for extension and extension-points. For example the *bridges* extension-point, addressed in the examples before, is tested through explicit adding of an extension-object by the test-method.

5.3 Scalability

This section presents the scalability of the views first, with respect to their understandability by human viewers and then, in terms of the manageability of disk space required for the trace files.

In general, the views provide several abstraction levels (e.g, system, plug-in and method scope) to better cope with scalability issues. We discuss scalability for both views at several of these abstraction levels.

Extension Initialization View. At system scope, the number of entities displayed turns the balance. The viewer has to be able to comprehend the relations the entities have with each other to understand the overall system. Within Mylyn, the system view is, with 15 related plug-ins, still understandable. On the other hand, the system scope view is not scalable enough to represent a plug-in system as complex as the Eclipse IDE in a usable way.

The evaluation for eGit showed eGit is too simple for the initialization view at system scope. eGit only has a few plug-ins with a few extension relations: the view is more helpful for more complex systems.

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

Extension Usage View. The extension usage view presents at all abstraction levels information that can be consumed per item. This means the entities do not have to be put in relation with 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 extension have been used, or the developer is concerned with a particular extension, or method.

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 reasonable, e.g., 32Mb for Mylyn and 52Kb for eGit. On the other hand, the trace file required for the identification of the Extension Usage includes trace data from several packages outside of the system-under test and can become large. The trace of Mylyn, for all of the 148 extensions has 6Gb. However, the number of packages included for tracing are affected by the number of extensions analyzed. The size of the file depends on this variable. We argue that an usual usage scenario for this view involves the inspection of a small number of extensions, e.g., 1-5. Then the according trace would be much smaller. Once this trace is analyzed the remaining information can be stored within the megabyte range, (e.g., 6Mb for Mylyn).

5.4 Accuracy and Correctness

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.

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 optimistic 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 if the extension class does not only extend the Type declared by the extension-point. Then, the view indicates more extension usages than happen, and the viewer has to reduce them manually.

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 elude 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.

6 Discussion

Limitations. 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. But, the Extension Usage View does not yet address directly extension-points defined by the system-under test and their extensions outside of the system-under test. That would be an extension e.g., defined by Eclipse for an extension-point inside the system-under test. Even though, the tester would have to think about if this part of Eclipse should not be included in the system-under test.

Recommendations. *Standardization.* As discussed, extensions can be of two types, data or executable extensions. In Eclipse there is no formal way to distinguish them. Further, an extension-point is not forced to provide an XML schema-file describing the syntactical contract between creator and contributor. We would recommend stricter declarations for extension-points. Also a standardization for core elements required within the meta data XML file would facilitate the comprehensibility of plug-in systems.

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 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 this circumstance. The test runner should allow to configure the set-up and tear-down behavior for the execution environment, in this case Eclipse.

Threats to validity With respect to *external* validity, the case studies chosen, Mylyn and eGit, can be considered representative for Eclipse plug-ins. In particular Mylyn is a complex plug-in, and hence we expect the views to be useful 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 useful in such a callback setting as well, in particular if they are, like Eclipse, based on OSGi.

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

7 Related Work

A recent survey on the use of dynamic analysis for program understanding purposes is provided by Cornelissen *et al.* [4]. 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.* [19] 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 [12]. Van Rompaey *et al.* propose a formalization of a series of test smells, as well as metrics to support their detection [20]. 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 [6].

In order to support the understanding of test suites, Cornelissen *et al.* investigate the automated extraction of sequence diagrams from test executions [3]. Zaidman *et al.* investigate implicit connections between production code and test code, by analyzing their co-evolution in version repositories [21]. While these studies provide important starting points, none of them approaches test suite understanding from an integration or extensibility point of view.

8 Concluding Remarks

In this paper, we have addressed the problem of understanding test suites for plug-in-based architectures. In particular, the following are our key contributions: (1) Two architectural views that can be used to understand test suites for plug-in-based systems from an extensibility perspective; (2) the Eclipse Plug-in Test Suite Exploration (ETSE) tool, that can be used to recover the proposed views from existing systems by means of static and dynamic analysis; and (3) an empirical study of the use of these views in Mylyn and eGit. In our future work, we will first of all apply the proposed approach to further plug-in-based architectures. 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] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Professional, Oct. 1999.

[2] R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel. Predictable dynamic plugin systems. In *7th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 129–143. Springer-Verlag, 2004.

[3] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 213–222. IEEE Computer Society, 2007.

[4] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented reengineering patterns*. Morgan Kaufmann, 2003.

[6] M. Gaelli, M. Lanza, and O. Nierstrasz. Towards a taxonomy of SUnit tests. In *13th International European Smalltalk Conference (ESUG 2005)*, pages 1–22, 2005.

[7] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[8] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-Based integration testing. In *International Symposium on Software Testing and Analysis*, pages 60–70. ACM, 2000.

[9] P. C. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30, 1994.

[10] K. Marquardt. Patterns for plug-ins. In *Proceedings 4th European Conference on Pattern Languages of Programs (EuroPLoP)*, page 37pp, Bad Irsee, Germany, 1999.

[11] J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development. In *International Conference NetObjectDays, NODe 2002*, pages 87–102. Springer-Verlag, 2003.

[12] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[13] M. Pezzè and M. Young. *Software Testing and Analysis*. Wiley, 2008.

[14] S. Reis, A. Metzger, and K. Pohl. Integration testing in software product line engineering:a model-based technique. *Lecture Notes In Computer Science*, pages 321–335, 2007.

[15] A. Rountev, A. Milanova, and B. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, June 2004.

[16] D. Rubio. *Testing with Spring and OSGi*, chapter 9, pages 331–359. Apress, Berkeley, CA, 2009.

[17] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[18] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 122–134. IEEE Computer Society Press, 2004.

[19] A. van Deursen, L. Moonen, A. van Den Bergh, and G. Kok. Refactoring test code. In G. Succi, M. Marchesi, D. Wells, and L. Williams, editors, *Extreme Programming Perspectives*, pages 141–152. Addison Wesley, 2002.

[20] B. van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. 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.

[21] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proceedings 1st International Conference on Software Testing Verification and Validation (ICST)*, pages 220–229. IEEE Computer Society, 2008.

TUD-SERG-2010-028
ISSN 1872-5392