Using Dynamic Analysis for API Migration*

Lea Haensenberger and Adrian Kuhn and Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
lhaensenberger@students.unibe.ch, {akuhn,oscar}@iam.unibe.ch

Abstract

When changing the API of a framework, we need to migrate its clients. This is best done automatically. In this paper, we focus on API migration where the mechanism for inversion of control changes. We propose to use dynamic analysis for such API migration since structural refactorings alone are often not sufficient. We consider JExample *as a case-study.* JExample *extends* JUnit *with first-class dependencies and fixture injection. We investigate how dynamically collected information about test coverage and about instances under test can be used to detect dependency injection candidates.*

Keywords: API migration, automatic software engineering, dynamic analysis, inversion of control.

1. Introduction

Large software systems are not written from scratch, but rather reuse functionality offered by third-party frameworks. Frameworks provide their functionality through an application programming interface (API) that typically inverts the control between client and framework.

"The framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application." – Ralph Johnson *et al* [6]

An API is a contract between framework and client that guarantees stability. No changes to the client are required when updating the framework, or even, when moving to a framework implementation of another vendor. However, sometimes comes the moment when we must migrate the client to a different API.

Migrating client code from one API to another is tedious work and thus best done automatically. Sometimes this can be done using a series of refactorings that map the structure of one API to the other. However, when the mechanism for inversion of control differs, a mere structural mapping is often not sufficient. Therefore, we propose to use dynamically collected information for automatic migration of APIs with different mechanism for inversion of control.

In this paper, we consider JExample as a case-study [7, 4]. JExample extends JUnit with first-class dependencies and fixture injection. We identified the following migration steps that require information obtained from dynamic analysis in order to be done.

- For detection of dependencies we propose to record the coverage set of each test, such that the partial order, *i.e.,* subset relationship, of coverage sets can be used to introduce dependencies.
- For detection of injection candidates we propose to use record the state of the instances under test, such that redundant setup code can instead be replaced with fixture injection.

The remainder of this paper is structured as follows. Section 2 introduces JExample. We propose why and how to use dynamic analysis to detect dependencies (Section 3) and candidate fixtures (Section 4). We discuss other that might require dynamic migration in Section 5, and Section 6 concludes.

2. JExample in a Nutshell

JExample introduces producer-consumer relationships to JUnit unit testing.

- A producer is a test method that yields an instance of its unit under test as return value.

* In *Proceedings IEEE Workshop on Program Comprehension through Dynamic Analysis* (PCODA 2008), October 2008, pp. 32–36.

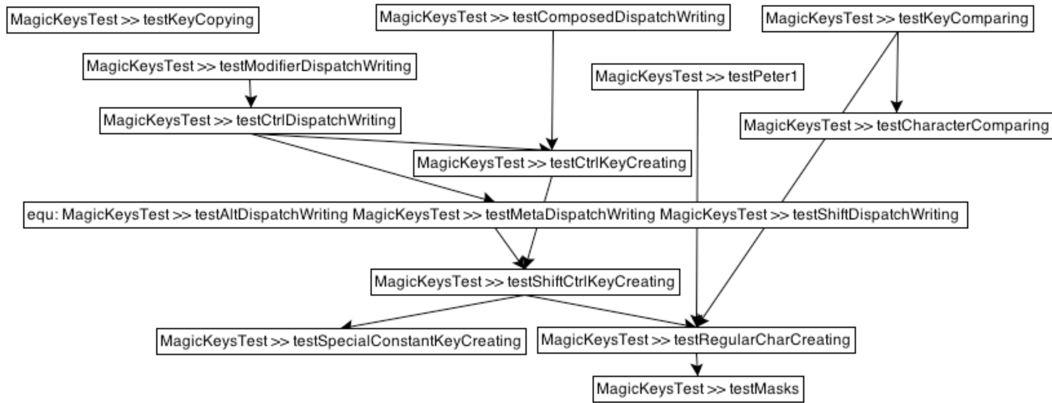

Figure 1. Partial order of the coverage sets of the test suite of `MagicKeys`. Figure courtesy of Gaelli et al [3].

- A consumer is a test method that depends on one or more producers.

JExample caches the return values of producer methods and injects them when a consumer is about to be executed. Producer-consumer relationships are first-class dependencies: when running a test suite, JExample will skips any test method whose producers have previously failed or been skipped.

For example, when testing a stack class, you may declare that `testPop` depends on the unit under test of `testPush` as follows.

```
@Test
public Stack testPush() {
    Stack $ = new Stack();
    $.push("foo");
    assertEquals("foo", $.top());
    assertEquals(1, $.size());
    return $;
}

@Test
@Depends("#testPush")
public void testPop(Stack $) {
    Object top = $.pop();
    assertEquals("foo", top);
    assertEquals(0, $.size());
}
```

We refer to producers and consumer methods as *example methods*. They do more than just test the unit under test. Producer methods consist of source code that illustrates the usage of the unit under test, and that may return a characteristic instance of their unit under test. Thus, producer/consumer methods are in fact examples of the unit under test.

As such, example methods tackle the same problem as mock objects, *i.e.,* "How to test a unit that depends on other units?" When working with mock objects, you solve this problem by creating a mock for each dependency. When working with examples, you solve this problem by declaring producer-consumer dependencies. Thus, instead of testing against mocks, you test against the previously created return values of other tests. Since example methods are both producers and testers of their returned value, all return values are guaranteed to be valid and fully functional instances of the corresponding unit. In addition, JExample will clone example objects to take care that no side-effects are introduced when two or more consumers use the same return value.

An example method may depend on both successful execution and return values of other examples. If it does, it must declare the dependencies using an `@Depends` annotation. An example method with dependencies may have method parameters. The number of parameters must be less than or equal to the number of dependencies. The type of the n-th parameter must match the return type of the n-th dependency.

Dependency declarations uses the same syntax as the `@link` tag of the Java documentation tool. References are either fully qualified or not. If less than fully qualified, JExample searches first in the declaring class and then in the enclosing package. The following table shows the different forms of references.

```
#method
#method(Type, Type, ...)
class#method
class#method(Type, Type, ...)
package.class#method
package.class#method(Type, Type, ...)
```

Multiple references are separated by either a comma (,) or a semicolon (;). As listed above, the hash character (#), rather than a dot (.) separates a member from its class. However, JExample is generally lenient and

will properly parse a dot if there is no ambiguity. This is the same as the Java documentation tool does.

3. Detecting Dependencies

Test methods in JExample can have explicit dependencies on other test methods. If dependencies are properly declared, a failing method directly points to the defect location (since all dependents that would otherwise fail as well are skipped) whereas in JUnit many dozens of test methods covering the same defect location fail and it is often not obvious where to start fixing the bug. A test method m_a should depend on a test method m_b, if m_a covers at least the same code as m_b. The execution of method m_a can be skipped, if the framework already knows that m_b fails.

Gaelli *et al* have shown that a partial order of test methods by means of *coverage sets* helps developers to locate a defect by pointing out the test method with the most specific debugging context [3]. Figure 1 illustrates the partial order of one test suite.

Thus, we propose to migrate JUnit tests to JExample by running the tests and recording the coverage set of each test. The coverage set of a test contains all methods that get invoked when running the test. In the main step we make a partial order of coverage sets, in order to detect coverage dependencies between test methods. If a JUnit method m_a is found to cover a superset of a JUnit method m_b, then m_b is migrated to a JExample method with a `@Depends` annotation to m_a.

Consider the following two test methods of a JUnit test case testing Java's Stack.

```
@Test
public void testPush() {
    Stack stack = new Stack();
    stack.push("foo");
    assertEquals("foo", stack.top());
    assertEquals(1, stack.size());
}

@Test
public void testPop() {
    Stack stack = new Stack();
    stack.push("foo");
    Object top = stack.pop();
    assertEquals("foo", top);
    assertEquals(0, stack.size());
}
```

In the example above `testPop` covers `testPush`, since the set of methods invoked by `testPop` is a superset of the methods invoked by `testPush`. Thus, in the JExample implementation of the Stack test, as given previously in Section 2, `testPop` declares itself to explicitly depend on `testPush` as follows:

```
@Test
@Depends("#testPush")
public void testPop() { ...
```

Please note, as we add a `depends` annotation but no methods parameters, a dependency without fixture injection is added. Detection of fixture injection candidates is covered in the next section.

4. Detecting Candidates for Fixture Injection

Test methods in JExample can pass an instance of the unit under test (instance under test) from one test method to another. If a test method m_a returns a value, the JExample framework caches this return value. If later the framework is about to execute a test method m_b that depends on m_a, the cached return value of m_a is cloned and injected as a parameter to the method invokation of m_b. As such, in JExample a test method may provide the fixture for its dependent methods. Thus, we refer to the former as the *producer* and to the latter as its *consumers*.

Again we consider the Stack example as given in Section 3. In JUnit both `testPush` and `testPop` create a new instance of Stack, the unit under test. Both methods push the same String onto their Stack instance, thus they share the same setup of the instance under test. The method `testPush` ends at this point, whereas `testPop` continues with further operations on its instance. In JExample we can rewrite this so that `testPush` returns its instance under test as return value and `testPop` expects this return value to be injected as a method parameter.

We propose to migrate JUnit tests to JExample by running the tests, but this time recording the created instances under test. If at any moment during the execution of test method m_a the instance under test has the same state as method m_b's instance under test at the end of method m_b, then we have found a candidate for fixture injection.

There are two possible techniques to check if the instances under test are the same:

- All fields of both instances are equivalent.
- The path that produced the instance is the same.

For example, even if the String pushed by m_a and m_b is not the same, we might consider it as a fixture injection candidate. This candidate might however be a false positive if m_a or m_b tests a boundary condition that particularly depends on the pushed value.

In the same way, it is possible to create an empty Stack instance by many different paths that might not all be equivalent. For example, a freshly created Stack might have a different modification count than one that has been filled and later emptied again.

We propose to use both techniques, since singeling out false positive candidates is straightforward: if the migrated tests do not run we have obviously introduced an error. Thus, if we migrate the candidates one by one we can make sure no false positives are migrated.

The migration script will collect all candidates, as well as the dependencies detected above in Section 3, in order to migrate JUnit test suites to JExample test suites. If both a dependency from m_a to m_b and a fixture candidate has been detected, the method m_a is rewritten to not only depend on m_b but to also take m_b's return value as a parameter as follows:

```
@Test
public Stack testPush() {
    ...
    return stack;
}

@Test
@Depends("#testPush")
public void testPop(Stack stack) {
    ...
}
```

In addition, if a JUnit test-class has a `@Setup` method, this method will be migrated to a JExample test method that is a producer, and all other methods in the same test-class become its consumers.

5. Current and Emerging API Trends

In this section, we provide current and emergent API trends where the mechanisms for inversion of control is affected, and suggest how dynamically obtained information might be useful in order to migrate these APIs:

- XML frameworks offer a wide range of APIs with different control mechanisms. The main divisions are tree- and streaming-based APIs, with the streaming APIs further subdivided into push- and pull models. For example, the DOM model is tree-based [11], whereas SAX uses a push-model streaming API [10]. In addition, non-imperative APIs are emerging that enrich XML processing with the functional and logical paradigm. For Example, LINQ uses functional queries to map objects to XML or SQL and back [8].
- The latest release of J2EE, Java's enterprise application framework, moves from EJB's heavy-weight applications servers to light-weight technologies such as Hibernate and Spring [5]. Both approaches use inversion of control[1], but employ different mechanisms in order to do so. EJB hardwires application code into the application server framework by passing around explicit references to the container. Spring on the other hand uses *dependency injection* to inject container-provided objects into annotated fields of the application code. At this moment, many J2EE systems are about to be migrated from EJB 2.0 to EJB 3.0, that is from conventional application servers to Spring and Hibernate.
- In the field of unit testing, frameworks with first-class dependencies are emerging. For example, both TestNG and JExample extend conventional unit tests with dependencies between tests [7, 4]. When running tests, the framework can skip tests whose dependencies have failed. In addition, JExample introduces producer-consumer relationships, where the return value of a producer test is cached by the framework and later injected into the consumers as their fixture.
- Web frameworks are another field where "inversion of control is inverted back" [9]. For example, the Seaside framework uses continuations rather than the goto-like style of page-centric programming [1]. Rather than writing the web application page by page, one (or more) main methods capture the complete flow of the application, using call-backs and control flow structures provided by the Seaside API to handle page transitions.

Migration between APIs with such different mechanisms for inversion of control, some even based on conflicting paradigms, cannot necessarily be done with a simple set of structural refactorings. Additional runtime information might be required.

For example, to migrate from a tree- to a streaming-based XML framework, we might dynamically record all operations performed on the tree and then check if these operations can be re-ordered such that they can be applied in streaming fashion.

1 "There is some confusion these days over the meaning of inversion of control due to the rise of IoC containers; some people confuse the general principle here with the specific styles of inversion of control (such as dependency injection) that these containers use. The name is somewhat confusing (and ironic) since IoC containers are generally regarded as a competitor to EJB, yet EJB uses inversion of control just as much (if not more)." – Martin Fowler [2]

For example, to migrate a conventional XML query to a LINQ query, we might dynamically record the imperative sequence of instructions (which will certainly include many `for` loops and `if` statements) performed during the query and then check if we can find a LINQ equivalent that returns the same elements as the recorded `for` loops and `if` statements.

For example, XML namespaces in LINQ are represented with a dedicated `Namespace` class rather than plain strings. Thus, when migrating towards LINQ, we might dynamically record the usage of all strings in order to single out those that can be migrated as XML namespaces rather than strings.

For example, to migrate a J2EE application, we might dynamically record the resources requested from the EJB container in order to replace these calls with corresponding Spring injection annotations.

For example, to migrate from a conventional web framework to Seaside, we might dynamically record the page transitions for different tasks in order to generate high-levels methods that capture this flow of pages in one method. For example, given a Wiki application, one could record tasks such as login, create page, edit page, remove page, and generate a corresponding high-level method that captures the entire page flow of each of these tasks.

6. Concluding Remarks

In this paper, we investigate API migration where the mechanism for inversion of control changes. We propose to use information obtained from dynamic analysis for such API migration.

We provided examples taken from current and emerging industry trends. As a case-study we investigated in further detail how to migrate JUnit tests to JExample using information obtained from dynamic analysis, and propose two particular migrations steps that require dynamic analysis.

The first author of this paper is currently realizing the proposed steps as part of her Master's thesis.

Acknowledgments: The authors thank Ralf Lämmel for many discussions on API migration, from which emerged the proposed examples on API migration of XML frameworks.

We gratefully acknowledge the financial support of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" and the Swiss National Science Foundation for the project "Analyzing, Capturing and Taming Software Change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[1] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[2] M. Fowler. Inversion of control, obtained from Martin Fowler's wiki, June 2005.

[3] M. Gaelli, M. Lanza, O. Nierstrasz, and R. Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 114–123, 2004.

[4] L. Haensenberger. JExample. Bachelor's project, University of Bern, Mar. 2008.

[5] R. Johnsohn and J. Hoeller. *Expert One-on-One J2EE Development without EJB*. Wrox, 2004.

[6] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[7] A. Kuhn, B. V. Rompaey, L. Haensenberger, O. Nierstrasz, S. Demeyer, M. Gaelli, and K. V. Leemput. JExample: Exploiting dependencies between tests to improve defect localization. In P. Abrahamsson, editor, *Extreme Programming and Agile Processes in Software Engineering, 9th International Conference, XP 2008*, Lecture Notes in Computer Science, pages 73–82. Springer, 2008.

[8] Language Integrated Queries. http://plone.org/products/archgenxml.

[9] C. Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

[10] Simple API for XML. http://www.saxproject.org/.

[11] L. Wood, J. Sorensen, S. Byrne, R. Sutor, V. Apparao, S. Isaacs, G. Nicol, and M. Champion. *Document Object Model Specification DOM 1.0*. World Wide Web Consortium, 1998.