EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors

Philipp Bouillon, Jens Krinke, Nils Meyer, Friedrich Steimann

Schwerpunkt Software Engineering
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen
Philipp.Bouillon@gmail.com, krinke@acm.org, nils.meyer@jcom1.com, steimann@acm.org

Abstract. Unit testing is essential in the agile context. A unit test case written long ago may uncover an error introduced only recently, at a time at which awareness of the test and the requirement it expresses may have long vanished. Popular unit testing frameworks such as JUnit may then detect the error at little more cost than the run of a static program checker (compiler). However, unlike such checkers current unit testing frameworks can only detect the presence of errors, they cannot locate them. With EzUnit, we present an extension to the JUnit Eclipse plug-in that serves to narrow down error locations, and that marks these locations in the source code in very much the same way syntactic and typing errors are displayed. Because EzUnit is itself designed as a framework, it can be extended by algorithms further narrowing down error locations.

1 Introduction

All contemporary integrated development environments (IDEs) mark syntax errors in the source code, in close proximity of where they occur. In addition, static type-checking lets the compiler find certain logical errors (sometimes called semantic errors) and assign them to locations in the source in much the same way as syntax errors. Today, remaining errors in a program are mostly found by code reviews and by testing, in the context of XP and other agile approaches especially by pair programming and by executing unit tests.

JUnit is a popular unit testing framework. It is based on the automatic execution of methods designated as test cases. A test case usually sets up a known object structure, called test fixture, executes one or more methods to be tested on the fixture, and compares the obtained result with the expected one (including the possible throwing of exceptions). Because the expected result must be determined by some other way than executing the method(s) under test (the test oracle), test cases are usually rather simple. However, there is no theoretic limitation on the complexity of test cases, other than that they must run without user interaction and that the result must be repeatable.

JUnit as currently designed reports errors in the form of failed tests. Contemporary IDE integration of JUnit lets the developer navigate from the test report to the failed test case, that is, to the test method that discovered an unexpected result. However,

the test method only detects the presence of a programming error — it does not contain it. The developer must infer the location of the error from the failed test case, which is not necessarily trivial. But even if it is, navigating from the error report to the source of the error currently requires a detour via the test case. Transferred to syntax and type checking, this would correspond to navigating from an error report to the error source via the syntax or typing rule violated, which would clearly be considered impractical.

Our ultimate goal is to lift unit testing to the level of syntactic and semantic checking: a logical error detected by a unit test should be flagged in the source code as close as possible to the location where it occurred. As a first step in this direction, we present here for the first time an extension of the JUNIT integration in ECLIPSE, named EZUNIT, that provides basic reporting and navigation facilities, and that accommodates for algorithms and procedures serving to narrow an error location.

2 The Framework

In JUNIT 4, test cases are tagged with the `@Test` annotation. When adding a test case through ECLIPSE's New > JUnit Test Case... menu and selecting a method to be tested, the test method is automatically annotated with a Javadoc tag saying that this method is a test method for the method for which it was created. We raise this comment to the level of an annotation, named `@MUT` (for method under test), and allow more than one method under test to be listed. This accommodates for the fact that the tested method may call other methods, which may also be tested by the test case, and that the initially called method may be known to be correct, while other methods it calls are not. To help the programmer with generating the annotations, a static call graph analysis of the test method is provided, listing all methods the test method potentially calls. From this the developer can select the methods intended to be tested by this test case. The generated list can be automatically filtered by an exclusion/inclusion of packages expression (e.g., excluding all calls to the JUNIT framework).

The `@MUT` annotations are exploited in various ways. Firstly, they aid with the navigation between test methods and methods under test: via a new context menu in the Outline view of an editor, the developer can switch from a method under test to the methods testing it and vice versa, without knowing or looking at the implementation of a method. Secondly, and more importantly, whenever a test case fails during a test run, corresponding markers are set in the gutter of the editor, in the Package Explorer, and in the Problems view. Fig. 1 shows a test method (from the well-known Money example distributed with JUNIT) with a corresponding `@MUT` annotation, and the hints provided by a test run after an error has been seeded in the `add()` method of `Money`.

Surely, in the given example associating the failed `testSimpleAdd()` with `add()` in `Money` is not a big deal, but then spotting the error in `add()` without knowledge of the test method isn't either, so that the developer saves one step in pinning down and navigating to the error. In more complex cases, especially where there is more than one method to which blame could be assigned, checking all methods that may have contributed to the failure requires more intimate knowledge of the test case. With EZUNIT, the essence of this knowledge, namely which methods are being tested

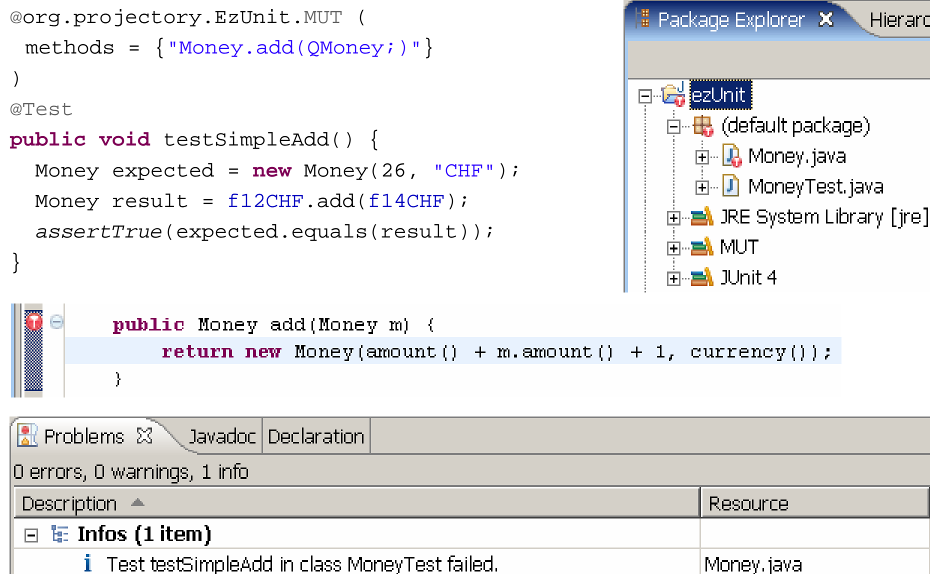

Fig. 1. An `@MUT` annotated test case and the markers it creates if the method under test is faulty.

by the test case, is contained in the Problems view and the various error adornments. By going through these methods one by one, the developer can look for a potential error, fix it, and rerun the test cases until the problem disappears. During this process, the developer can always consult the test case to get additional hints, simply by using the quick navigation facilities offered by the EzUnit framework.

The basic functionality offered by EzUnit is rather simple, one may even say simplistic. However, because EzUnit is designed as a framework, this basic functionality can be extended by adding methods capable of narrowing the possible source of the error, thereby providing assistance to the developer that is as yet unavailable. We are currently exploring various such extensions. The framework is downloadable as an Eclipse bundle from the EzUnit update site found at `http://www.fernuni-hagen.de/ps/prjs/EzUnit/update`.

3 Discussion and Related Work

Our approach is simple. Its appeal comes from the fact that we can exploit the special character of unit tests, namely that they are 100% repeatable (in that every run creates exactly the same object structure and calls exactly the same methods on it): several approaches to error locating practically intractable for the general case (such as program slicing [1, 2]) can therefore likely be used in our setting, simply because there is no dependency on input or other uncontrollable variation. With the provision of our framework, we hope to attract other researchers to contribute their ideas.

Our approach is somewhat related to David Saff's work on continuous testing [1, 4]. Continuous testing pursues the idea that test execution, like compilation, can be performed incrementally and in the background. Whenever a developer changes something and triggers a (successful) compilation, all tests whose outcome is possibly affected by that change are automatically rerun. Thus, like our own work Saff's raises unit testing to the level of syntactic and semantic (type) checking, yet it does so in an *orthogonal dimension*: continuous testing is about *when tests are executed*, our work is about *how the results are interpreted and presented*. It should be interesting to see whether and how the two approaches can be combined into one, particularly since the mutual dependency of testing and program units under test is common to both of them.

4 Conclusion

While unit testing automates the detection of errors, their localization is currently still an intellectual act. By providing a simple framework that

- allows the developer to select — based on the result of a static call graph analysis — which methods are being tested by each test case, that
- enables rapid switching between test methods and methods under test, and that
- marks failed tests as adornments in the editor and other representations of the source code,

we have laid the technical groundwork for a symptom-to-diagnosis mapping for programming errors. Extensions helping to narrow down error locations are easily conceived and added. In fact, we believe that the best results can be expected from applying several algorithms in parallel, and from combining the evidence collected from each. We have provided the framework for this.

References

1. H Agrawal, JR Horgan "Dynamic program slicing" Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (1990) 246–256.
2. B Korel, J Laski "Dynamic program slicing" *Information Processing Letters* 29:3 (1998) 155–163.
3. D Saff, MD Ernst "Reducing wasted development time via continuous testing" in: *ISSRE 2003*, *14th International Symposium on Software Reliability Engineering* (2003) 281–292.
4. D Saff, MD Ernst "An experimental evaluation of continuous testing during development" in: *ISSTA 2004, International Symposium on Software Testing and Analysis* (2004) 76–85.