Linguistic Support for Unit Testing

UUCS-07-013

Kathryn E. Gray
University of Utah
kathyg@cs.utah.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract

Existing systems for writing unit tests exploit built-in language constructs, such as reflection, to simulate the addition of testing constructs. While these simulations provide the minimally necessary functionality, they fail to support testing properly in many instances. In response, we have designed, implemented, and evaluated extensions for Java that enable programmers to express test cases with language constructs. Not surprisingly, these true language extensions improve testing in many different ways, starting with basic statical checks but also allowing the collection of additional information about the unit tests.

1. Testing Failure

Stories of catastrophic software failure due to a lack of sufficient testing abound. Proponents of test-driven development regale audiences with these stories to encourage developers to write adequate test suites. These stories, and the availability of testing frameworks such as JUnit, have motivated programmers across the board to develop basic unit test suites.

In writing unit tests, individual test cases should check that applicative methods (also known as observers) compute the expected results and that imperative methods (aka commands) affect the proper parts of the object's state (and nothing else). In addition, programmers need to be concerned with failures due to exceptions, especially ensuring that methods fail gracefully and as expected. While setting up tests for applicative methods tends to be straightforward, testing imperative methods and exceptional situations tends to require complex work. Specifically, it often demands calling a specific sequence of methods, and may benefit from performing a number of specific tests in sequence.

As a result, even with the wide-spread support for testing, programmers still don't develop sufficiently rigorous test suites, because creating and maintaining them remains a large burden. A thorough study of publicly accessible test suites (such as those in sourceforge) suggests that programmers construct few tests that check for modified state and even fewer (practically none) that exercise failure conditions.

We conjecture that a part of the problem in creating test suites lies with the lack of direct linguistic support for testing. This lack of testing constructs in the programming language itself has several symptoms, including silent failures and overly complex test case formulations. More precisely, our analysis shows that programmers fail to adhere to the protocol of the test suite, forgetting to prefix a method name with "test" or specifying formal parameters for testing methods when they take none. In such cases, the unit testing framework often simply ignores the tests without informing the programmer. Similarly, few (if any) programming languages allow the simulation of constructs that make it easy to set up exception handlers for tests of "exceptional" methods. Hence, programmers often don't test such scenarios or, if they do, it becomes difficult to maintain such tests due to the syntactic clutter.

A consequence of the lack of specific testing constructs is that compilers don't understand tests. Reflection constructs—the basis of JUnit—are simply too impoverished to communicate with compilers (properly). As a result, compilers don't gather enough information about the testing process. For failed test cases, information gathering makes testing truly rewarding; in contrast, lack of information makes it difficult to locate the source of bugs and to fix them. Put differently, on failure, testing tools should provide meaningful information on the actual vs desired behavior, the source of the failure, and information regarding the state of the program for the test. This kind of compiler-informed feedback from the testing tool would assist programmers in correcting errors quickly and economically.

To test our conjecture, we have designed and implemented an extension for Java, called TestJava, that includes constructs for testing. The compiler/IDE for TestJava gathers simple pieces of additional information to support error-