Continuous testing in Eclipse

David Saff Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab
Cambridge, MA, USA
{saff,mernst}@mit.edu

ABSTRACT

Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background, providing rapid feedback about test failures as code is edited. It reduces the time and energy required to keep code well-tested, and it prevents regression errors from persisting uncaught for long periods of time.

Categories and Subject Descriptors: D.2.5 (Testing and Debugging): Testing tools
General Terms: Measurement, Experimentation, Human Factors
Keywords: continuous testing, testing, development environments

1. INTRODUCTION

Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background as the developer edits code. It provides developers rapid feedback regarding errors that they have inadvertently introduced.

It is good practice to use a regression test suite while performing development tasks such as enhancing or refactoring an existing codebase. During development, running the test suite indicates whether the developer is making progress, and catches regression errors early. The longer a regression error persists without being caught, the larger its cost: more code changes must be considered to find the changes that directly pertain to the error, the code changes are no longer fresh in the developer's mind, and new code written in the meanwhile may also need to be changed as part of the bug fix.

Running tests (remembering to run the tests, waiting for them to complete, and returning to the task at hand) is a distraction from development. Developers may employ test case selection [4] and prioritization [5] to reduce the cost of running tests. They may also continue editing the code while tests run on an old version, but this further complicates reproducing and tracking down errors.

Continuous testing uses real-time integration with the development environment to asynchronously run tests against the current version of the code and non-intrusively notify the developer of regression errors. The version of the code being tested is kept in sync with the version being edited. Continuous testing can be combined with selection, prioritization, or other approaches. The developer no longer has to consider when to run the tests, and errors are caught more quickly, especially those that the developer had no cause to suspect.

2. EVALUATION

We have performed two separate evaluations of continuous testing: a prospective evaluation and a controlled human experiment.

The first experiment prospectively evaluated continuous testing [6], before any continuous testing tool was built. The results suggested that continuous testing could have reduced development time for two single-developer software projects by 10–15%, which is substantially more than would have been achieved by changing manual test frequency or reordering tests.

The controlled human experiment [7] evaluated whether students using continuous testing are more successful in completing programming assignments, without producing detrimental side effects like distraction and annoyance. In the experiment, 22 students in a software engineering course completed two unrelated programming assignments as part of their normal coursework. The assignments supplied partial program implementations and tests for the missing functionality, simulating a test-first development methodology. All participants were given a standard Emacs development environment, and were randomly assigned to groups that were provided with continuous testing, just continuous compilation, or no additional tools. Data was gathered from remote monitoring of development progress, questionnaires distributed after the study, and course records. In all, 444 person-hours of active programming were observed.

The experimental results indicate that students using continuous testing were statistically significantly more likely to complete the assignment by the deadline, compared to students with the standard development environment. Continuous compilation also statistically significantly increased success rate, though by a smaller margin; we believe this is the first empirical validation of continuous compilation. Furthermore, the tools' feedback did not prove distracting or detrimental: tool users suffered no significant penalty to time worked, and a large majority of study participants had positive impressions of the tool and were interested in using it after the study concluded.

3. ECLIPSE PLUG-IN

We have implemented a continuous testing infrastructure for the Java JUnit testing framework and for the Eclipse and Emacs development environments. We will demo and discuss the Eclipse version, which is publically available[1] and is in daily use by industrial developers.

The continuous testing tool represents an incremental advance over existing technology in Eclipse. Eclipse automatically compiles when the user saves a buffer, indicates compilation problems both in the text editor window and in the task list, and provides an integrated interface for running a JUnit test suite.

Continuous testing builds on the automated developer support in Eclipse to make it even easier to keep Java code well-tested when a developer has a JUnit test suite. If continuous testing is enabled, Eclipse runs tests in the background as the developer edits code, and notifies the developer if any of them fail or cause errors.

3.1 Error running and notification

Saving a code change triggers a run of the associated test configuration. Failing tests are indicated in several ways. A problem is added to the Problems view. The icon for this problem is a red ball, similar to a compile error, but with a T for "test failure". Double-clicking the item in the Problems view opens and highlights the test method that failed. The method name is marked with a wavy red underline and a red T in the margin.

The plug-in follows the principle of minimal distraction in several ways. Tests are run only when a developer saves, to avoid testing in states the developer knows are inconsistent. If there is a compile error anywhere in the project being edited or the project containing the tests, or in those projects' dependencies, no tests are run. Failure notifications are designed to be easily noticed when a developer looks for them, but not to break the developer's train of thought otherwise.

As soon as a change is introduced affecting a test that is currently marked as failing, the icon for that failure is changed from red to gray as the test is scheduled for running. If the test fails again, the icon is changed back to red. If it succeeds, the marker is removed. In this way, users can tell the difference between (red icon) failures known to be caused by the current version of the code and (gray icon) failures that are only known to have been present in a prior version, but are in the process of being rerun. This is consistent with Eclipse's handling of compile errors during automatic compilation.

3.2 Test prioritization and selection

By default, the Eclipse JUnit integration always runs all the tests in a suite in the same order (although it is sometimes difficult to predict what that order will be). This can be frustrating, if the test of most interest to the developer is run late in the suite. To alleviate this problem, the continuous testing plug-in allows for several kinds of test prioritization, configured through properties on the launch configuration. Prioritization can be based on which code has changed, and the results of previous test runs. We have found [6] that a useful simple strategy is Most Recent Failures First, because tests that are currently failing or have recently failed are likely to be of the greatest interest to a developer, and are more likely to fail again. We have also added support for custom test filters, allowing users to choose between different strategies of determining what subset of the tests in a suite must be re-run in response to a particular change.

[1] `http://pag.csail.mit.edu/continuoustesting`

3.3 Hotswapping and remote execution

For fast-running test suites, the time required to start up a Java virtual machine and set up global state can be much greater than that required to run the tests themselves. Continuous testing can run in hotswapping mode, in which a virtual machine dedicated to testing is kept running. Using the Java Debugging Interface, new class definitions are loaded into the running JVM and tests are rerun.

If a computationally expensive test suite must share a processor with the developer's programming environment, both may suffer. Therefore, continuous testing can run tests on a remote test server, and display results in real time in the local development environment.

4. RELATED WORK

Continuous testing can be viewed as a natural extension of continuous compilation [8]. Modern IDE's (integrated development environments) with continuous compilation supply the developer rapid feedback by performing continuous parsing and compilation, indicating (some) syntactic and semantic errors immediately rather than delaying notification until the user explicitly compiles the code.

Continuous execution [2], Programming by Example [1], and Editing by Example [3] all provide continuous feedback to developers about the results of their program on one or more inputs as the program changes. Our work abstracts from the entire output to the boolean result of each individual test case.

5. REFERENCES

[1] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.

[2] P. Henderson and M. Weiser. Continuous execution: The VisiProg environment. In *ICSE*, pages 68–74, Aug. 1985.

[3] R. P. Nix. Editing by example. *ACM TOPLAS*, 7(4):600–621, Oct. 1985.

[4] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE TSE*, 22(8):529–551, Aug. 1996.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE TSE*, 27(10):929–948, Oct. 2001.

[6] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, Nov. 2003.

[7] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, pages 76–85, July 2004.

[8] M. D. Schwartz, N. M. Delisle, and V. S. Begwani. Incremental compilation in Magpie. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 122–131, Montreal, Canada, June 1984.