Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 107 (2004) 103–117
www.elsevier.com/locate/entcs

Continuous Testing in Eclipse

David Saff Michael D. Ernst

saff@mit.edu, mernst@mit.edu
MIT Computer Science & Artificial Intelligence Lab
Cambridge, MA, USA

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 source code is edited. It is intended to reduce the time and energy required to keep code well-tested, and to prevent regression errors from persisting uncaught for long periods of time.
This paper reports on the design and implementation of a continuous testing feature for Java development in the Eclipse development environment. Our challenge was to generate and display a new kind of feedback (asynchronous notification of test failures) in a way that effectively reuses Eclipse's extensible architecture and fits the expectations of Eclipse users without interfering with their current work habits. We present the design principles we pursued in solving this challenge: present and future reuse, consistent experience, minimal distraction, and testability. These principles, and how our plug-in and Eclipse succeeded and failed in accomplishing them, should be of interest to other Eclipse extenders looking to implement new kinds of developer feedback.
The continuous testing plug-in is publicly available at
http://pag.csail.mit.edu/~saff/continuoustesting.html .

Keywords: Continuous testing, regression testing, software productivity, Eclipse plug-in, asynchronous notification

1 Introduction

A Java developer using Eclipse can take advantage of continuous compilation; after every saved change, a compilation is (incrementally) run, which brings the generated class files up to date with the source files, and quickly notifies the developer if any compilation errors are encountered. This feedback may sometimes be useless or redundant. However, the popularity and effectiveness of continuous compilation indicate that the feedback provided is valuable enough, and processors are fast enough, to make it worthwhile.

doi:10.1016/j.entcs.2004.02.051

Continuous testing [8,7] is a new feature for software development environments. It is based on the idea that processors have become fast enough not only to keep code compiled as it is changed, but to keep it tested, as well. As the code is changed, tests are selected and run in the background, and failures are flagged quickly for the developer to address. Testing becomes automatic; it catches regression errors earlier, gives developers more confidence that they are making progress, and requires less effort.

The purpose of continuous testing is to reduce two varieties of wasted time related to testing. The first source of wasted time is time spent running tests: remembering to run them, waiting for them to complete, and returning to the task at hand after being interrupted to run tests. The second source of wasted time is time spent performing development while errors exist in the system. Performing development in the presence of an error lengthens the time to correct the error: 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. The longer the developer is unaware of the error, the worse these effects are likely to be.

We implemented a preliminary version of continuous testing for Emacs, and used it for a user study on student developers (see Section 2.1). Encouraged by the results, we built a full-featured implementation of continuous testing as a plug-in for the Eclipse integrated development environment [3].

1.1 Design goals

Eclipse already contains support for integrated unit testing, automatic asynchronous compilation, and automatic notification of compile errors. This meant that much of the infrastructure for continuous testing already existed, both in the technical implementation, and in user expectations and metaphors. This allowed us to provide more functionality more quickly than we could in Emacs, and we hope it will allow us to develop a user base more quickly.

Developers using our plug-in can associate a test suite with each Java development project. Whenever the code in the project changes, the test suite is run asynchronously, and any test failures are indicated to the developer similarly to compilation errors. To reduce the average time needed to find a failure, developers can enable test prioritization, which tries to run first the tests most likely to fail. The plug-in does not interfere with manual use of JUnit, and it uses unobtrusive signals to indicate its progress to an interested developer, without being overly distracting.

In building this plug-in, we followed these architectural principles:

- **Reuse**: Whenever possible, reuse existing functionality. Eclipse was designed for extensibility, and in many places its plug-in architecture allowed us to simply contribute the functionality we wanted in the appropriate place.
- **Future reuse**: When reuse is impossible, design for future reuse. In some cases, the plug-in we wished to extend did not provide an extension point where we needed it, so it was necessary to cut and paste code from the existing plug-in into our plug-in. In these cases, we kept the copied code as close to the original as possible, so that if extension points are provided in the future, it will be easy to make use of them.
- **Consistent experience**: At all times, maintain a consistent front end to the user. Consistency meant not changing behavior in old circumstances (for example, leaving the existing JUnit result view to only show the result of manually launched tests), and allowing users to bring old metaphors to bear on new functionality (test failures look similar to, but are distinguishable from, compile errors).
- **Minimal distraction**: The plug-in uses common-sense heuristics to avoid annoying the user with obviously redundant information. If there are compile errors in the code under test, test failures are uninteresting. If two projects both reference the same test, its failure should be indicated only once. Any status change other than a newly failing test should be indicated only subtly, so that the developer can easily ignore it.
- **Testability**: Make the plug-in as testable as possible. Testability influenced our design in several ways, especially by leading us to build in API's for monitoring progress and to add additional listeners for test runs and marker creation and deletion. Testability also encouraged separating presentation and application logic, especially in the configuration dialogs.

We have measured productivity and morale benefits for users of continuous testing, so the public availability of a usable implementation should interest many Eclipse users. However, Eclipse developers and extenders may also be interested in the way that the above principles played out in the current design of the plug-in. It was encouraging how well Eclipse's modularity lent itself to a design that tied together several previously unrelated components. However, we also highlight opportunities for improvement, both in the plug-in and Eclipse.

This paper is organized as follows. We first review related work on continuous testing and related topics (Section 2). Then we describe the design of the continuous testing plug-in, both its user interface (Section 3) and technical architecture (Section 4). Finally, we describe opportunities to improve both our plug-in and the Eclipse frameworks on which it is based (Section 5), and

Treatment	N	Correct
No tool	11	27%
Continuous compilation	10	50%
Continuous testing	18	78%

Fig. 1. Treatment predicts correctness. "N" is the number of participants in each group. "Correct" means that the participant's completed program passed the provided test suite.

conclude (Section 6).

2 Related work

2.1 Continuous testing

We have performed two different types of study in order to evaluate the benefits of continuous testing.

First [8], we measured real development projects to estimate *wasted time*, consisting of preventable extra fixing cost added to the time spent running tests and waiting for them to complete. In the monitored projects, wasted time accounted for 10–15% of total development time (programming and debugging). We also developed a model of developer behavior, from which we inferred the *ignorance time* (from introduction to discovery) and the *fix time* (from discovery to fix) for each error. We found that ignorance time and fix time are correlated, and concluded that reducing ignorance time should reduce fix time. For the monitored projects, a continuous testing tool could reduce wasted time by 92–98%, which is more effective than test prioritization or manually testing more frequently.

Second [7], we performed a controlled experiment comparing three groups of student developers: one provided with continuous testing, one provided only with continuous compilation, and one provided with no asynchronous notification of any type of development error. All participants used Emacs as their Java development environment. Student developers using continuous testing were three times more likely than the control group to complete two different one-week programming assignments (which were part of their normal coursework); see Figure 1. These statistically significant effects are due to continuous testing: they could not be explained by other incidental features of the experimental setup, such as time worked, regular testing, or differences in experience or tool preference. Students with continuous compilation were twice as likely to complete the task as those without; this is the first empirical evidence of continuous compilation's effectiveness, to our knowledge.

A majority of users of continuous testing had positive impressions, saying that it pointed their attention to problems they would have overlooked, and

it helped them produce correct answers faster and write better code. Staff said that students quickly built an intuitive approach to using the additional features. 94% of users said that they intended to use the tool on coursework after the study, and 90% would recommend the tool to others. Few users found the feedback distracting, and no negative effects on productivity were observed.

2.2 Related ideas

Continuous testing can be viewed as a natural extension of continuous compilation, which is standard in Eclipse, and also of Extreme Programming [1], which emphasizes the importance of unit test suites that are run very frequently to ensure that code can be augmented or refactored rapidly without regression errors. Gamma and Beck [4], independently of the current authors, followed these threads to propose that "Test failures are compile errors" as part of a working example of contributing to Eclipse. Gamma and Beck's plug-in includes an editor for excluding tests from automatic execution. Ours includes asynchronous test execution through the build framework, detailed specification of the launch configuration and environment for automatic tests, test prioritization, and other enhancements. The two plug-ins were designed independently and concurrently, but we did refer to Gamma and Beck's implementation for ideas of how to integrate with the workspace and marker frameworks in Eclipse and build a testable plug-in.

Henderson and Weiser [5] propose *continuous execution*. By analogy with a spreadsheet such as VisiCalc, their proposed VisiProg system (which they hypothesized to require more computational power than was available to them) displays a program, a sample input, and the resulting output in three separate automatically updated windows. Rather than continuously maintaining a complete output, which would be likely to overwhelm a developer, the test suite abstracts the output to a simple indication of whether each individual test case succeeds.

3 User experience

3.1 Overview

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. It is most useful in situations where a developer has

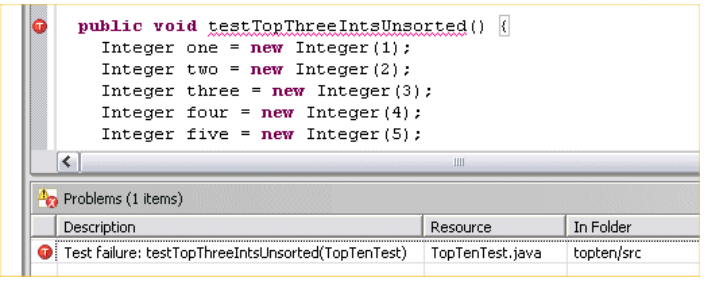

Fig. 2. A test failure notification is shown in three ways: in the Problems view, in the source code margin, and as a wavy red underline under the method name.

a test suite while changing code: when performing maintenance, bug fixing, refactoring, or using test-driven development [2].

Continuous testing builds on two other features of Eclipse:

- **JUnit integration** : Test suites run under continuous testing give the same information, in the same format, that developers already get from Eclipse's JUnit integration. However, continuous testing also helps to automate the choice of when to run tests, which tests to run, and in which order. Continuous testing does not interfere with the developer's concurrent use of JUnit, following the principle of consistent experience.
- **Continuous compilation** : As soon as a developer writes code that contains a compilation error, Eclipse highlights the error in the text and the margin, and creates a task in the Problems view. This makes it easy both to quickly learn of problems, and to step through and fix problems in an orderly fashion. With continuous testing, the developer gets the same kind of notification when writing or editing code that causes a test to fail. However, test failures differ from compile errors in that test failures sometimes cannot easily be tracked to a single line of code, and test failures can provide more information, such as a backtrace, than compile errors.

The plug-in is installed through Eclipse's built-in Software Updates utility. Enabling and configuring continuous testing is done on a per-project basis, through Properties dialogs. The plug-in comes with a full source plug-in for easier extension and debugging, and, following the principle of testability, a test suite using Eclipse's JUnit Plug-in Test support.

3.2 Error running and notification

Saving a code change triggers a run of the associated test configuration. The status bar displays a notification of each test as it starts and completes. Failing tests are indicated in several ways (Figure 2), similarly to a compile error:

- 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". Right-clicking this item in the Problems view opens a context menu with options to re-run the failing test, delete the marker, or view the stack trace associated with the 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.
- Tests being run by continuous testing are not shown in the standard JUnit output view, nor do they appear on the Run History. These places are reserved for manual launches. Details on the current automatic test run can be found in the Continuous Testing view, but we expect users to leave this view hidden most of the time. It would be ideal to change the standard JUnit interface in a way that recognizes and indicates both manual and automatic launches, but this would require changes to Eclipse (see Section 5).
- 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.
- If two projects are associated with the same test suite, only one failure marker per test is ever created.

As soon as a change is introduced affecting a test that is currently marked as failing (that is, the test is in the suite associated with the changed project), 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.

3.3 Test prioritization

By default, the Eclipse JUnit integration always runs 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

Most recent failures first: Tests that have failed most recently are ordered first.
Quickest test first: Tests are ordered in increasing runtime; tests that complete the fastest are ordered first.
No reordering: Tests are run in the default JUnit order on every change.
Round robin: Like "No reordering", but if testing is interrupted, start testing again at the interrupted test and wrap around to the beginning.
Most frequent failures first: Tests that have failed most often (have failed during the greatest number of previous runs) are ordered first.
Random: Tests are run in random order, without repetition.

Fig. 3. Test case prioritization strategies offered by the plug-in, in decreasing expected order of effectiveness [8].

allows for several kinds of test prioritization [8,9,6], configured through properties on the launch configuration. This is a relatively independent feature that could be implemented for all JUnit launches, but is currently only enabled for continuous testing. The six currently-supported strategies (Figure 3) all remember results of previous test runs to help prioritize the currently scheduled tests. See Section 4.2 for technical details. We have found [8] that the most useful of these strategies 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.

4 Plug-in Design

Here, we discuss the design features of Eclipse that influenced the plug-in (4.1), and the design of the plug-in itself (4.2).

4.1 Eclipse design overview

In this section, we put forth some of the salient design points of Eclipse that influenced the design of the continuous testing plug-in. We focus on the static structure and dynamic behavior of the auto-building framework and the JUnit launch framework. This is a high-level overview of the Eclipse design, omitting lower-level details where possible.

4.1.1 Auto-building

Figure 4 shows the static structure of projects, source files, and builders in Eclipse. Each project is at the root of a tree of Resources, which include folders, source files, and other file types. For simplicity in this discussion, we

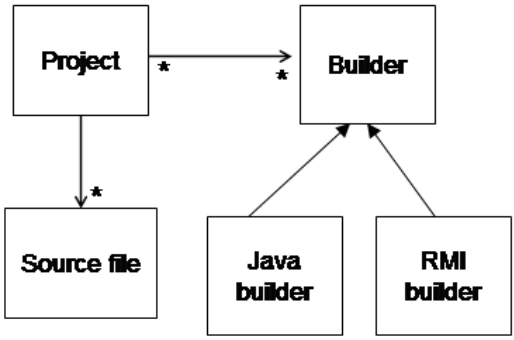

Fig. 4. Static structure of auto-build framework in Eclipse. Filled arrowheads indicate a subtyping relationship. Unfilled arrowheads indicate an association. Asterisks show multiplicity in one-to-many and many-to-many relationships.

will consider each project as a simple container of source files. Each project also references an ordered list of zero or more builders, which are contributed through an extension point. Each builder is responsible for responding to incremental changes to keep project state consistent. For example, the Java builder uses a compiler to keep built class files up to date with the Java source files in the project, whereas the RMI builder keeps generated skeleton and stub files up to date with any RMI interfaces in the project.

Figure 5 shows the behavior of the auto-building framework when the user changes one of the project source files. A delta is created and applied to the source file. A global build manager is notified of the delta, and decides which projects are affected. The build manager then starts a new asynchronous job to actually carry out the autobuild in a different thread. This new job passes control to each builder for the project in turn. If a builder encounters an error while responding to the delta (such as a compile error or classpath problem), it can create one or more markers that attach to source files in the project at particular places. The GUI is updated to indicate these markers.

4.1.2 JUnit launching

Eclipse provides a launching framework for users to start new Java processes to execute code within a project (the code is kept constantly compiled by the auto-build framework detailed above). Figure 6 shows the persistent static structure that enables this functionality.

Eclipse keeps references to a set of user-defined or automatically generated launch configurations. Each configuration references one or more projects that (by themselves or in referenced dependent projects and libraries) contain all of the classes needed for the process to run. Each configuration is of a particular launch type, such as for running Java applications, for running JUnit tests

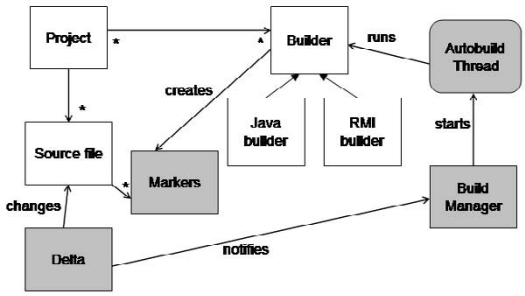

Fig. 5. Dynamic behavior of auto-build framework in Eclipse when the user changes a source file. Classes shown for the first time in this figure are highlighted in gray. Asynchronous jobs running in a different process or thread are shown as rounded boxes.

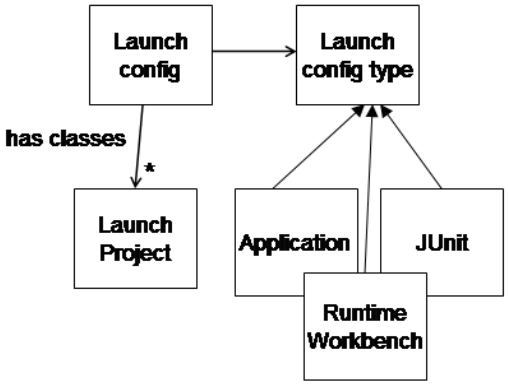

Fig. 6. Static structure of launching framework in Eclipse.

using a graphical test runner, and for starting a new instance of Eclipse that includes a plug-in under development.

Figure 7 shows how this is put to use when a JUnit launch is initiated. A new JVM instance is created, with all of the needed projects and libraries placed in its classpath. The main class in this new JVM is a remote test runner. A client in the Eclipse JVM opens a socket to the remote test runner, which the runner uses to communicate its progress and the outcome of each test. The client uses this feedback to update the test result view.

4.2 Design for continuous testing

Continuous testing combines the auto-building and launching frameworks. It uses a modified JUnit launch type to run tests and create markers for tests that have failed, and uses the auto-build framework to keep these markers up to date with the state of the code and tests. In keeping with the principle of reuse, new classes inherit from existing classes whenever possible, or are

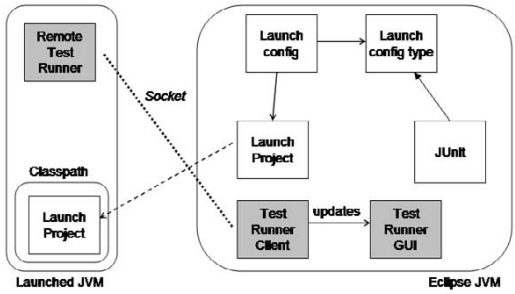

Fig. 7. Dynamic behavior of launching framework for JUnit in Eclipse.

otherwise copied from existing classes, with minimal changes

Figure 8 shows the additions to the static structure made by the continuous testing plug-in. First, a new continuous testing builder type is added to handle keeping the failure markers up to date. Projects that have the continuous testing builder enabled are annotated with a reference to a launch configuration, of a new continuous testing launch type, that will be run whenever the project detects a delta. Also, to enable test prioritization, a persistent dataspace per continuous testing launch configuration maintains testing metadata. A harness that wanted to run the quickest tests first, for example, would store the most recent observed duration for each test.

Figure 9 shows how everything works together when a source file is changed. As before (although unshown here), a delta is created, the build manager is notified, an autobuild thread is started, and each builder in turn is allowed to start. Assuming that Java compilation completes successfully, control is passed to the continuous testing builder, which uses the launch framework to launch the configuration attached to the changed project. The continuous testing configuration starts up modified versions of the remote test runner and local test runner client. It also uses a temporary file to pass the prioritization harness's metadata to a new instance of the harness within the launched JVM. The modified remote test runner asks the harness for the order in which tests should be run. The modified test runner client updates the test result GUI, feeds back changes to the harness metadata, and creates test failure markers to attach to the test project.

5 Future work

There is remaining work that can be done to improve the design of both our plug-in and Eclipse, if plug-ins such as ours are considered valuable.

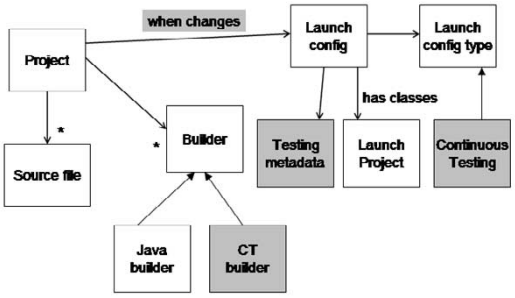

Fig. 8. Static structure of continuous testing plug-in. Continuous-testing specific features are highlighted in gray.

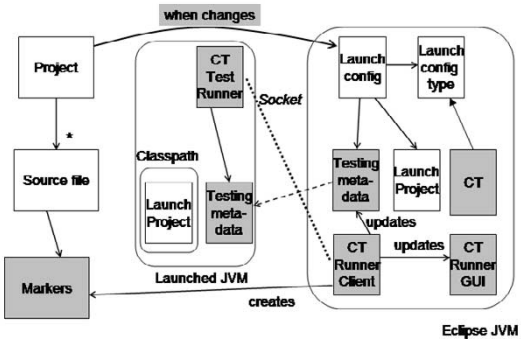

Fig. 9. Dynamic behavior of continuous testing plug-in.

5.1 The plug-in

The most important next step for the continuous testing plug-in is to break it up into several component plug-ins that communicate through extension points, so that Eclipse users and extenders can pick and choose from the functionality that is currently bundled up into one plug-in. This will allow future extenders to themselves follow the principle of reuse. We imagine separate plug-ins for

- test prioritization within JUnit,
- creating markers based on test failures,
- associating test launches with projects, and
- using a builder to automatically start tests when a project changes.

There are many possible future enhancements to the plug-in, including:

- Continuous testing could make use of the hotswapping JVM used by the Eclipse debugger to change code under test on the fly, without having to stop and restart the entire test process.

- Prioritization and selection could be based on the source delta, and not just on the results of previous runs, if the analysis required could be made fast enough. For example, tests that are known to exercise recently-changed methods could be given higher priority.
- Eclipse's quick fix feature could be extended to provide quick fixes to common dynamic exceptions such as ClassCastExceptions.
- It might be possible to use the fact that tests are often being run in the background to offer "instant variable watching": for example, one could imagine hovering a mouse over a variable to see what that variable's value is the next time that the test process executes that line of code. The authors find themselves already approximating this feature by adding debug print statements to their code, and watching the values printed in the Console as tests are automatically run.
- Continuous testing should be extended to Plug-in Development Environment test suites.
- It should be possible to associate launch configurations with different packages or even files, rather than entire projects.
- Just as continuous compilation on large programs is infeasible without incremental compilation, continuous testing on large test suites will require some form of incremental testing. For tests suites with long-running tests, approaches such as prioritization are insufficient: it will be necessary to use data collected on previous test runs to run only those *parts* of tests that may reveal recently-introduced errors. We are currently investigating *test factoring* to split long-running tests into smaller unit tests, even when the programmer has not provided unit tests.

5.2 Eclipse

It is a tribute to Eclipse that two mechanisms, each containing cooperating asynchronous processes, could be tied together in the intimate way we have done here, without any indication that such a feature was an original intent of Eclipse's designers and implementers. It is also inescapable that no system can anticipate every possible way that contributors might try to extend it. We found we wanted some additional extension points and API's, including:

JUnit integration: The JUnit integration in Eclipse assumes that the user is only interested in the progress and results of a single run of a single test configuration — the most recent configuration manually launched. This leads to a conflict when continuous testing is introduced:

- The configuration most recently manually launched is not necessarily the

most recently run. We handle this by having a separate Continuous Testing Results view that closely replicates the JUnit view, but it would be better to combine these.

- The user may be interested in reviewing the results of multiple test configurations, or the same configuration invoked at multiple times against different versions of the code. This conflict is already present without continuous testing, but is exacerbated by the fact that the user of continuous testing does not have direct control over the frequency or ordering of his test runs. The current plug-in does not resolve this conflict.

These conflicts suggest an extension of the JUnit interface that presents the results of multiple test runs (both manual and automatic) across multiple versions. This would require the classes implementing the JUnit launch configuration type and the JUnit view to be better designed for inheritance; our current plug-in was forced to override too many methods in order to only subtly change their behavior.

Problems view: We believe, based on our own usage, that developers are used to using the Problems view for quick notification of problems with their code. Rather than introduce a separate Test Failures view or Continuous Testing perspective, we felt it was more appropriate to reuse the Problems view and Java perspective. However, it was important to visually distinguish test failures from compile errors, and test failures against the current version from test failures against a previous version of the code, by changing the icon image and color (see Section 3). The current implementation of the Problems view hard-codes the icon images used, requiring us to use an unfortunate hack using the Java reflection API to achieve the intended behavior. The Problems view should use the same easily-extensible icon scheme as is used for showing markers in the margins of source files.

Further connections between resources and launches: It seems likely to us that once users become accustomed to the connection between projects and launch configurations used by continuous testing, they may see other applications of the idea. For example, each project (or finer-grained resource) could have a user-settable default debug configuration that could be invoked with a single keypress when any resource in that project was selected, or launch histories could be kept on a per-project basis. Parts of continuous testing's configuration interface could be subsumed into such a framework.

6 Conclusion

The continuous testing plug-in is publicly available at
http://pag.csail.mit.edu/~saff/continuoustesting.html .

The authors are using it themselves on Java application development, and have found that it is stable and unobtrusive. We believe that we are catching errors earlier, and we are able to focus on our development goals with confidence that we know the effects of our changes quickly.

From our experience in the design and implementation of this plug-in, Eclipse has shown great promise as a platform for the evaluation and distribution of experimental tools for software productivity. Individual components are well-designed for extensibility, both technically, and in terms of creating user metaphors that are easily extended to new functionality. The principles and experiences discussed in this paper should prove valuable as more researchers look to Eclipse as a proving ground for new ideas in software engineering.

We intend to work with the developers of Eclipse to make the necessary changes to the plug-in and Eclipse to allow them to integrate even better. We will continue to support and promote the plug-in, to build a user base that can benefit from the functionality, provide feedback about their usage, and help prioritize future development. We hope that many developers will find continuous testing to be an effective way to work more quickly and confidently.

References

[1] Beck, K., "Extreme Programming Explained: Embrace Change," Addison-Wesley, 1999.

[2] Beck, K., "Test-Driven Development: By Example," Addison-Wesley, Boston, 2002.

[3] *Eclipse*, http://www.eclipse.org.

[4] Gamma, E. and K. Beck, "Contributing to Eclipse: Principles, Patterns, and Plug-ins," Addison Wesley Longman, 2003.

[5] Henderson, P. and M. Weiser, *Continuous execution: The VisiProg environment*, in: *Proceedings of the 8rd International Conference on Software Engineering*, London, 1985, pp. 68–74.

[6] Rothermel, G., R. H. Untch, C. Chu and M. J. Harrold, *Prioritizing test cases for regression testing*, IEEE Transactions on Software Engineering **27** (2001), pp. 929–948.

[7] Saff, D., "Automated continuous testing to speed software development," Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA (2004).

[8] Saff, D. and M. D. Ernst, *Reducing wasted development time via continuous testing*, in: *Fourteenth International Symposium on Software Reliability Engineering*, Denver, CO, 2003, pp. 281–292.

[9] Wong, W. E., J. R. Horgan, S. London and H. Agrawal, *A study of effective regression testing in practice*, in: *Eighth International Symposium on Software Reliability Engineering*, Albuquerque, NM, 1997, pp. 264–274.