Automated Debugging in Eclipse
(at the touch of not even a button)

Philipp Bouillon
bouillon@st.cs.uni-sb.de

Martin Burger
mburger@st.cs.uni-sb.de

Andreas Zeller
zeller@cs.uni-sb.de

Software Engineering Chair, Saarland University

Abstract

Recent advances in debugging allow for automatic isolation of failure causes such as failure-inducing input or code changes. So far, these advances required a significant infrastructure, notably program analysis, automated testing, or automated construction. The ECLIPSE environment provides such an infrastructure in an integrated, user-friendly fashion. We show how developers and users of automated debugging tools can greatly benefit from such an integrated infrastructure.

1. Introduction

Debugging programs is no longer the tedious, long-lasting, boring work, it used to be. Better and better analysis tools are available that *prevent* failures by detecting errors in the source code. If a failure still occurs, we can *observe* what's going on in a program, using advanced debugging tools. Our ability to handle complexity has grown—unfortunately, so has the complexity of our programs.

In the past few years, a new generation of debugging tools has emerged that relies on *experimentation* rather than analysis or observation. The general idea is to run an automated test under various configurations of circumstances, in order to isolate the failure-inducing circumstances. The so-called *Delta Debugging* approach has been shown to isolate failure-inducing code changes [3] or failure-inducing input [4].

The problem with experimental approaches, though, is that they place a high demand on the infrastructure. For failure-inducing input, one needs an automated test and a means to access the input; for failure-inducing code changes, one additionally needs access to the version archive as well as automated reconstruction. Such services can all be accessed and integrated using, say, configuration scripts, which is fine for case studies—but who wants to set up a configuration script just to debug one failure?

To make automated debugging user-friendly requires an environment which *integrates* all these services—an integrated development environment such as ECLIPSE. In ECLIPSE, versioning is supported via CVS and local history, JUNIT is tightly embedded into ECLIPSE providing automated tests. Furthermore, automatic construction and execution are integral parts of ECLIPSE.

Figure 1 on the following page shows the result of our work—an ECLIPSE workbench with our Delta Debugging plug-ins. The programmer is implementing some JAVA code in ECLIPSE, using JUNIT to unit-test it. Suddenly, some test fails. Our plug-ins are automatically activated by this failing test, and after a moment, report the problem causes:

Failure-Inducing Input. In Figure 1, the *Isolate Failure-Inducing Input* plug-in reports that the test *testSimple* in the class *DDClipseTest* failed. This test opened a file called *sample.xml* which is responsible for the failure—more precisely, the word *Romeo* within this file.

Failure-Inducing Code Changes. In Figure 1, the *Minimize Source Differences* plug-in tells the developer which changes in the source code led to the failure: the failure occurs if and only if the statement `if (a == 3) i--;` has been inserted.

Given such information, the programmer has a rather precise hint to what caused the failure and where the error in the program might be. As the diagnosis is automatically started each time a JUNIT test fails, the programmer will thus gain not only the knowledge *that* something is wrong, but also *why* it is wrong.

The remainder of this paper is organized as follows: Section 2 summarizes how Delta Debugging isolates failure-inducing circumstances such as input or code changes. Section 3 describes our ECLIPSE plug-ins. Our experiences are discussed in Section 4. Section 5 closes with a conclusion and future work.

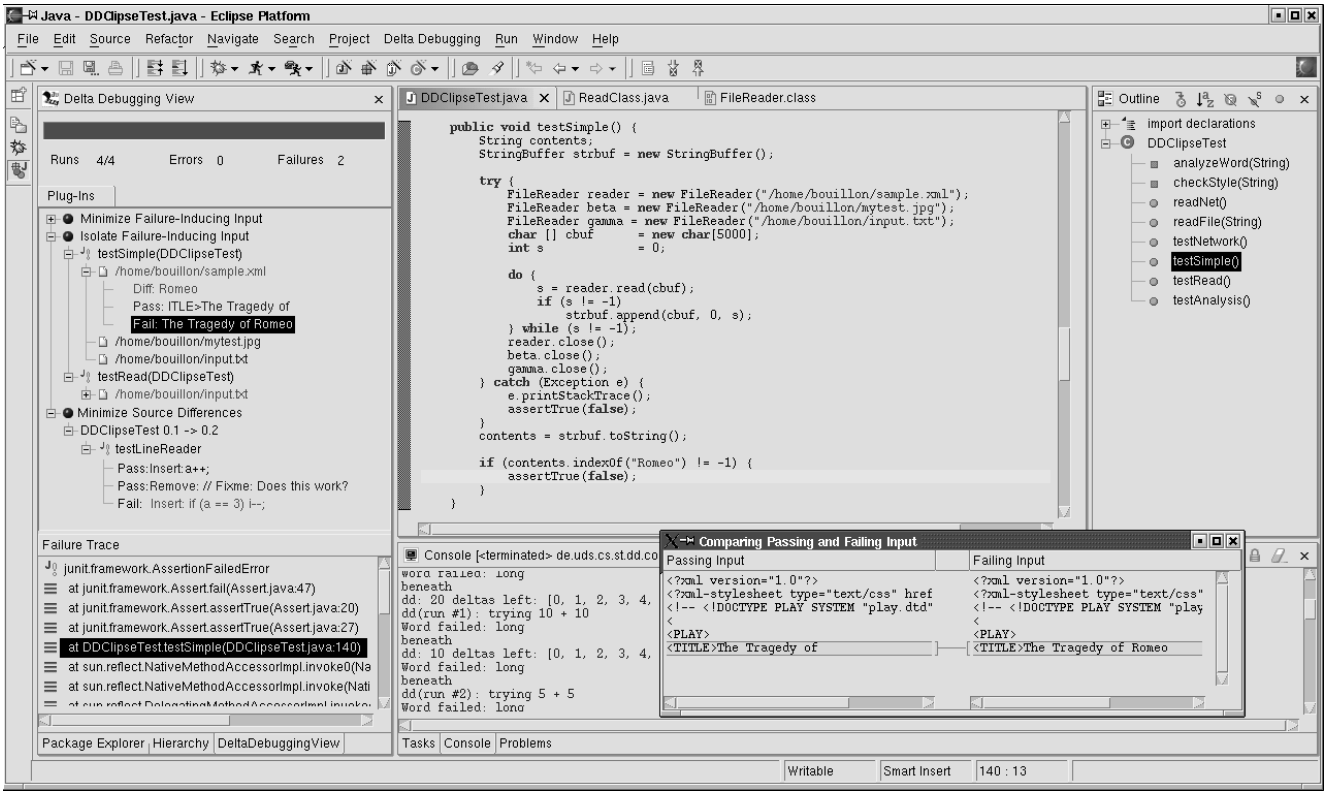

Figure 1. Our tools in action

2. Finding Failure Causes

Delta Debugging is an algorithm to automate code change- and input minimization [4]. To do this, the algorithm needs

Deltas. Delta Debugging takes some large set of differences — called *deltas* — between two program runs and computes the difference that is responsible for the failure. In our cases, a delta is a single change in the program code or a single character of the input of a program. All changes in the code from the last version to the current one or all characters of the input are combined in a set of deltas. Given this set, Delta Debugging finds the minimal subset of deltas that reproduces the failure.

A Test Function. In addition to the deltas, Delta Debugging needs a test function that determines whether the previously observed failure in the program occurs or not. Given a subset of deltas, this test function reruns the program (or even rebuilds the program in the case of minimizing code changes) and tests whether the failure still occurs. Depending on that outcome, Delta Debugging produces new sets of deltas by splitting and reducing the original set.

Delta Debugging depends on its test function telling, whether the test succeeds or not. But what happens, if the test does not produce an outcome (for example, because it could not rebuild the program due to a missing parenthesis)? Whenever all outcomes of the tests for the current set of deltas are *unresolved*, the sets of deltas are reorganized: not two subsets of deltas are produced, but four. Each combination of these deltas is subsequently tested, which eventually produces a passing or failing outcome.

In subsequent steps, Delta Debugging tests progressively smaller sets of deltas. If no smaller set can be tested without reproducing the failure, the algorithm terminates. The reduced set of deltas hopefully guides the programmer to the failure. In figure 2 this computation is demonstrated in an example. The word *Romeo* in a file makes a test fail. First, the lines of this file are analyzed and those lines not responsible for the failure are removed from it. Later, as shown in figure 3, single characters are analyzed. Since the whole minimization process is basically a binary search, the detection of the failure-inducing input is fast.

We have implemented several ECLIPSE plug-ins that implement and extend the Delta Debugging algorithm. In the case of minimizing code changes, we first determine the code differences between the current version, and an earlier, working version. Delta Debugging then systematically narrows down these differences, using JUNIT to test inter-

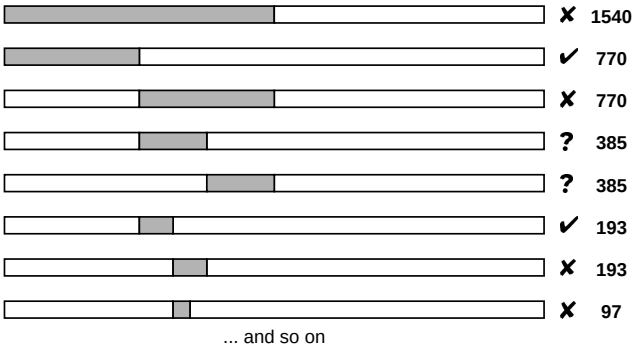

Figure 2. Minimizing input lines

```
51 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✘
52 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
53 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
54 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
55 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✘
56 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
57 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✘
58 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
59 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
60 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
61 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
62 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
63 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
64 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
65 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
66 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
67 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
68 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
69 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
70 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
71 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
72 <TITLE>The␣Tragedy␣of␣Romeo␣and␣Juliet</TITLE> ✔
```

Figure 3. Minimizing input characters

mediate versions, until a minimal difference (= change) is found.

The input minimization plug-in uses Delta Debugging slightly different. In this case, the algorithm determines a minimal difference, too — but this time between the failure-inducing input and an empty input. In short, it simplifies the input until every part left is relevant for producing the failure.

3. Integration with Eclipse

We implemented Delta Debugging as an ECLIPSE plug-in. The *core* plug-in implements the Delta Debugging algorithm itself, while our other plug-ins extend it to provide *code change minimization* and *input minimization*, respectively.

3.1. The Delta Debugging Core

The core implements the Delta Debugging class which provides the Delta Debugging algorithm variants called *ddMin* and *dd* (for differences, see [4]). Additionaly, the core includes several *Delta Creator* classes to create sets of deltas from any initial format. These are used, for example, to transform a file or a string into single characters, each representing a delta.

The Delta Debugging core works closely with four interfaces called *Tester*, *Builder*, *Splitter*, and *Resolver* (cf. figure 4). To extend the Delta Debugging core, it is only neccessary to implement a *Tester*. All other interfaces are optional and are used to specialize the algorithm. There should be no need for more interfaces in the future.

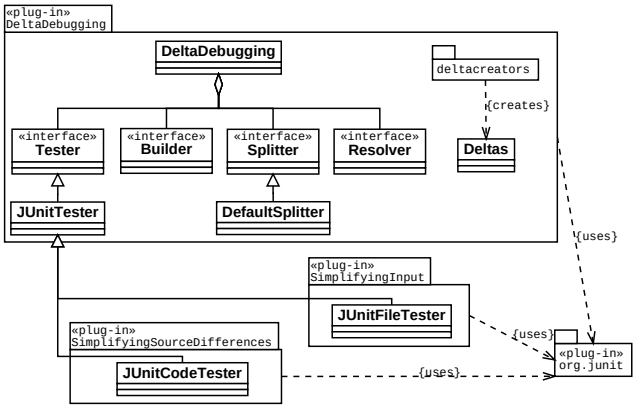

Figure 4. The Plug-In Architecture

Tester. A *Tester* implementation takes the given set of deltas and runs its test on it. The outcome is passed to the Delta Debugging algorithm. We would like to use JUNIT as an easy way for the user to access our analysis tools. Since *Tester* gets a current set of deltas and has to rerun its test based on them, we had to find a means to apply the deltas, before the JUNIT test is started. Therefore, we implemented a wrapper to JUNIT. In the case of the minimizing source differences plug-in, the wrapper (*JUnitCodeTester*) applies the deltas, rebuilds the program, and reruns the JUNIT test. For the input minimization plug-in, we use another wrapper (*JUnitFileTester*) which creates a new input by applying the deltas and reruns the JUNIT test afterwards.

Builder. A *Builder* implementation is used to perform any neccessary pre-formatting of deltas. As an example, consider a simple file format that stores its file length in its first four bytes. Afterwards follows the text. During the minimization process, the length of the file would be changed and so the value stored in the first four bytes has to be modified accordingly. This can be achieved by a builder implementation. Applied to the code change minimization plug-in, the builder is used to rebuild the project.

Splitter. A *Splitter* implementation is used to split any given set of deltas into *n* new sets of deltas, and preparing them to be tested in the next round of the Delta Debugging algorithm. The default implementation simply splits the original set into *n* equally sized sets. More sophisticated implementations could use information on the deltas which they are operating on to produce a *valid* set of deltas (e.g. a syntactically correct JAVA program instead of a JAVA program just split in half).

Resolver. A *Resolver* finally tries to resolve test outcomes that are unresolved. This could happen during the minimization process of a JAVA program for example: for some reason, an opening parenthesis is not closed again (because the closing parenthesis is part of a different set of deltas). So, the resolver tries to regroup the delta sets to produce a resolved test outcome.

Figure 5 illustrates the process and the interaction of those interfaces with the Delta Debugging algorithm. First, the original set of deltas is split into two halves. In the second step, the first half is built and tested, in this case with a passing outcome. So, this half is not responsible for the failure. Afterwards, the second half is built and tested and yields an unresolved outcome, so Delta Debugging resolves it, in this case by prepending a subset of the first half. Building and testing the so constructed set leads to a failing outcome which Delta Debugging splits again and continues until the set cannot be minimized any further.

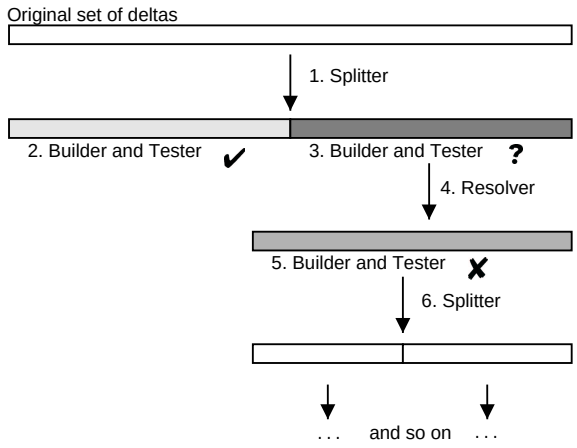

Figure 5. The Delta Debugging Algorithm

Apart from the algorithm, the Delta Debugging core implementation also provides an user interface which can be extended by other plug-ins. The basic user interface consists only of the *Delta Debugging View*. By extending the view, a plug-in will appear as an entry in a tree which can display any calculation results in its child nodes.

3.2. Detecting Failure-Inducing Code Changes

The reduction of a possibly large set of code changes to a set only containing the failure-inducing changes takes place in a plug-in that extends the Delta Debugging core plug-in. The test that has to be performed during the minimization applies a subset of changes to the working version, rebuilds it, and checks whether it still works or not. This is done by a class called *JUnitCodeTester* which extends *JUnitTester*. The original working version of the project source is not changed here but a copy of it is modified.

In order to retrieve the last version of the source that successfully completed its test, the plug-in uses the CVS functionality of ECLIPSE. Once retrieved, the code-changes between the current, failing version and the working version are computed and afterwards minimized to yield the failure-inducing change.

3.3. Detecting Failure-Inducing Input

To detect failure-inducing input, we have written another plug-in that extends the Delta Debugging core plug-in. The *JUnitFileTester* class extends *JUnitTester* which is an implementation of the *Tester* interface. The *JUnitFileTester* class creates a file from a given set of deltas (characters in this case). The created file is then read by the JUNIT test and results are passed to the Delta Debugging algorithm.

The plug-in has to determine (automatically) if a JUNIT test opens a file or not. This is done by debugging the first JUNIT run, suspending execution on all file-access methods (or constructors). If such a breakpoint is reached during the test execution, the name of the opened file is determined and — provided that the original test fails — Delta Debugging is applied to that file.

4. Experiences

The steep learning curve for plug-in developers in ECLIPSE is hard to master. There is plenty of documentation on how to write a very simple plug-in, but the step from the easy example to real world solutions is not covered by much documentation. The developer only has the API documentation to go on from this point.

Furthermore, the documentation does not provide an overview of the interaction between the classes. The classes and their methods are described in great detail, but how several classes can be combined to solve a specific task is not shown. Especially, it is often surprising for the developers that they have to cast an object into another one in order to achieve a certain behaviour. Often, if the programmer wants to accomplish something, she has to search the ECLIPSE source code for that functionality, copy it and customize it to suite her needs.

Once the learning curve is mastered, ECLIPSE turns into a very powerful platform. The existing CVS, JUNIT, and debugger interfaces allow us to easily add Delta Debugging to ECLIPSE and thereby providing automated debugging for JAVA programs. Compared to the complexity of implementing our own integrations of these tools, ECLIPSE provides an easy access to everything we need.

Because of its architecture ([1]), ECLIPSE allows the efficient development of new plug-ins which in turn are extensible, thereby allowing the world-wide sharing of ideas and plug-ins. The architecture allows all of the plug-ins to communicate and interact with each other via cleanly defined interfaces which is a great benefit for the user, the developer, *and* ECLIPSE itself, because its value is steadily increased by the encouraged software engineers.

Thanks to ECLIPSE, researchers finally benefit from an environment which allows them to directly transport current results from universities to the industry for their mutual profit.

5. Conclusion and Future Work

Without ECLIPSE, our automated debugging tools would never have found the way to the end user. ECLIPSE advances the integration and interaction of programming tools in a way that can only be compared to the advent of the UNIX system, 30 years ago. This integration and interaction also opens new, exciting research perspectives:

Exploiting syntactic information. Isolating failure-inducing code changes can greatly profit from syntactic knowledge. All changes belonging to one class or one method can be combined, thereby reducing the amount of unresolved tests that occur during the minimization process.

Integration with testing tools. Besides interacting with standard services, there are also opportunities for integrating multiple plug-ins. For instance, continuous testing [2] tells the programmer *while she is typing* which tests pass and fail. In combination with our Delta Debugging plug-ins, the programmer will see at once *why* the test fails.

Learning from test results. Isolating failure-inducing code changes requires some successful test of an earlier version. To achieve this, a *database* that stores all results of all JUNIT test runs (including the configuration that led to the result) would be highly useful. Eventually, one could devise tools that learn from earlier test results to predict possible failure causes.

Cause-effect chains. Delta Debugging can also be applied on *program states* to isolate *failure-inducing variables:* "Initially, variable $v1$ was $x1$, thus variable $v2$ became $x2$, thus variable $v3$ became $x3$... And thus the program failed." This explains failure causes automatically and effectively. We are working on implementing this approach for JAVA programs within ECLIPSE.

All in all, ECLIPSE promises to become a standard platform for development and analysis of JAVA programs. With more and more researchers porting their tools to ECLIPSE, users can immediately benefit from the researchers' work. We are happy to contribute our own research results to the ECLIPSE community—and hope that, in the future, programmers will not even have to press a button to find out why their programs failed.

Acknowledgments. We would like to thank Erich Gamma and Darin Wright for their support via the developer mailing list. Christian Lindig and Thomas Zimmermann for proofreading and Holger Cleve and Stephan Neuhaus for their constant support.

6. About the Authors

Philipp Bouillon and **Martin Burger** are students of computer science at the software engineering chair of Saarland University in Germany. Philipp has ported Delta Debugging to ECLIPSE and implemented the plug-in that automatically minimizes or isolates failure-inducing input, and Martin has implemented the plug-in that automatically isolates failure-inducing code changes.

Andreas Zeller is a software engineering professor at Saarland University in Germany. His research interests include dynamic program analysis, configuration management, and program visualization. Zeller received a PhD in computer science from the Technical University of Braunschweig, Germany.

References

[1] OTI Labs. Eclipse platform technical overview. White Paper, 2003. Available online at www.eclipse.org.

[2] D. Staff and M. D. Ernst. Can continuous testing speed software development? In *14th International Symposium on Software Reliability Engineering*, Denver, Colorado, Nov. 2003.

[3] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, volume LNCS 1687. Springer Verlag, 1999.

[4] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.