# Automated Debugging in Eclipse

Martin Burger · Karsten Lehmann · Andreas Zeller
Department of Computer Science
Saarland University
Saarbrücken, Germany
{mburger, lehmann, zeller}@st.cs.uni-sb.de

## ABSTRACT

Your program fails. What is the cause of this failure? In this demo, we present two delta debugging plug-ins for the Eclipse environment which isolate failure causes in the program history and in the program's execution.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, diagnostics, testing tools, tracing*; D.2.6 [**Programming Environments**]: Integrated environments; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control

## General Terms

Experimentation, Reliability

## Keywords

Programming environments, testing, debugging, version control, program comprehension.

## 1. INTRODUCTION

Your program fails. What is the cause of this failure? In this demo, we present two delta debugging plug-ins for the Eclipse environment which isolate failure causes in the program history and in the program's execution:

**DDchange: failure-inducing changes.** The *DDchange* plugin is useful if an old (working) version of the unit is available. By systematic tests, it narrows down the set of code changes until it has isolated the failure-inducing change: "'The failure was caused by a change to output.cpp at line 197 on June 30th"'.

**DDstate: failure-inducing program states.** The *DDstate* plugin requires an passing test run of the unit. By narrowing down the differences between program states, it isolates the variables and statements that cause the failure: "At input.java in line 307, this.buffer became NULL, and therefore, the program crashed."

Both plug-ins are fully automatic; all that is needed is an automated test, such as JUnit. As soon as a JUnit test fails (and a passing version pr run is available), the appropriate plugin kicks in and produces a diagnosis within minutes.

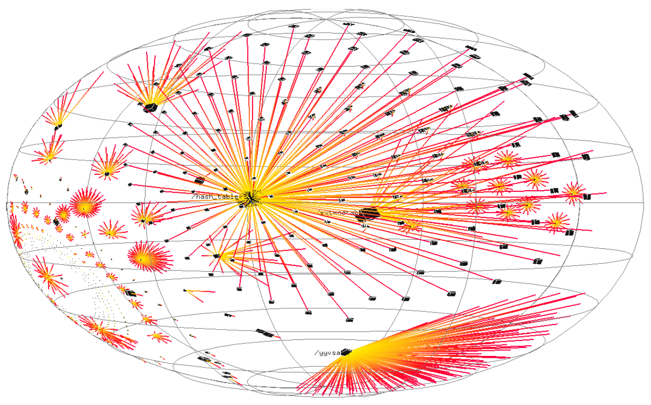


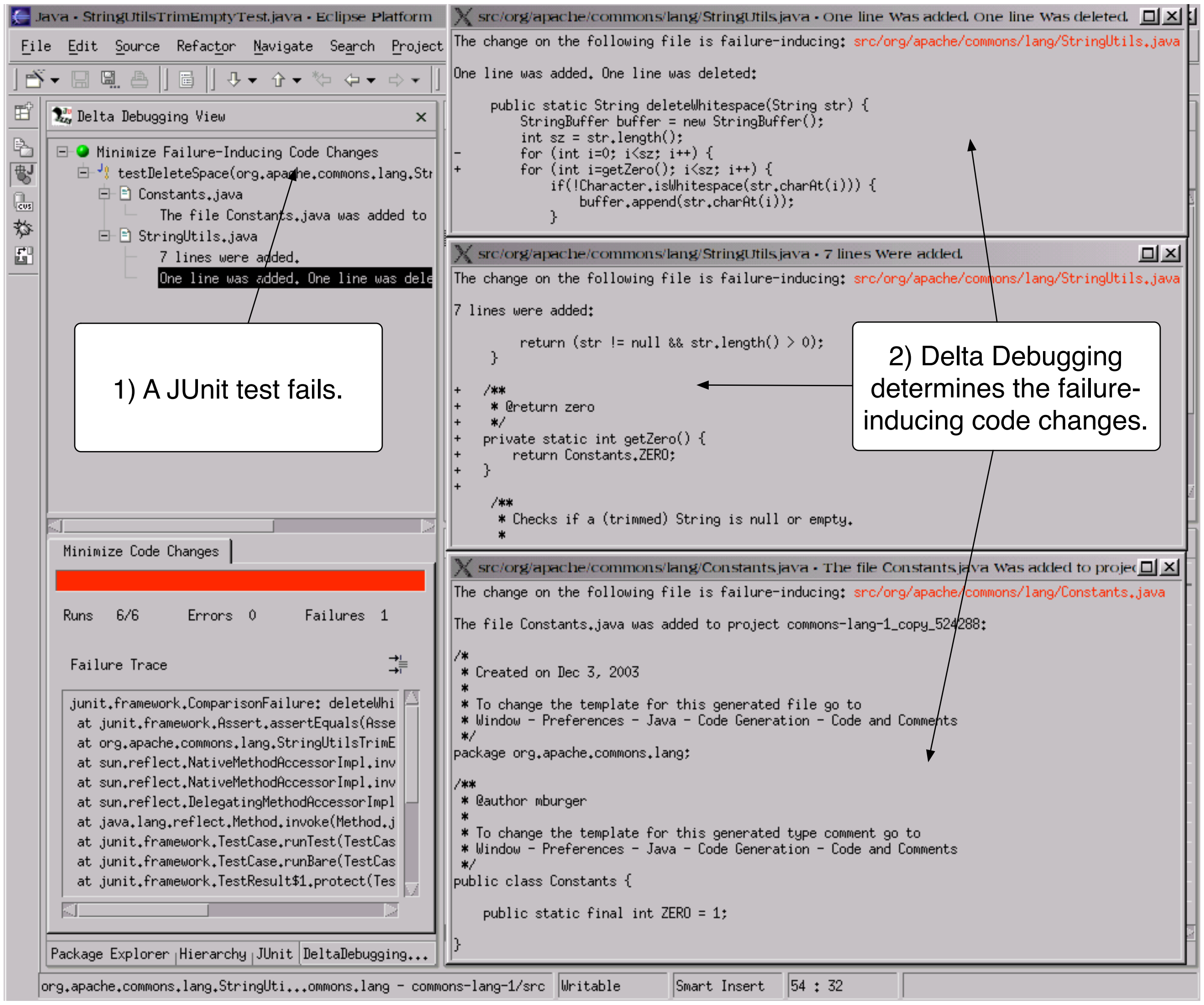
**Figure 1: A memory graph**

## 2. HOW DOES IT WORK?

The plugins do not use program analysis to isolate failure causes; in fact, they do not analyze the source or machine code at all. Instead, they rely on two principles:

**Causes as Differences.** A cause is always a difference between a world where the effect occurs and a world where the effect does not occur. Hence, one can narrow down causes from the differences between these two worlds. In our case, these two worlds are the two differing program versions (DDchange) or program runs (DDstate). For versions, We use standard differencing techniques to extract and compare them. For runs, we extract the program state as *memory graphs* (Figure 1), encompassing values and structures in a single representation which can be compared using standard graph comparison techniques.

**Narrowing through Experimentation.** The difference between actual program versions or program states may still be too large to be useful. Therefore, delta debugging narrows down that difference systematically—by creating an *intermediate* version which takes half of the differences into account. In practice, this means either a version where half of the code changes are applied (DDchange), or a "mixed" state consisting of both "passing" and "failing" values (DDstate). The plugins construct these intermediate versions and *experiment* how the program then behaves. Every time the program still fails, or still passes, the difference is reduced by 50%; eventually, only a minimal failure-inducing difference remains—the change which made the program fail (DDchange) or the variable(s) whose value(s) causes the failure to occur (DDstate).


1) A JUnit test fails.
2) Delta Debugging determines the failure-inducing code changes.

```
Java - StringUtilsTrimEmptyTest.java - Eclipse Platform
File  Edit  Source  Refactor  Navigate  Search  Project

Delta Debugging View
Minimize Failure-Inducing Code Changes
  testDeleteSpace(org.apache.commons.lang.Str
    Constants.java
      The file Constants.java was added to
    StringUtils.java
      7 lines were added.
      One line was added. One line was dele

Minimize Code Changes
Runs   6/6       Errors   0       Failures   1

Failure Trace
junit.framework.ComparisonFailure: deleteWhi
 at junit.framework.Assert.assertEquals(Asse
 at org.apache.commons.lang.StringUtilsTrimE
 at sun.reflect.NativeMethodAccessorImpl.inv
 at sun.reflect.NativeMethodAccessorImpl.inv
 at sun.reflect.DelegatingMethodAccessorImpl
 at java.lang.reflect.Method.invoke(Method.j
 at junit.framework.TestCase.runTest(TestCas
 at junit.framework.TestCase.runBare(TestCas
 at junit.framework.TestResult$1.protect(Tes

Package Explorer | Hierarchy | JUnit | DeltaDebugging...
org.apache.commons.lang.StringUti...ommons.lang - commons-lang-1/src   Writable   Smart Insert   54 : 32
```

```
src/org/apache/commons/lang/StringUtils.java - One line was added. One line was deleted.
The change on the following file is failure-inducing: src/org/apache/commons/lang/StringUtils.java

One line was added. One line was deleted:

    public static String deleteWhitespace(String str) {
        StringBuffer buffer = new StringBuffer();
        int sz = str.length();
-       for (int i=0; i<sz; i++) {
+       for (int i=getZero(); i<sz; i++) {
            if(!Character.isWhitespace(str.charAt(i))) {
                buffer.append(str.charAt(i));
            }
```

```
src/org/apache/commons/lang/StringUtils.java - 7 lines were added.
The change on the following file is failure-inducing: src/org/apache/commons/lang/StringUtils.java

7 lines were added:

        return (str != null && str.length() > 0);
    }

+   /**
+    * @return zero
+    */
+   private static int getZero() {
+       return Constants.ZERO;
+   }
+
    /**
     * Checks if a (trimmed) String is null or empty.
     *
```

```
src/org/apache/commons/lang/Constants.java - The file Constants.java was added to projec
The change on the following file is failure-inducing: src/org/apache/commons/lang/Constants.java

The file Constants.java was added to project commons-lang-1_copy_524288:

/*
 * Created on Dec 3, 2003
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package org.apache.commons.lang;

/**
 * @author mburger
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
public class Constants {

    public static final int ZERO = 1;

}
```

**Figure 2: The DDchange plugin**

Delta Debugging is the first program comprehension approach to exploit these principles; full details of the approach are available [1].

## 3. ECLIPSE PLUGINS

Figure 2 shows an early version of the *DDchange* plugin in action. As soon as a test fails, *DDchange* retrieves an older (passing) version from the version archive and isolates the failure-inducing difference; all this takes place automatically. *DDstate* does a similar job, but displays the failure-inducing variables and statements in the failing run instead. Note that a cause, as isolated by these plug-ins, need not be an error—but a cause can provide an excellent starting point when it comes to fix the program.

We expect that automated diagnosis techniques will have many further benefits in program comprehension and we welcome demo participants to bring in their own ideas and join in the discussion.

## 4. DOWNLOAD

The delta debugging plugins are available for download from:

`http://www.st.cs.uni-sb.de/eclipse/`

## 5. ACKNOWLEDGMENTS

Delta Debugging research is funded by Deutsche Forschungsgemeinschaft, grant Sn 11/8-1. The development of the ECLIPSE plug-ins was made possible by an IBM ECLIPSE innovation grant. We thank Ken Bauer for organizing the OOPSLA 2005 demonstrations.

## 6. REFERENCES

[1] A. Zeller. Why Programs Fail: A Systematic Guide to Debugging. Morgan Kaufmann, October 2005.