Software Engineering Chair

Saarland University

Locating Failure-Inducing Code Changes
in an Industrial Environment

Diploma Thesis

Martin Burger

mburger@st.cs.uni-saarland.de

Advisor: Prof. Dr. Andreas Zeller
Referees: Prof. Dr. Andreas Zeller and Prof. Dr. Reinhard Wilhelm

Saarland University, Department of Informatics,
Software Engineering Chair, Prof. Dr. Andreas Zeller

December 14, 2005

Contents

Preface

Abstract

A regression is a common type of failure that occurs while changing source code of a working program: "*Yesterday, your program worked. Today, it does not. Why?*" Manual debugging of regressions is costly in terms of labour and time, it annoys developers and costs organisations lot of money.

This thesis presents a platform that integrates seamlessly *automated determination of failure-inducing code changes* with common development processes. Automated determination may increase efficiency, improve productivity, while saving time and money. Because established processes need not to be adapted, start-up costs and risks are low.

The platform supplies two tools in form of plug-ins. A Eclipse plug-in integrates automated determination of failure-inducing changes with a prominent IDE. As soon as a unit test fails, it can be debugged automatically. A Maven plug-in enriches continuous building and testing with automated debugging. Instead of a simple failure report, we obtain valuable information about failure-inducing changes—without lifting a finger: "The failure cause is the change in line 37 of file StringUtils.java."

The plug-ins are instances of a framework that contains all basic functionality in order to build tools that enable automatic determination of failure-inducing changes. Using that framework, implementing new tools that debug other types of failure is quite easy and simple—again saving time and money.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

Members of the Software Engineering Chair at the Saarland University: Prof. Andreas Zeller (supervisor and provided the opportunity to write this thesis), Holger Cleve (lent helpful books to me), Dr. Christian Lindig (LaTeX guru), Stephan Neuhaus (we have no fun at work), and Tom Zimmermann (revised parts of this work).

Staff members of WEB.DE's department of New Technology: Jörg Beyer (head of New Technology and responsible for the initiation of the cooperation), Wolfgang Miller-Reichling (my advisor at WEB.DE), Gisbert Amm (Maven and CruiseControl guru), and the automatic coffee maker that serves hot drinks for free.

Philipp Bouillon (fellow student, author of [Bou04] and support hotline), Karsten Lehmann (fellow student, presenter at OOPSLA'05, friend and jointly responsible for some all-night parties), Lorenz Boerger (native speaker and friend), and Christian Tiator (a friend in need is a friend indeed).

I am indebted to the members of the Corps Franconia Karlsruhe im WSC for giving me accommodation facilities and having interesting discussions as well as fun with them at their bar.

Last, but not least—for beeing here: Tatjana Weiß and Therese Burger. Thanks!

Testimony

Herewith I declare that I have written this thesis myself and have not used sources and aids other than those listed.

Saarbrücken, December 14, 2005

. . . .
Martin Burger

Copyright

1 Introduction

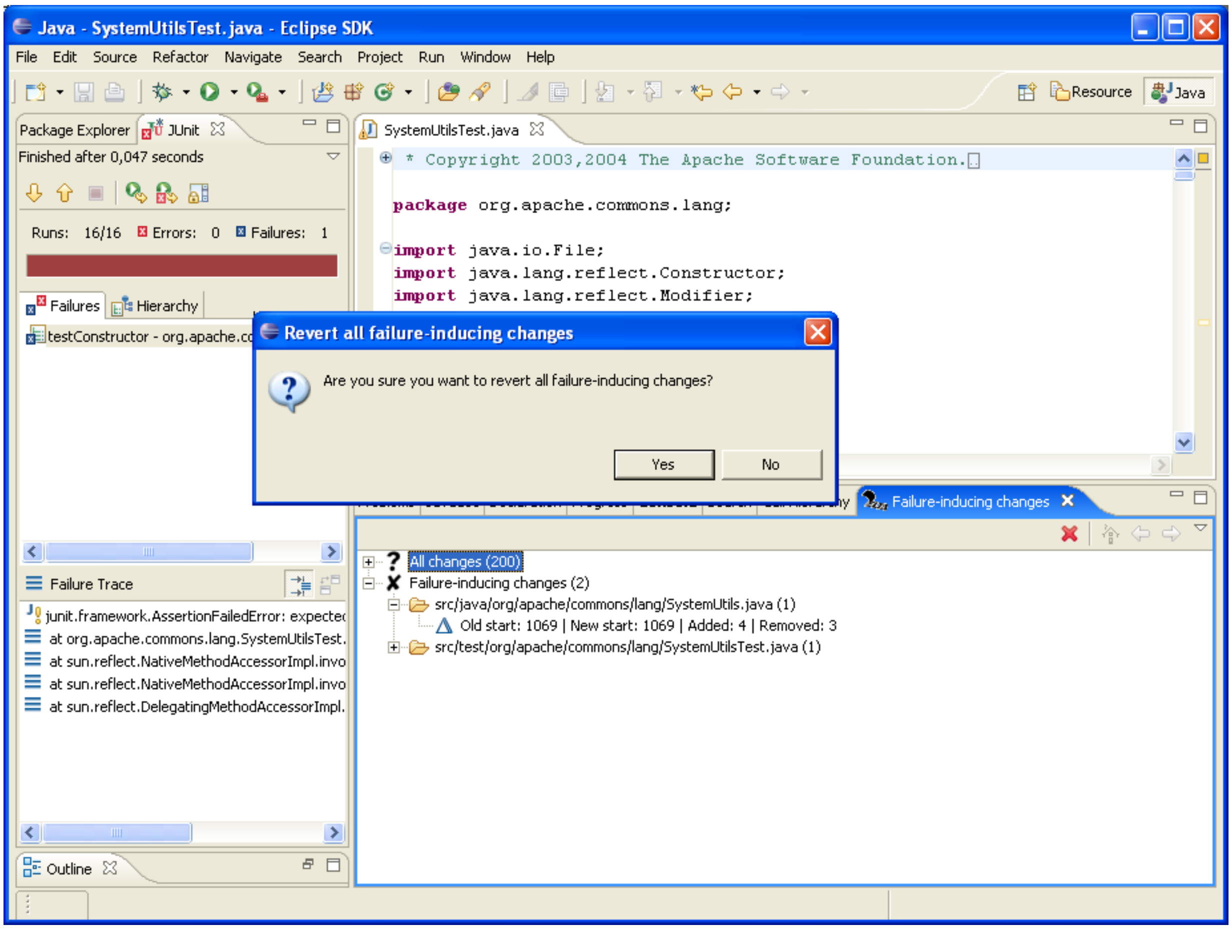
1.1 The Problem

The WEB.DE GmbH—operating company of WEB.DE, Germany's second largest Internet portal—utilise agile software development in order to implement new features and products for their portal. These agile methods and processes include Extreme Programming and *Scrum*, an iterative, incremental process for developing products. It produces a potentially shippable set of functionality at the end of every iteration (in the context of Scrum called *Sprint*). A so-called *Sprint Backlog* defines all tasks required to complete an iteration. Working through that backlog, developers implement all listed features and tests these applying various techniques, such as unit and continuous testing. A Sprint has not finished until all features are implemented *and* pass the tests, including tests implemented during previous iterations—tests are a substantial part of the software development process operated by WEB.DE[1] development teams.

If a developer introduced a defect into a product that previously passed all tests, she made a regression. As a software product is developed, this type of defect is quite common. Even if you are not familiar with agile methods and processes, you will know this scenario: "*Yesterday, your program worked. Today, it does not.*". Your key question will be: "*Why?*" Normally, only a few changes on the program code are responsible for today's failure—out of many changes. At WEB.DE, as well as other larger development teams, many programmers and others have a hand in development of a software product, making many changes everyday. If a regression occurs, finding the defect in the code will be a vehement challenge.

Determining *failure-inducing changes*, the changes that introduced the defect in your code and therefore caused the failure, is tedious and time-consuming into the bargain. This debugging task is typically accomplished manually and ties up resources that are busy with other things. Therefore, manual debugging is not only tedious from the individual programmer's point of view, but also costly in terms of labour and in terms of time. In short: *debugging regressions costs WEB.DE money*.

[1]In this work, the term WEB.DE is used instead of WEB.DE GmbH unless otherwise noted.


Java - SystemUtilsTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Resource
Java
Package Explorer
JUnit
SystemUtilsTest.java
Failures
Hierarchy
Failure Trace
Outline
Revert all failure-inducing changes
Yes
No
Failure-inducing changes

```
* Copyright 2003,2004 The Apache Software Foundation.

package org.apache.commons.lang;

import java.io.File;
import java.lang.reflect.Constructor;
import java.lang.reflect.Modifier;
```

Finished after 0,047 seconds

Runs: 16/16 Errors: 0 Failures: 1

testConstructor - org.apache.co

```
junit.framework.AssertionFailedError: expecte
at org.apache.commons.lang.SystemUtilsTest.
at sun.reflect.NativeMethodAccessorImpl.invo
at sun.reflect.NativeMethodAccessorImpl.invo
at sun.reflect.DelegatingMethodAccessorImpl.
```

Are you sure you want to revert all failure-inducing changes?

```
All changes (200)
Failure-inducing changes (2)
  src/java/org/apache/commons/lang/SystemUtils.java (1)
    Old start: 1069 | New start: 1069 | Added: 4 | Removed: 3
  src/test/org/apache/commons/lang/SystemUtilsTest.java (1)
```

Figure 1.1: As soon as a test failed, you can revert the failure-inducing changes. There were 200 individual changes, only two of them are relevant to the failure.

1.2 The Proposed Solution

The purpose of this thesis is to present a solution that reduces the time spent identifying the changes that introduced a failure. The main objective of the study is to provide *tools* that enrich the familiar development process with automated determination of failure-inducing changes, not even at the push of a button—*completely automatically*. As soon as there is a failing test, a tool should identify the changes as illustrated in Figure 1.1. Although there were 200 changes since the last passing run of the test, only two of them are relevant to the failure. Thus, the developer has to examine only two instead of 200 changes, saving a lot of time.

Preferably, the tools are integrated seamlessly into the current development process—without bothering the developer with additional tasks or other prerequisites. For instance, existing failure reports should be extended with information about failure-inducing changes.

The determination of failure-inducing changes is based on several existing theoretical concepts.

These underlying terms and concepts are discussed in an academic way, we review topics that include approaches to simplify problems and to isolate failure causes. Having the theoretic foundation, we are able to deduce straightforward a strategy that determines the failure-inducing changes.

We can integrate the automated determination of failure-inducing changes with development processes at several stages. For instance, a failure can occur both at a developers workstation when running individual unit tests and at the central server that runs continuous integration and testing. If we would implement different tools without abstracting common functionality, we had to duplicate similar code.

Therefore, we implement the deduced strategy and related functionality in form of a *very general framework*. That framework should supply the complete functionality in order to accomplish the determination of the failure-inducing changes. Furthermore, that framework must be easy to use and to extend, because we want to ensure the cost-benefit over designing tools from scratch.

Finally, we implement concrete instances of the developed framework in order to integrate the determination of the failure-inducing changes with WEB.DE's current process. These instances enrich the development process at two different points. First, a plug-in for the Eclipse IDE enables the individual developer to determine the failure-inducing changes on hers local workstation. Second, a Maven plug-in enables the quality assurance department to incorporate the determination into the automatic build process.

The framework as well as the tools—the *platform*—are released under an open-source license. Thus, everybody can use all results of this thesis.

1.3 Structure

The content of this thesis evolves along a continuum from the underlying theory to concrete tools that are in use at WEB.DE. Figure 1.2 illustrates the continuum as described in detail below.

Chapter 2 of this thesis discusses the fundamental terms and concepts regarding debugging in general, simplifying a problem, causes and effects, and isolating failure causes. In Chapter 3, we derive failure-inducing code changes and their automatic determination from the acquired theoretic equipment. This chapter contains a strategy to determine these changes in theory. Chapter 4 introduces the framework called DDCHANGE, the practical core of the whole platform. That framework supplies all required parts in order to accomplish the strategy—it enables you to develop tools that determine failure-inducing changes. The concrete tools in the form of two plug-ins for well-known Java development tools are presented in Chapter 5. The plug-ins are instances of the framework, they enable developers to determine the wanted changes in practice using their familiar tool. Conclusions are drawn in Chapter 6, it summarises aspects of the previous chapters and suggests topics for further studies. Finally, the Appendix A lists the content of the included CD. That CD contains the complete source code of the platform, the JavaDoc

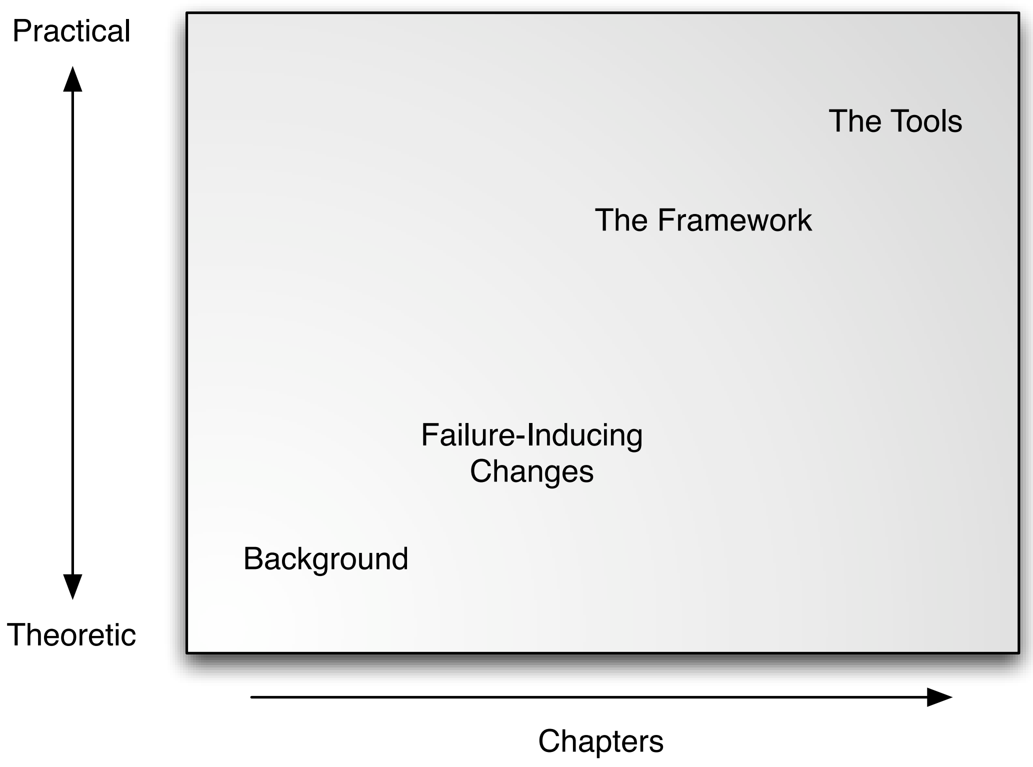


Figure 1.2: The content of this work in a continuum from theory to practice. The platform draws upon the theory of possible worlds.

API documentation, movies that show the tools in action, and the framework and tools ready for use.

1.4 Results of the Thesis

This thesis proved that the automated determination of failure-inducing changes can be integrated seamlessly with the established development process at WEB.DE. The automation may save a developer a significant amount of time, because failure-inducing changes are identified much faster than they are with manual debugging techniques.

Both tools help the developers at WEB.DE with their day-to-day business. First, the Eclipse plug-in enables developers to debug failures faster and with less effort using her familiar IDE. Second, the Maven plug-in can be integrated seamlessly with the continuous build process; thus, it enriches automated testing with automated debugging. Developers are able to concentrate on other tasks then the tedious and costly debugging of failures introduced by changes.

In addition to the simple usage of the tools, you can extend the framework and develop new instances, respectively. As seen in this thesis, you need verry little effort to implement such tools, because the framework, which is fully reusable and extendable, supplies the core functionality. The framework and the tools are released as open-source, allowing everybody to benefit from the results. For instance, you could create a tool that debugs failed acceptance tests, on the basis of the framework.

1.5 Future Work

Because of time limitations and limited resources, the following issues occurred during this work and are potential subjects for further study and development:

- *Evaluation.* We cannot prove that DDCHANGE and its two instances save a developer a significant amount of time. An evaluation could fill this gap. The tools are ready for use and provide basic statistics. The evaluation could research also the different types of failures and defects that can be debugged with the platform.
- *Optimisations.* There is room for additional optimisations. Currently, we disregard the chronological order of the changes, and we do not take the dependencies of different changes into account. Thus, we could improve the runtime of the determination by regarding history and grouping of the changes. Moreover, related work may be used in order to improve the runtime of the debugging process. That includes "Continuous Testing" [SE04b] and "Change Classification" [SRRT05].
- *Reproducing* the *failure.* The current implementation interprets all failures as a failing outcome. For instance, if the location of the failure has another backtrace, that failure should be interpreted as an unresolved outcome.
- *Improvements of the user interfaces.* Both the report of the Maven plug-in (a special form of a user interface) and the user interface of the Eclipse plug-in can be improved. For instance, we could extend the report with information about the originator and the time of the change.

2 Background and Basics

What is a bug? How to find a bug? How can we automatise tests—thus making a program fail automatically? What are the relevant circumstances for a problem? Can we find the relevant ones automatically? What is a failure cause? How can we isolate failure causes—perhaps even automatically? This chapter answers these important questions in a nutshell. The answers to these questions form the background of this work, they cover the theory, concepts, principles and algorithms that lead straightforward to the fundamentals of the platform for automated determination of failure-inducing changes.

The following content follows the structure of [Zel05] and summarizes the most important aspects for this work. A complete explanation of these topics goes beyond the scope of this work. For more details, examples and a comprehensive discussion of (systematic) debugging see that book.

2.1 Essential Concepts

This section defines some essential terms and gives an introduction of the process called *debugging*.

2.1.1 A Defect Causes a Failure

When talking about bugs, different terms are used: error, defect, flaw, mistake, failure, or fault. To have a differentiation let's see how a defect causes a failure.

At first, a programmer creates a *defect* in the program code (also called bug). If the program is executed, the incorrect code may cause an *infection*: the actual program state differs from the intended one. Finally, the infection in the program state causes a *failure* if the effect is an externally observable error.

For instance, a programmer writes a condition-controlled loop, where the loop condition is never changed within the loop (the defect). If the program is executed, the incorrect condition causes an infinite loop but the programmer's intention was to repeat the loop only a few times (the infection). A user that uses the program may observe that the user interface does not respond anymore because of the never-ending loop (the failure).

To summarize (cf. [Zel05]):

- *Defect*: An incorrect program code.
- *Infection*: An incorrect program state.
- *Failure*: An observable incorrect program behaviour.

In this work the term bug is avoided, instead the terms as defined above are used.

2.1.2 Debugging in Seven Steps

If we find a failure, we want to identify the defect in the program code in order to remove the defect. After removing the defect, the failure does not longer occur. That process is called *debugging*.

Zeller splits this process in seven steps [Zel05]:

1. *Track the problem.* Enter the issue in a ticket tracking system that manages software problems. This ensures that the defect will not be lost. Most tracking systems define a workflow that automates the lifecycle of the issue.
2. *Reproduce the failure.* Create a test case that reproduces the failure. This helps to verify the fix for the defect and to find regressions[1].
3. *Automate and simplify.* Concentrate on the relevant circumstances by leaving out the irrelevant. Ideally, use a method that automates this process. One such method is the delta debugging algorithm.
4. *Find infection origins.* Every failure can be traced back to the defect via its infection. Find possible infection origins by going back this trail.
5. *Focus on likely origins.* If you find more than one possible origin, focus on the most likely (for example so-called anomalies or code smells).
6. *Isolate the infection chain.* By isolating transitively the origins, create an infection chain from the defect to the failure.
7. *Correct the defect.* Remove the defect from the code and verify the result for example by running the created test case. Now, that test case must not reproduce the failure anymore.

[1] Generally spoken a regression is a relapse to an earlier development stage or stage of maturation, a step backwards. In software engineering, a regression is the *degeneration* of software under development: A failure occurs after a change in a part of the software that was working before that. A regression is often a side effect that was not thought of by the programmer who changes another part of the software. Regression testing is an important part of the extreme programming methodology. That approach again uses unit tests to define the functionality of certain parts of the code

The subject of this work is related to the steps *reproduce*, *automate and simplify*, and *focus on and isolate origins*.

Having the ability to reproduce the failure, automate the test case, focus on causes in the code and the method to isolate the origin of the infection, we can focus on implementing the platform.

Locating the defect is the most time consuming activity in the debugging process. Providing a good hint for the location of the defect, the whole debugging process could be completed faster. In most cases, the costs of correcting the defect are negligible compared to the preceding six steps (see above).

2.1.3 Automated Debugging

Instead of accomplishing the whole debugging process manually (a boring and tedious task), the computer can assist the programmer by debugging a program nearly automatically.

Some techniques that follow that path, one of these techniques is the most interesting relating to this work: the *delta debugging* algorithm narrows down the difference between a passing and a failing run. Before we introduce that algorithm, we will see how tests help on debugging.

For a more detailed view on defects, infections, failures and debugging see Chapter 1 in [Zel05].

2.2 Automated Tests

Creating a test case is the first step before a program can be debugged: A test case executes a program with the intention to make it fail (see [Zel05]). This work concentrates on testing the unit layer, however, other layers may be tested using the framework as well.

2.2.1 Testing

Reports by the user are one possibility to call attention to problems. But, the earlier a defect is found the cheaper it is to fix the problem—compare a defect that is found and fixed during the implementation with a defect that emerges after the deployment of the program. So, one reason for testing is to find defects as early as possible.

In the *test driven software development* model tests that might break the code are written by the programmer first. Initially, these tests fail. After the tests are written, the programmers implement the code that has to pass more and more of the existing tests. The development has finished after no test fails anymore. The test suites (a collection of test cases) will be updated during the whole process, for example if the requirements change or if the programmer discovers new possible problems. See [Bec02] for more information about the *test driven development* process.

This view on testing aims to verify the correctness of the code, more precise testing compares the actual with the expected behaviour. Testing can never be a complete evidence of the correctness of software in the same way that an experiment cannot prove a theory. Beside this classical intention of testing it is often required during the debugging process: A test can reproduce a problem, rerunning a test multiple times helps to simplify a problem and to observe the run. Running a test after the fix verifies its success and running it before a new release is deployed ensures that the problem does not occur a second time.

Because a test is required often, it should be automated. Furthermore, automation is a prerequisite of several *debugging techniques*, particularly with regard to the delta debugging algorithm. Automated tests are used to isolate failure causes automatically, including failure-inducing code changes.

2.2.2 Unit Tests

A program can be separated into different layers: the presentation (covering user interaction), the functionality (independent of a specific presentation) and the unit layer. Each layer can be tested with different techniques. This work concentrates on the unit layer. One motivation for that focus is that WEB.DE uses the extreme programming approach and the agile software development framework. Extreme programming adheres to the test driven development model and as mentioned earlier tests are essential for that model. Besides that motivation unit tests are well known and there are many frameworks for specifying and running unit tests, including for the Java programming language[2]. Most modern IDEs and other tools related to the software development process support unit tests innately.

Units are modules of source code, for example classes, packages, functions and all other items that form a unit according to a selected design or language. Each individual test executes a specific unit, the unit that is covered by that test. Having these tests, the execution of the covered units can be automated. Because early testing is important in extreme programming, unit tests are a good candidate for that method. Units are implemented before the whole functionality and the presentation layer is available.

A tool that runs unit tests provides a test framework. The framework collects unit tests in test suites and runs all of them or selected ones without any user interaction. Thus, the unit tests are run automatically.

Besides the automation of executing parts of the program, unit testing allows the programmer to change code often and with lower risk. For example, if the programmer refactors the structure of the program she can be sure that the changed unit still works as expected (provided that all

[2]Opensourcetesting.org lists 115 unit testing tools for more than ten programming languages—2005-11-05.

aspects of the desired behaviour are covered by the test). The programmer just runs the unit test after the change. Unit tests provide a kind of documentation of the units. Looking at the unit test another programmer can learn how to use the API of a particular class, especially in what order the methods have to be called. But unit tests are afflicted with some limitations. They do not cover all possible origins of failures. If the programmer of the unit test and the tested unit is the same person, she tends to test the *known* aspects, thus unexpected origins are not tested. Because unit tests only execute the unit under test by definition, they do not catch other problems like integration errors or performance problems and any other issues that affect the system-wide behaviour of the program.

In the context of this work, unit tests are very useful:

- They can easily be automated, a prerequisite for automated debugging.
- If you are confronted with a problem in a unit, the unit test that covers that unit will be useful to reproduce the problem.
- There are testing frameworks available that run unit tests.
- Existing tools provide functionality to run unit tests innately.

Chapter 3 of [Zel05] provides a detailed description of basic testing techniques, including testing the presentation and the functionality layer. One section of that chapter describes how to isolate units.

2.3 How to Simplify a Problem

After reproducing a problem, for example with help of a unit test, the next step in the debugging process is to simplify the problem: Check every circumstance of the problem and decide whether it is relevant or irrelevant for the problem to occur. If it is not relevant, disregard that circumstance. Simplifying results in a set of circumstances that are all relevant to the problem. Circumstance is an extensive term; it includes all aspects that may affect the problem. Because there are many aspects that may influence a problem, simplifying is an important step in debugging. To check the relevance of a circumstance, you conduct some experiments. Remove a few circumstances and check whether the problem still occurs. If it does, the removed circumstances are irrelevant; otherwise, they are relevant for the problem.

2.3.1 Simplifying Manually

The method outlined in [KP99] results in a divide-and-conquer process:

> Proceed by binary search. Throw away half the input and see if the output is still wrong; if not, go back to the previous state and discard the other half of the input.

That method was intended for the input of a faulty program, but the same procedure can be used in general. Proceeding by that method simplifies the problem, but doing it manually has some drawbacks. The tests have to be run repeatedly and that exercise is quite mind-numbing. That is a good encouragement to try to automate that process.

2.3.2 An Algorithm for Automatic Simplification

In order to achieve an automation of the simplification, two integral parts are required. First, we need an automatic test that decides whether a set of circumstances is relevant or not and second, we need a strategy that implements the binary search method by running the test on some subset of circumstances. For example, unit tests are at our disposal to test the subset, only the strategy is missing.

Before we introduce the strategy, we have to complete the test: Sometimes, you cannot clearly decide whether a test has failed or passed. In that case, the test outcome is *unresolved*. Such a case would, for example, occur when you examine a certain subset of circumstances of a failure and your program refuses to start—the program run does not arrive at the incorrect state.

As mentioned above, proceeding by binary search is an appropriate attempt to simplify a problem. But on closer examination you run into a problem and its solution is trickier than extending the test. To adapt the binary search you must:

1. Throw away half the circumstances and run the test. If the test fails, continue the process with the remaining half.
2. If the test does not fail, go back to the previous state and discard the other half of the circumstances.

What to do if neither half fails, if testing both halves results in a passing result? Proceeding by simple binary search does not suffice anymore. The answer: do not test half the circumstances; test a smaller subset—quarters, eighths, sixteenth parts, and so on. Thus, after testing the halves, we test all quarters, then we test all eighths and all the rest of it if required. Now, we have all prerequisites to introduce a simplification algorithm.

The Delta Debugging Algorithm

The *delta debugging algorithm* is a general approach to isolate failure causes. It narrows down the differences between runs of a program—the *deltas*. The delta debugging instance *ddmin* is a

Let $\mathscr{C}$ be the set of all possible circumstances. Let $test : 2^{\mathscr{C}} \rightarrow \{✘, ✔, ?\}$ be a testing function that determines for a test case $c \subseteq \mathscr{C}$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (**?**).
Now, let *test* and $c_{✘}$ be given such that $test(\emptyset) = ✔ \wedge test(c_{✘}) = ✘$ hold.
The goal is to find $c'_{✘} = ddmin(c_{✘})$ such that $c'_{✘} \subseteq c_{✘}, test(c'_{✘}) = ✘$, and $c'_{✘}$ is 1-minimal.
The *Minimising Delta Debugging algorithm ddmin(c)* is

$ddmin(c_{✘}) = ddmin_2(c_{✘}, 2)$ where

$$ddmin_2(c'_{✘}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \dots, n\} \cdot test(\Delta_i) = ✘ \\ ddmin_2(\nabla_i, max(n-1, 2)) & \text{if } \exists i \in \{1, \dots, n\} \cdot test(\nabla_i) = ✘ \\ ddmin_2(c'_{✘}, \min(|c'_{✘}|, 2n)) & \text{if } n < |c'_{✘}| \\ c'_{✘} & \text{otherwise} \end{cases}$$

where $\nabla_i = c'_{✘} - \Delta_i, c'_{✘} = \Delta_1 \cup \Delta_2 \cup \dots \cup \Delta_n$, all Δ_i are pairwise disjoint,
and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{✘}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{✘}) = ✘ \wedge n \leq |c'_{✘}|$.

Figure 2.1: The Minimising Delta Debugging Algorithm in a Nutshell.

variant of the algorithm and simplifies problems by minimising the differences. See Figure 2.1 for a formal definition in a nutshell[3] of that variant.

The algorithm requires a test function *test(c)* that decides whether some input *c* results in a passing, a failing, or an unresolved test outcome. Having this function the algorithm is able to decide whether to continue with the first failing half, the second failing half or whether it must *increase the granularity n* and split the current (sub) set into four (or even more) subsets. The advantage compared to the simple binary search is the last step, increasing the granularity. At this point, the binary search does not provide a solution.

Executing the algorithm on an initial set of circumstances results in a *minimal set of circumstances* under that the problem or the failure occurs. Or, in terms of delta debugging: the algorithm *minimises a failure-inducing configuration*, a set of circumstances is called configuration.

One property of the ddmin variant is the *1-minimal configuration* as result of the algorithm: Every element in the configuration is relevant for the failure to occur, if you remove one element, the failure does not occur any longer. That configuration is also called *relevant*. The resulting configuration is not necessarily *minimal*. If you remove two elements (or even more), it could be that the test fails, too. To get the minimal configuration the algorithm has to test all subsets of the initial configuration. To test all subsets of a configuration with *n* elements the algorithm

[3]This *definition in a nutshell* is taken from [Bou04] with permission of the author and was modified slightly.

has to execute 2^n tests. To compute the relevant configuration, the number of tests carried out by ddmin is in $O(n^2)$.

Chapter 5 of [Zel05] contains a detailed description of the delta debugging process including some descriptive examples (simplifying input that causes a program to crash). For formal definitions, axioms, corollaries, and propositions see [ZH02]. That paper defines changes and n-minimal configurations, prooves best and worst case complexity, and shows that ddmin minimises, thus ddmin computes an 1-minimal configuration—and much more.

2.4 Causes and Effects

In Section 2.3, we introduced a strategy and an algorithm to simplify a problem. The minimising algorithm simplifies *all* circumstances. Thus, only the failing configurations make a contribution to the minimisation process, the passing ones are disregarded (see Figure 2.1, most notably the first two cases of the case differentiation).

Instead of that, you can alternatively simplify only the *difference* between a passing and a failing configuration (compare the case differentiation in Figure 2.4 with the one in Figure 2.1). If you also consider the passing tests, the algorithm needs far fewer test runs to determine the difference compared to the determination of the relevant configuration. Another advantage is *focusing*: the difference between a passing and a failing configuration is a *failure cause*, and the smaller the difference, the more precise the failure cause. See Section 2.5 for a discussion of "Isolating Failure Causes".

Before we will discuss the isolation, we have a look at causes and effects in general. While debugging, we can observe many circumstances. What are the most *expedient* circumstances? A small subset containing the *causes* answers that question.

2.4.1 Causality and Counterfactual Conditionals

In Section 2.1, we talked about causes—a defect *causes* a failure. To comprehend better the meaning of that term we will define it more precisely. Zeller defines that term in [Zel05] as follows:

> A cause is an event proceeding another event without which the event in question (the *effect*) would not have occurred.

With regard to the failure: without the defect the failure would not occur, thus the defect is a cause for the failure. This means that the largest part of the debugging process is the search for *causality*. After determining the possible causes of a failure, we have to verify whether they are actually causes. We try to reproduce the failure without the cause in question.

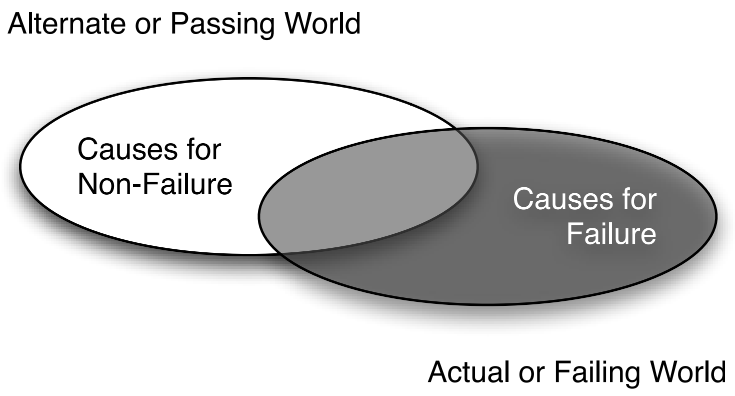


Figure 2.2: The theory of two worlds. A cause becomes a difference between the two possible worlds. In the majority of cases, the two worlds overlap each other—the *common context* excludes causes from the search space.

Adding David Lewis' disquisition on causes and effects with the analysis of *counterfactual conditionals* [Lew73] in terms of the *theory of possible worlds* [Lew86] a cause becomes a difference between the two possible worlds (see Figure 2.2 for an illustration): The semantic of the conditional $A > B$ is based upon considering the most likely situations (alternate worlds) in which A is true, and verifying whether B is true in all of them.

> Formally: $A > B$ is true at a world w if, in all the worlds closest to w where A is true, B is also true [Wik05a].

w is the *actual world*, all the world closest to w are the *alternate worlds*. A world that mirrors the actual world more suitably than another world is called *closer* to the actual world.

For instance, you can regard the alcohol level in one's blood as the cause of a car accident: "If I had not drunken ten beers, my car wouldn't have crashed in the ditch." The counterfactual implication of the negation: "Not drinking ten beers" implicates counterfactually "Car does not crash in ditch". In one of the alternate worlds, the person causing the accident drinks no beer and as a consequence the driver reaches home safely.

To get the cause for a problem (for example the car accident) we have to determine the difference between the world where the effect occurs and an alternate world where it does not occur. In our example, we know the actual world, but it is quite impossible to create all the alternate worlds (imagine the street was wet because it was raining, perhaps the ABS failed, the driver was hurt, ...). Normally, creating the alternate worlds at least generates huge costs. Therefore, we have to speculate about the causes, we have to bring to our mind what *would* happen in such an alternate world.

Fortunately, we want to debug a program. In that domain, it is much easier to create alternate worlds and to rerun such a world. We have influence on the circumstances (at least most of them, for example we can't control physical influences like solar winds)—while debugging the program is under almost total control.

2.4.2 Verification and Actuality

In order to verify whether some aspect of the alternate world or whether some properties of a program run causes the failure in question, the effect, we have to setup an *experiment*. The actual world, the world in which that effect occurs, is the real world. In the alternate world, we need to show that the property causes the effect. If we carry out our experiment in the alternate world and the effect does *not* occur, we have verified the cause—we have shown *causality*.

Finally, we come to the main problem with causes and effects: Finding (and verifying) *a* cause is trivial, but finding *the* cause among many is more complicated. If a program fails, a trivial cause would be: use another program in place of the failing. Using the other program, we have proved that the defect is causing the failure—omitting the defect, we cannot reproduce the failure. In the example of the car accident, you could completely ommit the "defect" by arranging for a cap.

In debugging, we do not want to find the first cause that comes along, we want to find a cause that is not far away from the defect, and we want to find the *actual cause*:

> An *actual cause* is the difference between the actual world and the *closest possible world* in which the effect does not occur.[4] [Zel05]

Because a closer world mirrors the actual world more suitably than another world, *the* cause is the *minimal difference* between the actual world and the closest possible one where the effect does not occur (see Figure 2.2).

2.4.3 Narrowing Down

A simple strategy to find an actual cause of a given failure follows:

1. Find an alternate world wherein the failure does not occur.
2. Narrow down the initial difference to an actual cause, using the scientific method to debug.

Here is a sketch of a scientific method to debug: Invent a *hypothesis* that is consistent with the *observations* about the failure. Make *predictions* using the hypothesis. Test the hypothesis by *experiments* and further observations: if the experiment complies with the predictions, refine your hypothesis; if not, you have to create an alternative hypothesis. Continue with making predictions and conducting the experiments until the current hypothesis can no longer be refined[5].

[4] In short and in the spirit of Occam's Razor: "Keep it simple" [Wik05d]

[5] [Zel05] contains an appropriate chapter about the scientific method. That chapter contains techniques of creating and verifying hypotheses and more on how to make the debugging process explicit.

These two steps are sufficient in order to find an actual cause. You need the current world—wherein the program fails—and an alternate world—wherein it does not fail. The alternate world needs not to be a world wherein the program has been corrected. Thus, there must be another difference causing the program to fail. For instance, that difference can be found in the input or in the program's execution. The main task is to find the initial difference. After that difference has been found, it can be narrowed down to an actual cause.

Zeller encapsulates the philosophical aspects of causes and effects in the twelfth chapter in [Zel05]. Just like the other chapters, that one contains enlightening examples in addition to the unavoidable theory.

2.5 Isolating Failure Causes

All prerequisites are now at hand: After we have seen how to (automatically) simplify a problem and having introduced a theory that allows us to narrow down causes, we will focus on the automation of most of the debugging process. As we will see within a short time, delta debugging is able to isolate failure causes automatically.

2.5.1 Isolating Automatically

Just like simplification (cf. Section 2.3.1) narrowing down causes as described in Section 2.4.3 can be a tedious and boring task if it is accomplished manually. Again, we will automate that process.

In order to automate the scientific method of debugging, we need the following ingredients:

- An automated test that decides whether the failure is present or not—it conducts our experiment,
- an instrumentation that narrows down the difference, and
- a strategy for proceeding.

How can we accomplish that process automatically? The remainder of this chapter will answer this question which is fundamental in the context of this work.

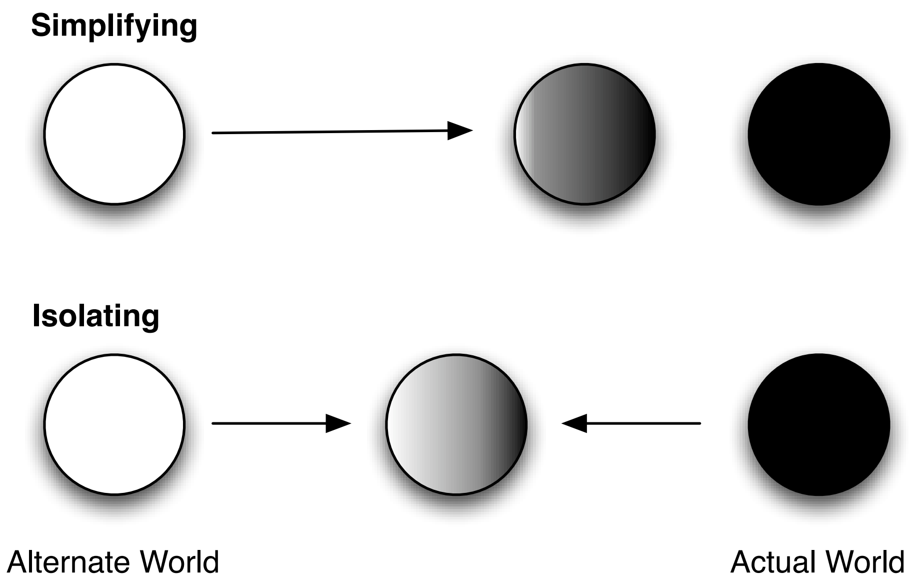


Figure 2.3: Simplifying compared to Isolation. While simplifying, we move the alternate towards the real, fixed, world. In contrast, while isolating we move both worlds towards one another to determine the smallest difference between the two worlds.

2.5.2 Isolating in Comparison with Simplifying

In Section 2.3 we saw how to simplify problems using the delta debugging algorithm. In terms of Lewis' theory of possible worlds (see Section 2.4.1) simplification means finding an alternate world whose difference is as close as possible to the real world—a most *simple* difference. To illustrate: The real world is *fixed* while we *move* the alternate world as close as possible towards the real one; we try to remove all differences that are not relevant in order to reproduce the problem. At the end of the process, we have a difference between the two worlds whose every aspect is relevant. An important property of that difference is being an actual cause (cf. Section 2.4.2).

As addressed briefly at the beginning of Section 2.4 there is a more efficient approach to narrow down differences than conducting simplification: *isolation*. While obtaining by simplifying a difference where each single aspect is relevant, isolation creates a *pair* of two worlds whose difference is minimal between them. Thus, that difference is an actual cause.

To illustrate again: Both worlds are moving forward each other to determine the minimal difference. In contrast to simplifying, where only the alternate world gets changed, we add circumstances to the real world (in fact, the actual world is not the actual one anymore, it is transformed into an alternate one closer to the actual than the failing one). Figure 2.3 demonstrates that process in a simplified manner.

Simplification and isolation compared in a nutshell:

- *Simplifying* results in a simplified difference: each circumstance is relevant for reproducing the problem. If you remove any of them then the problem is not reproducible.
- *Isolation* results in minimal difference between a failing and a passing world: that difference is one relevant part. If you remove this particular part then the problem is not reproducible.

Exemplified based on the car accident in Section 2.4.1: Simplifying that accident returns the set of all circumstances to make the driver completely drunk (for example, the driver has to drink eight beers). Isolating determines two worlds that differ by a set of circumstances. In one world the driver is slightly drunk, in another she is completely drunk. The difference is a failure cause and because it is minimal, it is an actual failure cause: the one beer that makes the difference between being fit to drive and being unfit to drive.

Commonly isolating is much more efficient than simplifying. Nevertheless, isolating has an important drawback: its minimal difference may have much lesser context than the one determined by simplification. In the example of the car accident, it is imaginable that the driver has suddenly pulled around the steering wheel because he lost control in consequence of being drunk. We may isolate that pulling around the wheel was the cause for the accident, but in that case we would not come to know that the driver was drunk—an important aspect when investigating that accident.

2.5.3 An Algorithm for Automatic Isolation

In Section 2.3.2 we introduced the delta debugging algorithm, an algorithm that automates the simplification. That algorithm can be extended to determine a minimal difference. The extension is called *general delta debugging algorithm*. The distinction between the two variants of the algorithm is the specification of the set for the next step to proceed: If the computed subset (the *mixed* world) fails the test, it will be regarded as the refined failing one. If it passes the test, it will be regarded as the refined passing one. Thus, instead of testing only a removal, an addition is also tested (therefore the two worlds are *moving* forward each other). See Figure 2.4 for a formal definition in a nutshell[6] of the general algorithm.

The new variant has the same worst-case complexity. If almost all tests have an unresolved outcome, the number of performed tests is in $O(n)$ (again, n is the cardinal number of the initial configuration). The algorithm is more efficient the more tests are failing or passing (resolved). If nearly all of them are resolved, the algorithm comes up to a binary search and it has logarithmic complexity. Therefore, a goal is to keep the number of unresolved tests to a minimum.

With reference to the scientific method (see Section 2.4.3), the delta debugging algorithm automates that method. It invents a hypothesis (configuration). It tests that hypothesis and refines the hypothesis depending on the outcome of the test. The delta debugging algorithm is a quite

[6]This *definition in a nutshell* is taken from [Bou04] with permission of the author and was modified slightly.

Let $c_{\checkmark}$ and $c_{\times}$ be test cases with $c_{\checkmark} \subseteq c_{\times} \subseteq \mathcal{C}$ such that $test(c_{\checkmark}) = \checkmark \wedge test(c_{\times}) = \times$. $c_{\checkmark}$ is the "passing" test case (typically, $c_{\checkmark} = \emptyset$ holds) and $c_{\times}$ is the "failing" test case.
The *Delta Debugging* algorithm $dd(c_{\checkmark}, c_{\times})$ isolates the failure-inducing difference between $c_{\checkmark}$ and $c_{\times}$. It returns a pair $(c'_{\checkmark}, c'_{\times}) = dd(c_{\checkmark}, c_{\times})$ such that $c_{\checkmark} \subseteq c'_{\checkmark} \subseteq c'_{\times} \subseteq c_{\times}$, $test(c'_{\checkmark}) = \checkmark$, and $test(c'_{\times}) = \times$ hold and $c'_{\times} - c'_{\checkmark}$ is *1-minimal*—that is, removing a single circumstance of $c'_{\times}$ makes the failure disappear.
The dd algorithm is defined as $dd(c_{\checkmark}, c_{\times}) = dd_2(c_{\checkmark}, c_{\times}, 2)$ with

$$dd_2(c'_{\checkmark}, c'_{\times}, n) = \begin{cases} dd_2(c'_{\checkmark}, c'_{\checkmark} \cup \Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\checkmark} \cup \Delta_i) = \times \\ dd_2(c'_{\times} - \Delta_i, c'_{\times}, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\times} - \Delta_i) = \checkmark \\ dd_2(c'_{\checkmark} \cup \Delta_i, c'_{\times}, \max(n-1, 2)) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\checkmark} \cup \Delta_i) = \checkmark \\ dd_2(c'_{\checkmark}, c'_{\times} - \Delta_i, \max(n-1, 2)) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\times} - \Delta_i) = \times \\ dd_2(c'_{\checkmark}, c'_{\times}, \min(2n, |\Delta|)) & \text{if } n < |\Delta| \\ (c'_{\checkmark}, c'_{\times}) & \text{otherwise} \end{cases}$$

Figure 2.4: The General Delta Debugging Algorithm in a Nutshell.

simple strategy for proceeding and a human may be more *creative* while refining the hypothesis. However, the automation of that tedious and boring (and therefore error-prone) process is a noticeable improvement[7].

In the next chapter about "Failure Inducing Changes", we will see how delta debugging allows for the automatic determination of changes that introduce a defect in our code.

In Chapter 13 of Zeller's book [Zel05], you will find detailed information about that topic. It includes several examples that apply the delta debugging algorithm in order to isolate failure inducing *practical* circumstances including failure-inducing input and failure-inducing schedules. If you are interested in the formal definitions, proofs and propositions have a look at the appendix of that book. [ZH02] also contains all formal definitions, axioms, corollaries, and propositions.

[7] And remember Occam's Razor: "Keep it simple".

3 Failure-Inducing Changes

In Chapter 2, we introduced the theoretic equipment required for this work. Using that basic equipment the introduction of failure-inducing changes and their determination can be done in a straightforward way. Before we derive the concrete methods and techniques to determine such changes, we will have a closer look at code changes in general.

3.1 Code Changes

You are writing a program and you know that it has worked as expected yesterday. While implementing new features, improving the current and refactoring your code, you inevitably introduce some changes to your program code. These changes may alter existing code, may create new units like classes, or may remove existing units. Even being an experienced and gifted programmer, it is not too unlikely that you accidentally introduce some defect while programming—suddenly your program fails. Because you have learned that a test case is the first step before a program can be debugged and you want to ensure that your software has an acceptable defect rate, you are using unit tests to test your code (cf. Section 2.2).

To be more concrete and giving an example: Imagine you test your code during the development process using the JUnit unit testing framework[1]. In your current Java project you work on a package that contains classes concerning customer creditworthiness. At 9:30, you assured yourself of the correctness of the two classes Customer and PremiumCustomer by running their unit tests. During your workday, you change that package by altering the class Customer, the class PremiumCustomer is removed and a new one called Debitor is created. At 14:00, you rerun the unit tests and you find out that one of the unit tests is failing now—a typical *regression*. Thus, you have introduced a defect in the code and that defect causes a failure.

After the unit test has failed, the main question is: "Why does the test fail?", or in other words "What change has introduced that failure?". Having changed only three classes in a short period debugging that failure and finding the failure-inducing change is not too complicated. You could conduct the scientific method (see Section 2.4.3) to isolate that change, even using a simple trial-and-error method will arrive at your destination. Thus, having only about a handful changes,

[1]JUnit is an unit testing framework for the Java programming language. It was created by Kent Beck and Erich Gamma and was released as open source software. That framework is arguably the most common instance of the xUnit architecture. See the website of JUnit [GB05] for more information.

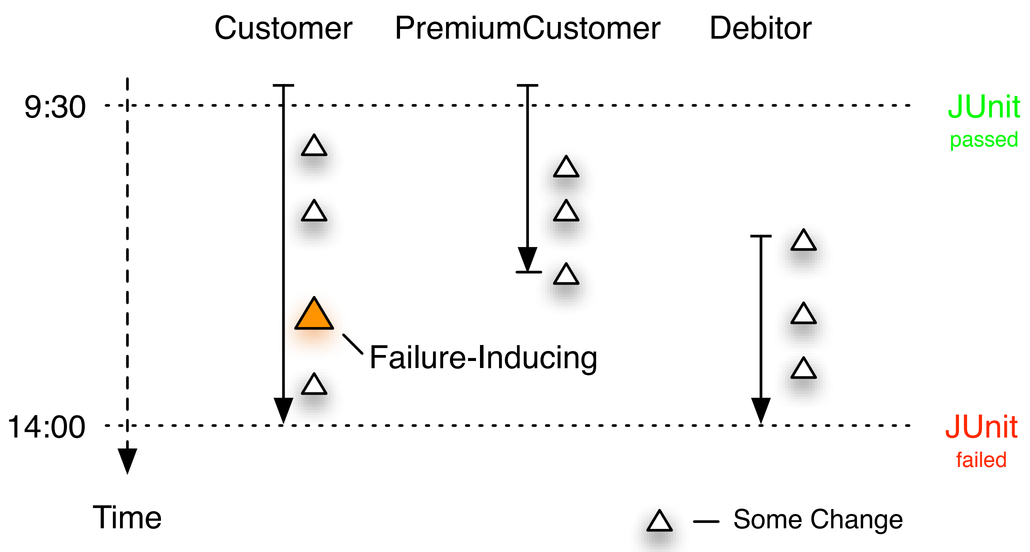


Figure 3.1: A few changes introduced a defect. Finding the failure-inducing change by hand is quite easy.

isolating the defect in the code should not be too difficult and time-consuming. Figure 3.1 illustrates that case. In our example one change in the Customer class has introduced the defect, that change is failure-inducing.

We can easily extend the complexity of that example. If we consider a large project, more than only a few classes (and other fragments of code) come into play. More than one hundred classes will be the rule, not the exception. As well as the complexity in space (for example the number of classes), the complexity in time may increase. The period in question could be a few days or even weeks. Imagine a large project with many team members. One member of the team has programmed a new module of the software under development at the beginning of this year. Over a period of time the software is under heavy development, many changes were introduced. The team member responsible for that module has meanwhile left our team, taking out some part of his knowledge. Then, in the middle of the year a unit test covering the module fails. Now, it is much harder to isolate the change that introduced the failure. Figure 3.2 illustrates the case with many changes. In comparison with Figure 3.1 it is obvious by intuition that the debugging is harder in the second case.

You could argue that the failure-inducing change should be one of the recent ones. Nevertheless, that constraint may not be valid. The failure-inducing change that has introduced the defect or is close to the defect may have slipped in much earlier. In consequence of the recent change, the failure was discovered, but the actual defect exists for a longer time.

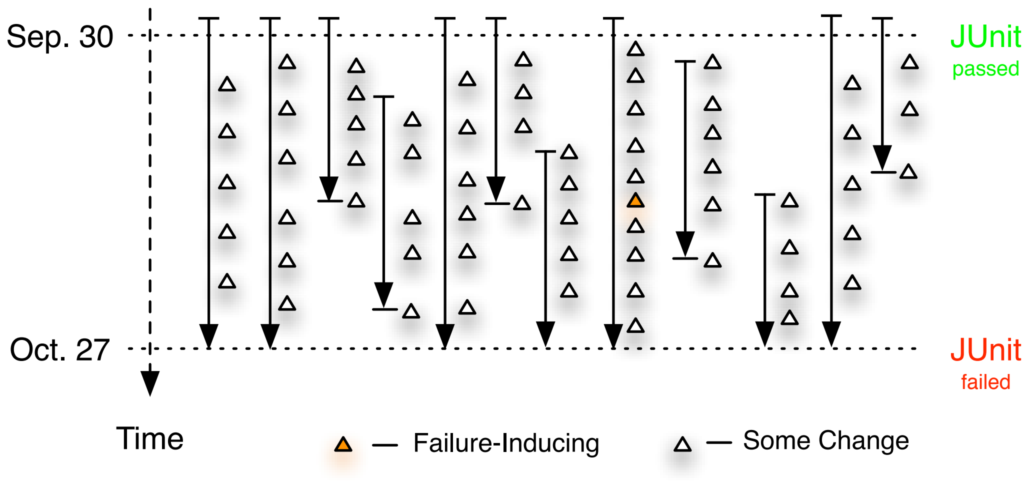


Figure 3.2: Many changes introduced a defect. Finding the failure-inducing change by hand is annoying and time-killing.

3.2 Version Differences

In the context of the theory of possible worlds (see Section 2.4.1) the runs of two program versions, the one that is failing and the passing one, can be regarded as two alternate worlds. In the first one, our program code contains the defect and therefore the unit test fails. In the second one that defect does not exist and therefore the unit test passes. From the programmer's point-of-view the cause of the failure can be found somewhere in the difference between the two program versions (see Figure 3.3).

The universal concepts *actual cause* and *closest possible worlds* are transferable to causes required for debugging. If we want to find the actual cause of a program failure, we have to search the closest possible world wherein that failure does not occur. Thus, in order to find the actual failure cause in a program code we will search the minimal difference between the actual (failing) code and the closest possible code where the failure does not occur.

In Section 2.5, we have seen how to isolate failure causes, and the delta debugging algorithm allows to accomplish that process automatically. Now, we can arrive at a conclusion: *Using the delta debugging algorithm we can determine failure-inducing code changes automatically.* Thus, we can narrow down the causes of a regression by focusing on the changes we made.

3.3 Challenges

Before we introduce the *general plan*, the steps to use the delta debugging algorithm, we are faced with some challenges. Most of them are problems and limitations of the delta debugging

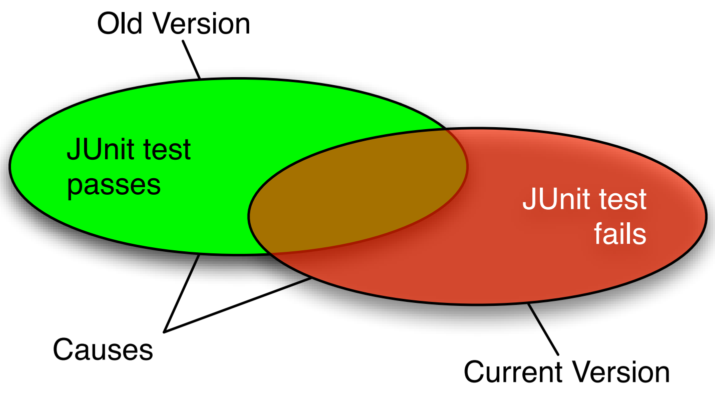


Figure 3.3: The universal concepts *actual cause* and *closest possible worlds* are transferable to causes required for debugging. The difference between the two program versions contains the failure-inducing change.

algorithm as described in [Zel05]. We do not discuss these issues in general; we focus on the aspects that are important to code changes.

- *An appropriate alternate world.* The choice of the alternate world (the program version that does not contain the defect) has major influence on the time required by the algorithm to determine the result. In order to keep the search space as small as possible we should choose a program version that is as close as possible to the failing version.

- *Decomposition of the changes, granularity.* A change can alter a single line of a file or replace a whole directory containing multiple files. In order to get a *small* failure-inducing difference we should decompose *large* changes into smaller. The motivation should be clear: We can treat the initial difference as one change. Consequently, the algorithm can determine the failure-inducing one very quickly. However, the result is not too helpful. To gain enlightening insights, the failure-inducing change has to be small, affecting only a small part of the software to be debugged. One solution would be the decomposition of the changes into changes that affect only a few lines.

- *Reproducing* the *failure.* The test that checks the mixed program version (constructed during the algorithm's run) has to ensure that a *different* failure is not interpreted as a failing outcome in terms of the algorithm. Only the failure of the original test case should be returned as failing to the algorithm, all other failures should result in an unresolved outcome. Applying some changes can cause another defect in the code, thus another failure is caused by that defect. If our test function returns failure in that case, the algorithm will not be able to isolate the actual cause. For example, our test function could compare the backtrace at the time of the failures. An alternative could be comparing the exception or a combination of both methods.

- *Correction of the defect.* The algorithm returns an actual cause that is a *fix* rather than a correction (see Section 3.6 for details). The algorithm is not able to distinguish between fixes and corrections, someone has to *decide* where and what the defect is. In order to get the actual defect further investigation by the programmer may be required.

- *Inconsistency.* If we decompose the difference between the actual and the alternate program version, we could get changes that may have to be combined to get testable code. For example, if a new functionality was added and that is used in another part of the code, the change that introduces the usage will depend on the change that introduces the definition. The algorithm would handle this (an unresolved outcome), but such changes negatively affect the runtime of the algorithm (remember: if nearly all tests are unresolved the number of tests can equal the number of changes squared).

While designing and implementing, we have to keep these challenges in our mind. Now, we are ready to introduce the single steps that are required to use delta debugging on code changes.

3.4 The General Plan

In order to use the delta debugging algorithm we need at least two things: a configuration and a test. As discussed in Section 3.2 the set of changes between two program versions can be treated as a set of failure inducing circumstances. That is the first ingredient required by the algorithm, only the test remains. If you use some testing technique, then that technique can be extended to satisfy the requirements of the delta debugging test. Normally, a test reports whether the aspect under test passes or fails that test. Extending the test with the unresolved outcome in some appropriate way complies with the algorithm. Thus, we have all required ingredients. In the context of this work, we use unit tests to test our code. Hence, we will concentrate on unit tests. In Section 4.5, we will see other tests that may be used to determine other types of failures that can be debugged with the delta debugging algorithm.

Now, we will see the sequence of steps to use the algorithm on changes:

1. *Establish a history.* To know when a failing unit test was passing we have to remember the date. In addition, the date should not be the date of any passing run; in fact, it should be the one of the latest passing. To reduce the search scope the chronological difference between the passing and the failing run should be as small as possible assuming that a shorter period of time comes along with fewer changes (cf. Section 3.3). One possibility to store the dates is a database.

   Storing the dates of the passing test run is insufficient. If we know the date of the latest passing run, we also need the associated version of the code. Without that version, we cannot determine the initial set of changes. There are tools exactly designed for this task: so-called revision control systems[2].

Keeping track of the dates in a database and having a revision control system, we can establish a history of the former running versions of our program.

2. *Determine changes.* If one of our unit tests fails, we have to determine the initial changes between the current failing and an older passing version of our program. Using the established history, it is a relatively unproblematic procedure to get these changes.

3. *Apply the algorithm.* In order to apply delta debugging to the initial changes, we must construct a function $test(c)$ that decides whether the set of changes c results in a passing, a failing, or an unresolved outcome (cf. Section 2.3.2). The following steps describe that test. It can test any subset of changes.

   a) *Apply current changes.* Before we can test the *mixed* version of our program we have to create it. Thus, we must apply the current subset of changes selected by the algorithm to the source code. For instance, we could use the UNIX PATCH tool[3] to apply the changes assuming the changes have the appropriate format.

      It may be the case that not all changes can be applied, for instance if some of them depend on other changes not contained in the subset. In that case, the test has to return unresolved as outcome (and the test will return without even running the unit test—but it should undo all applied changes, see below).

   b) *Reconstruct the program.* After all changes have been applied successfully, we must reconstruct or build the program. For example the changed source code has to be compiled or other files may require some other procedure in order to build some artefacts.

      Applying a subset of changes that is selected out of a large set at random it is quite likely that the constructed source code is not translatable, for example because of a compilation error. Again, the test will return the unresolved outcome and will not run the unit test. Before returning the result, the applied changes should be undone.

   c) *Run the unit test.* Having the constructed program, we can run the unit test. If the unit test passes, we will return the passing outcome. If it fails, we have to decide whether we should return the failing or the unresolved outcome. See Section 3.3 for a discussion on that topic.

[2] A revision control system is a system that typically stores information about versions of software and that controls the shared access on the source code. The system collects all changes in the central *repository* and provides the changes with a timestamp; if required, one can return to an earlier version of the software. Examples for such a system are the well-known Concurrent Versions System (CVS) and the newer Subversion (SVN), which aims to replace CVS by addressing some of its limitations.

[3] PATCH is a Unix tool that changes files containing ordinary textual characters (for example source code files) according to some instructions. These constructions are often contained in a separate file, called a patch file. They define operations like adding, deleting or changing a line.

d) *Undo current changes.* Before our test returns its outcome to the algorithm, we undo all applied changes. Thus, we have again the unchanged version and in the next round, we can start from scratch. If we are not able to undo the changes, we have to get the unchanged version in another way.

e) *Return outcome.* The last step is to return the computed outcome to the algorithm. Depending on that outcome the algorithm selects another subset and runs this test again, or it has completed its work and returns the determined failure-inducing changes.

You have seen a rough sketch of all required steps to determine the failure-inducing changes. Now, the tasks for the platform are defined: it has to provide a history, methods to apply and undo changes, automated construction and regression tests—and the delta debugging algorithm, of course. Before we come to the practical part and describe the platform, we will discuss some (optional) optimisations.

3.5 Optimisations

We have made available all the ingredients required to build a platform that enables us to determine failure-inducing changes. However, some issues demand optimisation. The first one listed here is related to delta debugging in general, the subsequent issues are specific to applying the algorithm to changes.

- *Caching.* The delta debugging algorithm gives no guarantee that each configuration is tested only once. Because applying changes, constructing the code, running the unit test and undoing the changes in a final step is very time-consuming, we should *cache the outcome* of a run test. If the algorithm wants to test a configuration a second time, we can look up its outcome in the cache and return it immediately.

- *History.* One possibility to get the initial changes is to consider only the passing and the failing program version, thus we create the changes between these two versions without considering other information. As mentioned before, we will use a revision control system to get the changes. Such a system contains more information about the difference: it records all changes and therefore contains all versions between the passing and the failing one, it knows the *chronological order* of the changes (cf. Figure 3.1). By grouping these changes by their creation time, we could ensure that later changes are always applied along with earlier ones. Thus, we get fewer errors during the reconstruction stage and therefore more resolved test outcomes. The benefit: unresolved test outcomes result in a quadratic number of tests, in contrast to resolved: they result in a logarithmic number of tests—we come up to a binary search along the change history.

- *Reconstruction.* Before we can run the unit test, we have to reconstruct the program. To reduce the time spent for this stage, we should not do a *full* reconstruction that rebuilds all artefacts. Instead of that, we should do an *incremental* reconstruction. For instance, only the sources changed since the last construction have to be compiled.
- *Grouping.* After applying a subset successfully, the mixed program version may not be buildable, that case results in an unresolved outcome. For instance, change D adds the declaration of a variable and change R adds a reference to that variable. Obviously, change R depends on change D. Therefore, every subset which contains R but not D results in a program version that cannot be constructed. If we pay attention to the *scope* of the changes, we can speed up the debugging process by returning an unresolved outcome without even applying the changes. Furthermore, we can keep those changes together that affect the same directory, file, class or method. In further iterations, we can break up that cohesion to get repeatedly a smaller set of failure-inducing changes.

If we pay attention to all of these optimisations while constructing the platform, the process to determine the failure-inducing changes would be much faster than without them. Therefore, we should implement at least some of the described optimisations.

3.6 Fixes or Corrections?

If we found a failure-inducing code change using delta debugging, that change is not necessarily a *correction*. If we undo that change, the failure will vanish. Therefore, the undo of the failure-inducing change is a *workaround* that suggests a *fix*. Such a fix may be an important starting point for further investigation on that failure. At least, the programmer has to proofread the changes. Thus, the determination of failure-inducing changes may not provide a tool for automatic corrections of defects, but may be an important and helpful technique to improve the debugging process—easing the programmer's life.

In order to achive that goal, the determination has to be reliable and has to result in changes that are close to changes that would be accomplished by a real programmer to correct the defect.

[Zel05] contains a section about failure-inducing changes in addition to failure-inducing input and thread schedules. This chapter is based on the section in that book, but it is organized in a different way. So most of the information provided here could be found in another order in the book.

Now, we are ready to describe the platform.

4 The Framework

This chapter contains the description of the framework called DDCHANGE, the core of the platform. The framework consists of about 260 files, including metadata about the framework and nearly 100 Java classes defining circa 560 methods—without making mention of the unit tests.

Describing every aspect of the framework goes beyond the scope of this elaboration and would lengthen it with unnecessary material. The comprehensive API documentation covers most of the framework's documentation. It contains the specification of public, protected, and private packages, classes, interfaces, fields, and methods. The PDF version of that API documentation would span over 390 pages. Therefore, it is not included in the printed version of the elaboration; enclosed you will find a CD that hosts the API amongst other useful documents. See Appendix A.1 for more information about the CD.

The following sections cover the most relevant aspects and concepts of the framework at a glance. In conjunction with the theoretical (see Chapter 2) and related to practise (see Chapter 3) background it should be easy to use the framework, to navigate in the source code, to extend the framework in order to customise it, and to contribute new features.

First, we will briefly describe a framework in general and an existing implementation of the delta debugging algorithm. In the following main part, we will describe the several components of the framework considering *the general plan* (see Section 3.4). The succeeding sections cover the attempts to gain some quality and some examples that explain how to extend the framework. The closing sections describe some possible optimisations and draw a conclusion about the framework.

4.1 Why a Framework?

Class libraries can be classified into two categories (cf. [Bal01]):

- *Plain class libraries.* These libraries do not force predetermined application architecture; such a library is a collection of independent classes designed primarily for reusability of code. In the context of the Java programming language, the "Apache Jakarta Commons Lang" [Tea05b] component is probably the best known class library.

- *Frameworks.* The complement of a class library as described above is a so-called framework. A framework forces well-defined application architecture by adapting the principle *Inversion of Control*[1]. One well-known framework for the Java language is the Spring Framework, a layered Java/J2EE application framework, based on code published in [Joh02].

The purpose of frameworks is the reusability of design, not primarily the reusability of code. It defines the architectural design of the classes and objects and their responsibility and collaboration; it specifies the classes' coupling and the control flow of the provided operations. A concrete application registers own classes to the framework by using well-defined interfaces (called Dependency Injection[2]); the framework calls these classes in its own control flow. Because the framework defines a preliminary design, the programmer is able to concentrate on the details of the application.

Frameworks can be classified by the techniques used to extend them, which range along a continuum from *White-Box-Frameworks* to *Black-Box-Frameworks* (cf. [JF88]).

- *White-Box-Frameworks.* Based on inheritance and subclassing. They rely on inheritance and dynamic binding to achieve extensibility. Existing functionality is reused and extended by inheriting from framework base classes and overriding pre-defined hook methods using patterns like *Template Method* [GHJV95].
- *Black-Box-Frameworks.* Based on composition and parameterisation. They provide extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by defining components that implement a particular interface and integrating these components into the framework using patterns like *Strategy* and *Abstract Factory* [GHJV95].

Using a White-Box-Framework requires detailed knowledge by the programmer about the framework's internal structure. Consequently, an application that uses such a framework tends to be coupled tightly to the framework's inheritance hierarchies. In contrast, Black-Box-Frameworks are structured using composition and delegation more than inheritance. Thus, they are generally easier to use and extend.

Frameworks in general reduce cost and improve quality [FS97]; they reduce the required source code to write an application. Because they encapsulate implementation details behind interfaces, they localise the impact of changes, too. The usage of interfaces enhances reusability; reusability leverages the domain knowledge and can yield improvements in programmer productivity [FS97]. These advantages have to be seen alongside several problems. According to [Pre96], a

[1] Inversion of Control (IOC) is an object-oriented design pattern. That principle inverts the way an object gets its dependencies—it adopts the Hollywood principle "Don't call us, we'll call you!". The application hands the control flow of particular subroutines over to the framework instead of managing that flow by itself. Cf. [Wik05c] and [GHJV95]

[2] Dependency Injection is an adaptation of IOC.

high development effort is required to realize a framework. Learning to use a framework effectively requires large investment of effort—the learning curve is steep [FS97].

In order to allow the straightforward development of tools that determine failure-inducing changes, the platform's *core* will be a framework that tends to be a Black-Box-Framework. That framework called *DDCHANGE* will provide the required functionality and operations to accomplish the general approach to determine these changes (see Section 3.4). DDCHANGE must be easy to use to ensure cost-benefit over designing the tools from scratch. It will provide built-in mechanisms for customisations like factories and persistent storages. Because the tools depend on that framework, it must be stable and well tested.

Note: The guest editorial by Fayad and Schmidt in [FS97] contains a compact but extensive overview on object-oriented frameworks. It is a good source for further readings on that topic.

4.2 Delta Debugging Algorithm

The implementation of the delta debugging algorithm is certainly one of the most important components of the framework. Fortunately, Philipp Bouillon has implemented the algorithm in the Java programming language as part of his diploma thesis [Bou04]. DDCHANGE uses an extended and partially reworked version of that implementation.

Bouillon has developed an automated debugging framework using Eclipse (see Section 5.2.1 for more information about Eclipse) to automate the process of minimisation of failure-inducing input. That framework is divided into two Eclipse plug-ins; one of them contains the implementation of the algorithm. That plug-in depends on some Eclipse libraries, so it was necessary to resolve these dependencies in order to create a simple JAR file that contains the algorithm. Now, Maven (see Section 5.1.1 for more information about Maven) can be used to create that JAR with a modicum of effort.

The changes required by the task to decouple the implementation from Eclipse were minor compared to the second change on the implementation. Bouillon's implementation does not provide a working cache for the test outcomes. As described in Section 3.5, the algorithm gives no guarantee that each configuration is tested only once. Thus, a working implementation of an outcome cache may result in noticeable performance improvements—depending on the duration of a single test. Because we have to reconstruct the code amongst others steps (see Section 3.4), that cache should have a huge impact on the run-time.

Bouillon has tried to use a *treap*[3] to store already known test outcomes, just like the original implementation in the Python programming language by Andreas Zeller [Zel01]. In Java, Bouillon's implementation of this treap uses a large amount of memory. Thus, searching for an alternative implementation was still an issue.

[3]A treap (binary search tree + heap) is a binary search tree. Each node consists of two elements: a key and a priority (cf. [SA96]).

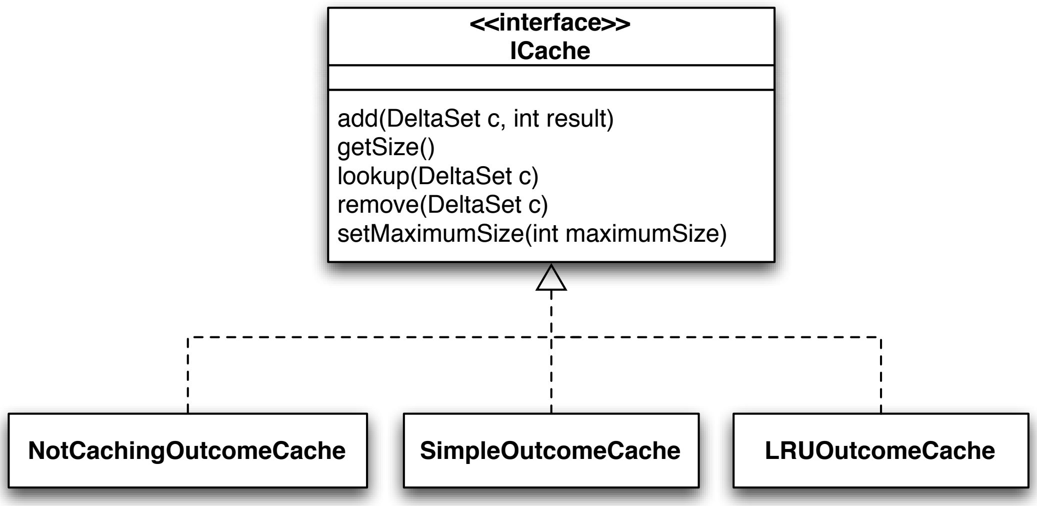


Figure 4.1: UML class diagram of new cache for delta debugging algorithm. The diagram shows only the most important methods.

In addition to the release of the JAR library, DDCHANGE provides a new, working implementation of the cache. The extended version of the algorithm contains a new interface `ICache` and three implementations of that interface.

- `NotCachingOutcomeCache`. This outcome cache does not cache any outcome. Added outcomes are ignored, the lookup methods return “no outcome exists” at every call.
- `SimpleOutcomeCache`. A cache that holds the outcomes in a `Map`[4]. You cannot set the maximum size of this cache. Thus, it grows until an `OutOfMemoryError` is thrown.
- `LRUOutcomeCache`. A cache that holds the outcomes in a LRU `Map`: A map which has a maximum size and uses a Least Recently Used algorithm to remove items from the map when the maximum size is reached and new items are added.

Figure 4.1 shows the UML class diagram of the cache; only the most important methods are included. The API documentation (see Appendix A.1) contains the complete specification of the cache.

The main delta debugging class `DD` provides two new methods, a getter and a setter for the cache used by the algorithm. By default, `DD` uses the `NotCachingOutcomeCache`.

In order to add instances of the class `DeltaSet`[5] to a `Map`, that class was extended by proper implementations of the methods `equals(Object)` and `hashCode()`.

[4]An instance of `java.util.Map`

[5]`DeltaSet` stores a specific test configuration. This is the primary data structure the algorithm operates on.

Extended by the cache as described above, the algorithm does not test the same configuration twice. According to first observations, enabling the cache saves about 50% of the tests as performed without a cache—depending on the number and type of initial changes, and so on. Because the run of an individual test in our context tends to be time-consuming, that interim result is promising. However, further research on that topic is required to get reliable numbers.

Having the new interface, other (maybe more sophisticated) implementations are imaginable. For instance, a cache that provides memory and disk stores. Such a cache is especially suited to store outcomes of profoundly time-consuming tests. An existing cache appropriate to perform that task is the `ehcache` [LDK05], a pure Java in-process cache.

Further research on caching test outcomes goes beyond the scope of this work; even though the extended implementation allows for accomplishing that research.

One last minor change affects the licence under which the implementation is released. Bouillon has released his implementation under the "Common Public License v1.0". In the context of DDCHANGE the new implementation is released under the "Eclipse Public License v1.0" (not without prior consultation with Philipp Bouillon). The Eclipse Public License Frequently Asked Questions [Off04] contains information about the differences between these licenses.

In the following sections, we regard this component as a black-box. Thus, we pass all required parameters (for example the tester and the initial set of changes) to that component and it will compute the failure-inducing changes.

4.3 Components

A framework is an appropriate technique to improve reuse—different applications can be realised by instantiating it. However, the instantiation process can be complex, requiring a good understanding of the framework design and implementation.

To accomplish the desired flexibility, frameworks provide special constructions. They have fixed parts, called *frozen spots* [Pre99], which reflect the common behaviour of applications in the domain. Similarly, frameworks have parts that need to be kept flexible, called *hot spots*, which have to be adapted according to the specific requirements of concrete applications derived from the framework.

In the following sections, we will describe the different components concentrating on the hot spots of this framework. This will enable you to customise the framework in order to instantiate own tools or applications. If we can gain valuable insight, we will discuss frozen spots and default implementations of hot spots in addition. As we will see in Chapter 5, the framework provides default implementations for all hot spots allowing the rapid derivation of a tool. Furthermore, it is flexible in order to instantiate tools with new features and using the framework for the most important tasks at the same time.

4.3.1 Database

As seen in Section 3.4, we have to establish a history of test runs to remember the date of the most recent passing run. We have to store the run's date and the version of the program. The program version is stored using a revision control system; it keeps track of all work and all changes. In the context of this work, we act on the assumption that such a system is used. Thus, the developer commits the changes to that system. Consequently, DDCHANGE is not responsible to store the version information. However, it has to obtain the initial set of changes from the system (see Section 4.3.3).

As the component name suggests, its domain is the *persistence of test results*.

Test Results

A test result is composed of the outcome (passed, failed or unresolved), a possible stack trace, the date of the test execution and a reference to the run test case. The test case consists again of a method that executes the test, a class that contains the method, a package that contains the class (may be unnamed), and a reference to a project that contains the package. This results in three classes as shown in Figure 4.2. The three classes can describe a test and its results unambiguously; results of JUnit test runs can be stored in instances of these classes without going a long way round.

Persistence of Results

DDCHANGE uses *Hibernate* [Com05a], an object/relational persistence and query service for Java, to store the results in a database. That open-source object-relational mapping tool allows the bidirectional mapping from Java classes to database tables and provides data query and retrieval facilities. The advantages in a nutshell: you can develop persistent classes while using the common Java idiom (associations, the Java collections framework, and more), most of the common data persistence-related programming tasks are done by Hibernate, and you are independent of a particular database product.

Describing Hibernate and its features goes far beyond the scope of this work. The Hibernate documentation [Com05b] provides users FAQ, product evaluation, discussion on performance, tutorials, and more.

DDCHANGE uses Hibernate's bidirectional many-to-one association to declare the parent/child relationship between the three classes that represent test results. One advantage of that bidirectional association is the automated deletion of orphan children. If we remove a project, all associated test cases and results are removed automatically from the database; the same holds true for the test cases.

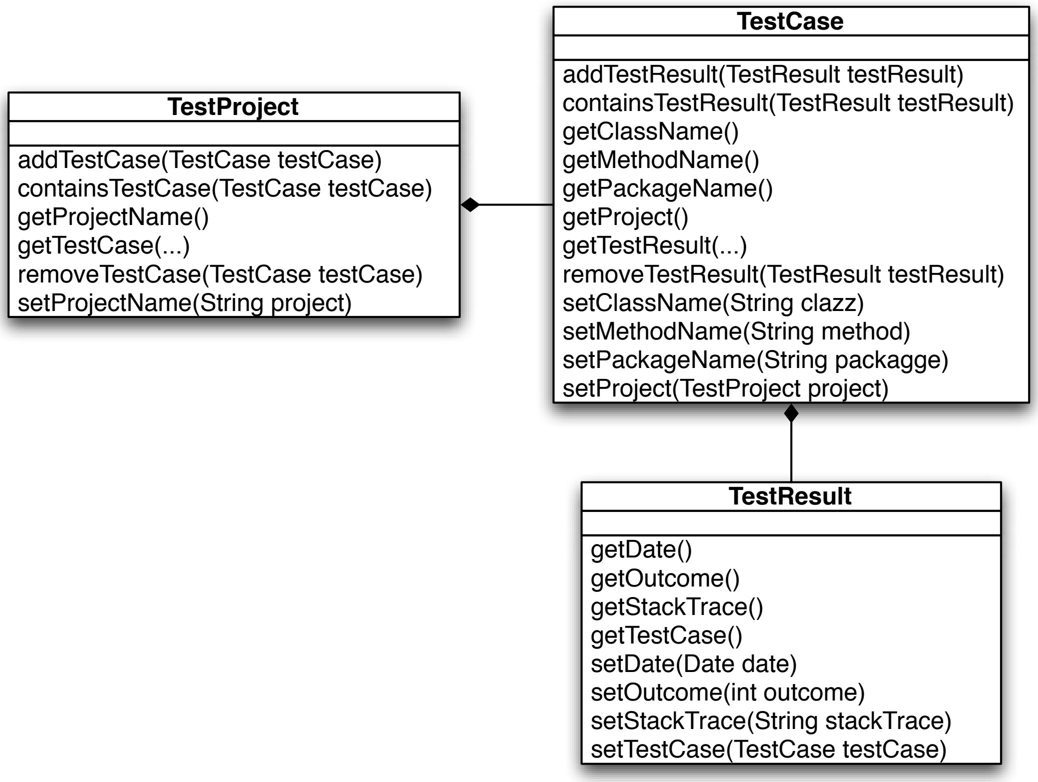

TestProject
addTestCase(TestCase testCase)
containsTestCase(TestCase testCase)
getProjectName()
getTestCase(...)
removeTestCase(TestCase testCase)
setProjectName(String project)
TestCase
addTestResult(TestResult testResult)
containsTestResult(TestResult testResult)
getClassName()
getMethodName()
getPackageName()
getProject()
getTestResult(...)
removeTestResult(TestResult testResult)
setClassName(String clazz)
setMethodName(String method)
setPackageName(String packagge)
setProject(TestProject project)
TestResult
getDate()
getOutcome()
getStackTrace()
getTestCase()
setDate(Date date)
setOutcome(int outcome)
setStackTrace(String stackTrace)
setTestCase(TestCase testCase)

Figure 4.2: UML class diagram of the three classes that represent a test result unambiguously. Hibernate persists these classes. The diagram shows only the most important methods.

Hibernate supports different caches to speed up the database access. DDCHANGE uses *ehcache* [LDK05] as second-level cache system with Hibernate. The default configuration uses ehcache for JVM-level read-write caching to ensure best performance for reads and writes. If you want to use that strategy in a cluster, you should ensure that the underlying cache implementation supports locking. Hibernate's built-in cache providers do not.

The most important aspect in the context of this work is the independence from a particular database. Thus, we can store the results in nearly every (central) database. Hibernate supports currently more than twenty different SQL dialects, including Oracle, MySQL, Progress, and HypersonicSQL.

Provided Database

One goal of DDCHANGE was, that it should be usable out-of-the-box. To fulfil that requirement, DDCHANGE provides a SQL database ready to use. The framework includes classes to launch and stop an instance of the *HSQLDB database* [hDG05a]. HSQLDB is an open-source relational database management system written in Java. It is small in size, can run completely in memory (in-memory and disk-based tables can be used), and it is fast. As mentioned above, Hibernate supports HypersonicSQL, the HSQLDB SQL dialect.

To run the database, you have to instantiate one of the subclasses of `DatabaseConfiguration` and to pass it to a concrete `DatabaseLauncher`, an abstract launcher for a database. Currently, there are two different implementations of configuration and launcher. The first one launches an HSQLDB instance in the so-called "In-Process (Standalone) Mode". In that mode, the data is not converted and sent over the network. One drawback of this mode is that it is not possible to connect to the database from outside your application. The second mode, "Hsqldb Server Mode", provides external accessibility, applications programs (clients) can connect to the server using the HSQLDB JDBC driver. According to the HSQLDB manual [hDG05b], the first mode can be faster than the second (for most applications), so you should use the first mode when you deploy your application. The manual contains further information about the two modes. Figure 4.3 shows an UML diagram of the classes to configure and launch a database.

If you instantiate one of the two implementations of the configuration, sensible default values are chosen (for example the database files are stored in the system's temporary folder). Thus, it is sufficient to instantiate a configuration and to pass it to the matching launcher—two steps to configure a runnable database.

Data Access Object

The framework uses a *Data Access Object (DAO)* to isolate the application from the underlying persistence technology (in our case Hibernate). A DAO is a component that provides a common

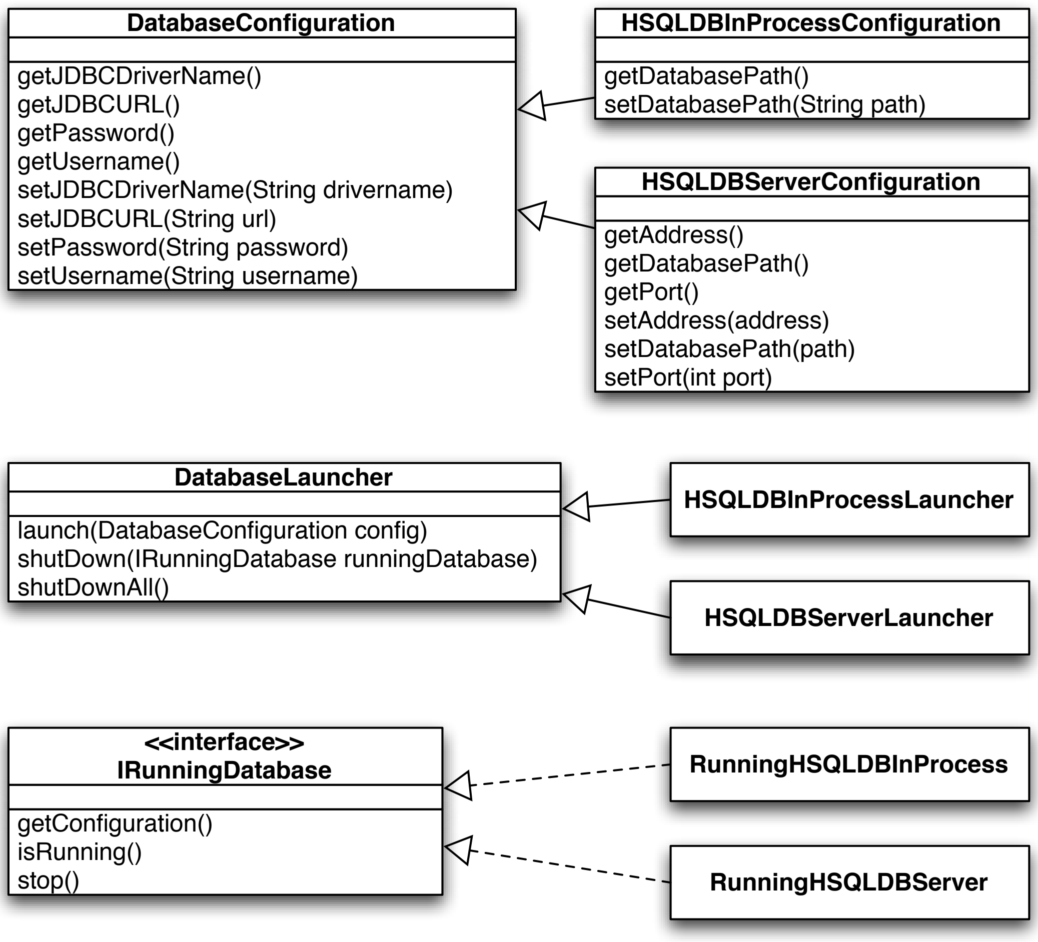


Figure 4.3: UML class diagram of the classes that provides a mechanism to launch a database. DDCHANGE can launch HSQLDB database instances innately. The diagram shows only the most important methods.

interface between the application and one or more data storage devices [SM05]. Hibernate hides programming tasks like writing SQL queries, but you have, for instance, to open and close so-called sessions. Some knowledge about Hibernate is required to persist and request test results.

The advantage of using DAO is that any business object (which contains application or operation specific details) does not require knowledge of the final destination for the information it requests or modifies. As a result, if it is necessary to change where or in what way that data is stored, that modification can be made without requiring changing the application. The underlying technology can be changed or replaced without the need to change other parts of the framework. Thus, your application can relay on the DAO, even if DDCHANGE would use another storage service.

DDCHANGE provides an *Abstract Factory* that produces factories, which produce again DAOs. The concrete subclass `HibernateDAOFactory` provides *Factory Methods* to produce DAOs that use Hibernate. Thus, if you want to use another storage service to persist results, you should implement an own factory that returns `IResultDAO`, the interface that specifies the DAO.

As we have seen, storing the test results in a database using DDCHANGE is quite easy and unproblematic. DDCHANGE provides classes that represent unambiguously test results, it can launch a database, and DAOs isolate your application from the used object-relational mapping tool. If you use the provided DAO, Hibernate will store your results in one of more than 20 different database products.

4.3.2 Differences

The second task in the general plan (see Section 3.4) is to determine the initial set of changes. The domain of the “Differences” component is the *computation and applying* of changes. This component plays an important role in several tasks of the general plan, including the determination of the initial changes.

So far, we have not defined how we will represent a change. In software engineering, appropriate techniques and programs exist to compute and to apply changes between text files (thus, source code files).

Diff

The diff program, developed in the early 1970s on the Unix operating system, can compute the differences [Wik05b]. The basic algorithm is described in [MM85] and [Mye86]. The most common implementation of the diff program is probably the “GNU diff” tool [EHH$^+$05].

Contrary to what you would expect, it was problematic to find an appropriate implementation in the Java programming language. There is a translation of the GNU diff algorithm to a Java class [Gat05]. That class would enable DDCHANGE to compute the required differences. Because that port is based on GNU diff, which is licensed under the GPL[6], the usage of the port (DDCHANGE would be compiled against it) would force DDCHANGE to be released under the GPL, too. The license of our choice is the Eclipse Public License, a less restrictive one; thus, using that port is no option.

The first approach to solve this problem was to call an installed version of the GNU diff tool. Calling the diff command line tool has two main disadvantages. At first, that method is quite slow, Java has to spawn a process, and the strings to be compared have to be written previously to disk. Second, the command line tool has to be installed on the computer DDCHANGE runs on. That tool is installed on almost all Linux machines, whereas it is not on Windows machines. On the last named, you have to install a GnuWin32[7] package. That solution does not fulfil the out-of-the-box requirement.

One library used by the "Subclipse" component provides an Java implementation (not ported from GNU diff) of the diff algorithm. Fortunately, that implementation can be used by the "Differences" component. Thus, the component works without the need to install additional software.

You could ask why I have not implemented the diff algorithm by myself. The GNU diff algorithm is an optimised version of the original algorithm. If you try to optimise the original, you will be nearly unavoidably close to that implementation, violating the GPL. Finally, using an existing implementation saves a lot of time.

Currently, DDCHANGE is able to create *unified diffs*[8] using either the command line tool, or the Java implementation. That unified diff represents a difference between two arrays of strings, for example, lines of a source code file. By default, the Java implementation is used. Other implementations are possible; you should use the interface `IDiffCreator` provided by DD-CHANGE.

Patch

If you want to apply a (unified) diff, you probably use the patch program. Larry Wall (most widely known for his creation of the Perl programming language) wrote that program. Now, it is part of the GNU project [WE05]. Applying a patch changes the text (file) according to the instructions contained in the diff.

The Eclipse Project (see Section 5.2.1) contains classes that implement the patch algorithm. Because these classes were linked heavily with Eclipse and we want to use them in DDCHANGE, independent of Eclipse, it was necessary to uncouple them from Eclipse. Now, the affected

[6]The GNU General Public License (GNU GPL or simply GPL) is a free software license, originally written by Richard Stallman for the GNU project.
[7]GnuWin32 provides Win32 ports of tools with a GNU license.
[8]The "unified" format is one of the different formats provided by the diff tool. This format is often used as input to the patch program.

classes are part of DDCHANGE. During this work, I found two bugs[9] in Eclipse's implementation; I have fixed them and submitted the patches to Eclipse's bugtracker [Fou05b].

The interface `IPatch` represents a textual difference between two arrays of strings ("original" and "target"), or in short diff. In addition to a diff, an `IPatch` can be applied using the provided implementation of the patch. Thus, instances of that interface are the conjunction of a diff with the patch utility.

You can use the utility class `DiffUtilities` to obtain instances of that interface. Given two strings (or arrays of strings), the utility class constructs instances that can be applied to a string (or array of strings) in order to change that string according to the internally created unified diff. If you have obtained already a unified diff, the utility class can create an instance by passing that given unified diff.

Hunks

When comparing two sequences of lines (for example files or arrays of strings), diff finds subsequences of lines common to both sequences. The subsequences are disjoined by groups of differing lines called *hunks*. Comparing two identical sequences, diff computes one sole (sub) sequence of common lines and no hunks—no lines differ. If the two sequences are entirely different, the diff algorithm results in one large hunk and no common lines. In general, there are many ways to match up lines between two given sequences. The algorithm tries to minimise the total hunk size by finding large sequences of common lines separated by small hunks of differing lines.

As discussed in Section 3.3, the decomposition of the changes is important in order to get a small failure-inducing difference. Obviously, the decomposition of a diff into its hunks is one way to break a diff that affects a whole file into smaller parts; usually a hunk affects a few lines.

The `DiffUtilities` class can compute one (possible) large diff between given strings. Furthermore, it can break the internal diff into its hunks. Using the related method, the utility class returns an array of instances of `IPatch`, whereas every individual instance represents one single hunk—small patches.

Reverting

The provided patches by DDCHANGE can be undone. Thus, after a change was applied, it can be reverted. That allows to switching between the program versions while the algorithm is running. Without that operation, DDCHANGE would have to obtain the unchanged version of the program repeatedly after a test was run. In the case of the version control system, that operation would be much slower than reverting the changes.

[9]See bugs with ID 93810 and 93901 in Eclipse's Bugzilla bug database.

Using the utility class `DiffUtilities` you can create small patches and apply them to strings or arrays of strings. A utility class reads the content of a file into an array of strings. Thus, you can create patches that change the content of a file according to hunks. These patches can represent the changes between the passing and the failing program version. Using the delta debugging algorithm and these patches, DDCHANGE determines the failure-inducing changes.

4.3.3 Subversion

We have seen the way DDCHANGE stores the chronological information about the run tests (see Section 4.3.1). DDCHANGE can create and apply changes between program versions (see Section 4.3.2). However, how can we obtain the initial set of changes from the revision control system? The “Subversion” component manages to do exactly that. Its domain is to create diffs between two versions of a program in the Subversion version control system [dev05b]. The utility class `DiffUtilities` can create patches using that given diff.

In order to access the working copy[10] and the repository[11], DDCHANGE can use different client adapters. Two are supported out-of-the-box.

SvnClientAdapter

Subversion was designed from the start to be an API, in contrast to CVS[12]. It is written in C as a set of libraries. The command line tool `svn` is the default UI of Subversion that uses these libraries. In addition, Subversion provides language bindings for various programming languages. Therefore, the native C libraries can be used in other languages. The Subversion project provides a Java language binding named JavaHL [dev05a], it implements the API via a thin layer that uses JNI[13].

The *svnClientAdapter* [Céd05] is a Java project that was developed for Subclipse, a user interface to Subversion from within the Eclipse IDE. SvnClientAdapter is a higher level API that uses JavaHL; this client adapter is easier to use than JavaHL in many cases.

[10] A Subversion working copy is an ordinary directory tree on your local system, containing a collection of files. That collection is under control of Subversion.

[11] A Subversion repository is the central store of data. The repository stores information about all changes on a central server.

[12] CVS (Concurrent Versions System) is another version control system. A number of key developers who have worked on CVS are now responsible for Subversion, which aims to replace CVS by addressing some of its limitations.

[13] The JNI (Java Native Interface) is a framework with a standardised API that allows Java to call native applications and libraries written in other languages, such as C, C++, and other. For instance, a program running in the Java virtual machine can use operations defined in a DLL (Windows Dynamic Link Library) or in a shared library on Linux.

If the JavaHL library is available on your system, or easily attainable, then it is probably the best choice. It supports all features of Subversion because it uses the native libraries. At the same time, that is the main drawback; if there are no native libraries on your system, you will be lost. For instance, a Windows user has to install several non-documented DLLs in the system's path. On a Linux system, you will have to compile these libraries by yourself, if there is no precompiled package available. Thus, using only that client adapter does not fulfil the out-of-the-box requirement.

JavaSVN

In contrast to the JavaHL bindings described above, *JavaSVN* is a pure Java Subversion client library [Sof05]. Thus, it does not need any additional configuration or native binaries to work on any OS that runs Java. However, there are some issues of this adapter:

- It cannot access subversion servers 1.0.*, the server has to be at least version 1.1.*.
- It does not support file:// URL's.
- It uses pure java JSCH library to establish SSH connection. This library supports only SSH version 2, with password or private key authentication.
- It uses SSL support included into JDK. Some JDK versions do not support SSL server certificates longer than 1024 bytes or do not support certain Cypher Suites.

If these limitations do not apply to your application or system, you should use the JavaSVN client adapter. This client adapter was added later in the development of the framework. At the beginning of this work, there was no stable version of JavaSVN available. Now, there is a stable, public version and that version is used by DDCHANGE.

Autoconfiguration

If you do not configure DDCHANGE explicitly to use one of the two client adapters, DDCHANGE automatically uses one that is available. At first, it tries to initialise the svnClientAdapter. If that attempt fails (for example because the required native libraries are not available), the JavaSVN client adapter will be used.

The class `SubversionToolFactory` can be used to produce instances of `ISubversionTool`, a toolset for Subversion. Other than that mentioned operation to create diffs, the toolset provides methods to undo local edits in the working copy (revert), to checkout a working copy from a repository, and more. The Factory uses the mechanism described above to determine the best client adapter that is used by the returned instance of the toolset.

Because the JavaSVN client adapter does not support all features, your application will be warned if the svnClientAdapter is not available. However, this method ensures that most of the possible repositories are supported.

Other Version Control Systems

At present, DDCHANGE supports directly only Subversion as version control system. Other systems, such as CVS and Perforce, are imaginable. DDCHANGE does not provide an abstraction of accessing a version control system. However, as we will see in Section 5.2, using CVS to obtain the initial set of changes does not require much cost. Furthermore, the Eclipse plug-in uses another repository to get these changes. Only a few classes were needed to accomplish this task.

Nevertheless, providing an interface that specifies the access would be enhance the reusability and localise the impact of changes. That one issue could be tackled on further work.

Subversion was chosen because of several reasons. First, WEB.DE uses that system. Obviously, DDCHANGE, or at least the tools, have to support that system. Second, Subversion aims to replace CVS, a widely used version control system that has become popular in the open-source world. It begins to show that new open-source projects use Subversion instead of CVS, and newer projects tend to use unit tests rather than older projects. Thus, Subversion is a good choice regarding current and future projects.

The Subversion component provides all methods in order to compute the initial set of changes between the failing and the working program version. In conjunction with the Diff component, DDCHANGE is able to compute the initial set of changes. These changes can be applied and reverted, thus the delta debugging algorithm can test different configurations.

4.3.4 Test

As seen before, DDCHANGE is able to establish a history, to determine changes, to apply, and to undo current changes. One of the main tasks in the general plan (see Section 3.4) is missing: to run the unit test. Exactly this task is the domain of the “Test” component, to *run JUnit tests*. However, this component provides interfaces and classes in order to use other types of tests, too.

JUnit tests

As described in Section 2.2, unit testing is an essential part of debugging. In the Java world, JUnit is the most popular unit-testing framework (see Section 3.1). Therefore, DDCHANGE provides support for running JUnit out-of-the-box.

DDCHANGE supports running tests in a general fashion (see below), but it implements all prerequisite in order to execute JUnit tests. The class `SimpleJUnitTester` will be the first contact point if you want to understand the mechanism to run the tests. Given a description of the test to run, the tester will launch that test in the same virtual machine (VM). The aspect mentioned at last is the main drawback of this tester. If you run the JUnit test in the same VM and this test crashes the VM, your complete application will crash, too. Especially, if the crash of the VM *is* the failure, DDCHANGE would not be able to debug that failure. Therefore, DDCHANGE is able to run tests in a separate VM.

Remote tests

The class `RMIJUnitTester` is the actual tester used by DDCHANGE (that uses again internally the simple tester). That tester can be contacted via *RMI*[14]; you can call a remote method that runs the test specified by the given description. Thus, tests can be executed in a separate VM—DDCHANGE will execute its JUnit tests in the separate VM communicating via RMI. The main intention is to run the tests in a separate VM. However, that mechanism enables you to run tests even on another machine.

RMI Registry Remote objects can be obtained from remote objects already known. The first contact requires an URL that contains the address of the server and the name of the requested object (in terms of RMI: service name). If you have the URL of the desired object, you can request the actual reference to the remote object from the *RMI registry*, a naming service that associates readable names with remote objects. The remote object has to be registered with the naming server.

Obviously, the RMI registry has to be started before the JUnit test can be run. Generally, the rmiregistry command creates and starts a remote object registry on the specified port on the current host. To fulfil the out-of-the-box requirement, the application should not be responsible for launching that naming service. In fact, the remote testing procedure should be as far as possible transparent to the application. Therefore, DDCHANGE is able to launch and stop the registry programmatically. The Factory `RegistryManagerFactory` returns an instance of `IRegistryManager`, a service that is able to start and stop the registry. Currently, a manager starting and stopping the registry in a separate thread is available.

Launching the tester In order to launch the tester, you have to start at first the registry. As described above, DDCHANGE provides the required operations. After the registry has started, you can start the separate tester. The class `RMIJUnitTester` provides a `main(...)` method,

[14]The Java Remote Method Invocation API (short: RMI) is a Java application programming interface for performing remote procedural calls. Remote means the called object can be instantiated in another VM. That VM can be running on the same host or even on another machine reachable over the Internet.

thus can be started as a Java program. The main method starts several threads (one is required in order to shut down cleanly the tester, another runs the tester itself).

Java does not provide an easy way to run other (Java) programs programmatically. Thus, DDCHANGE provides the class `RMIJUnitTesterLauncher`. That class launches the remote tester in a separate VM. The running tester can be contacted via RMI.

Because the JUnit test to run may relay on system properties, the launcher writes the current properties to a temporary file; the remote tester will load the stored properties as its system properties. Thus, the JUnit test finds the same properties in the separate VM as it would find in the VM running DDCHANGE.

The remote tester requires some third-party libraries. It uses the Log4j logging service to log messages and relies on the Commons Lang component. Therefore, the launcher provides methods to add the required libraries automatically to the class path.

Reloading Class Loader

The default class loader[15] loads the requested class only once from the file system. After the class was loaded when requested for the first time, subsequent requests will return the already loaded implementation. Normally, that improves the performance because it saves accesses to the hard disc. In our case, that is a big drawback. While applying changes, the source code of the classes may be changed. Thus, the implementation of the named class changes. If the unit test is run again, this changed implementation of the class must be loaded.

One possibility to resolve that problem is to restart the remote tester between the individual test runs. Though, launching of a separate VM is time-consuming, the runtime of the algorithm would jump up. Therefore, a better solution is to implement a class loader that reloads the classes every time.

DDCHANGE uses an own class loader to accomplish this task. The class loader `ReloadingClassLoader` reloads classes at every request. Because some classes cannot be loaded by a custom class loader for security reasons (for instance, the classes in the java.lang package), the reloading loader uses a given *fallback class loader* to load these restricted classes. By default, it uses the actual instance of `sun.misc.Launcher$AppClassLoader` (the class loader that typically loads the application) as fallback. That issue is the reason for running the tester in a separate thread in the separate VM. The remote tester sets the context class loader of the thread running the tester. Thus, the reloading class loader loads all classes in the context of the run JUnit test.

A very interesting and enlightening article about class loaders is [Gül05b]. It describes exactly the problems (and causes) I run into while implementing the remote tester.

[15]The class loader concept is one of the cornerstones of the Java virtual machine. It describes the behaviour of converting a named class into the bits responsible for implementing that class.

Currently, the reloading class loader reloads the requested classes every time, for every request. A better solution would be some kind of a flag that indicates the need to reload the classes once. Thus, if a class is requested for the first time, its actual implementation will be loaded. Subsequent requests will result in returning the already loaded implementation, until that flag is set again. Obviously, that flag would be set before an individual test will be run.

Other Tests

Unit tests are one possible type of tests to detect failures; other types are imaginable and existing. Therefore, DDCHANGE provides classes and interfaces that specify the description of a test to be run and the tester that runs the described test. Given a description, the tester will execute the appropriate test and will finally return the test outcome. Figure 4.4 shows an UML class diagram of these classes and interfaces.

As we will see in Section 4.3.5, DDCHANGE uses the interfaces and not the concrete implementations to run the tests. Thus, you can implement your own concrete test and DDCHANGE will use your test while running the delta debugging algorithm.

DDCHANGE is able to run a test that decides whether a set of changes is relevant or not. It provides an implementation that runs JUnit tests; other types of tests are possible.

The JUnit tests are run in a separate VM to ensure stability while the tests are run. The reloading class loader enables DDCHANGE to change the implementation of the classes while the tester is running. All these features are usable out-of-the-box, thus DDCHANGE allows using unit tests without larger costs. As we will see in Section 5.1, instances of the framework can use JUnit tests without the need to implement a single class.

4.3.5 Debugger

In order to complete the tasks required by the general plan (see Section 3.4), we need to reconstruct the program and to call the delta debugging algorithm. Furthermore, the Diff and the Subversion components provide utilities to create changes, but they are not responsible for the creation of the initial set of changes.

All these tasks compose the domain of the "Debugger" component, the last component not described, so far.

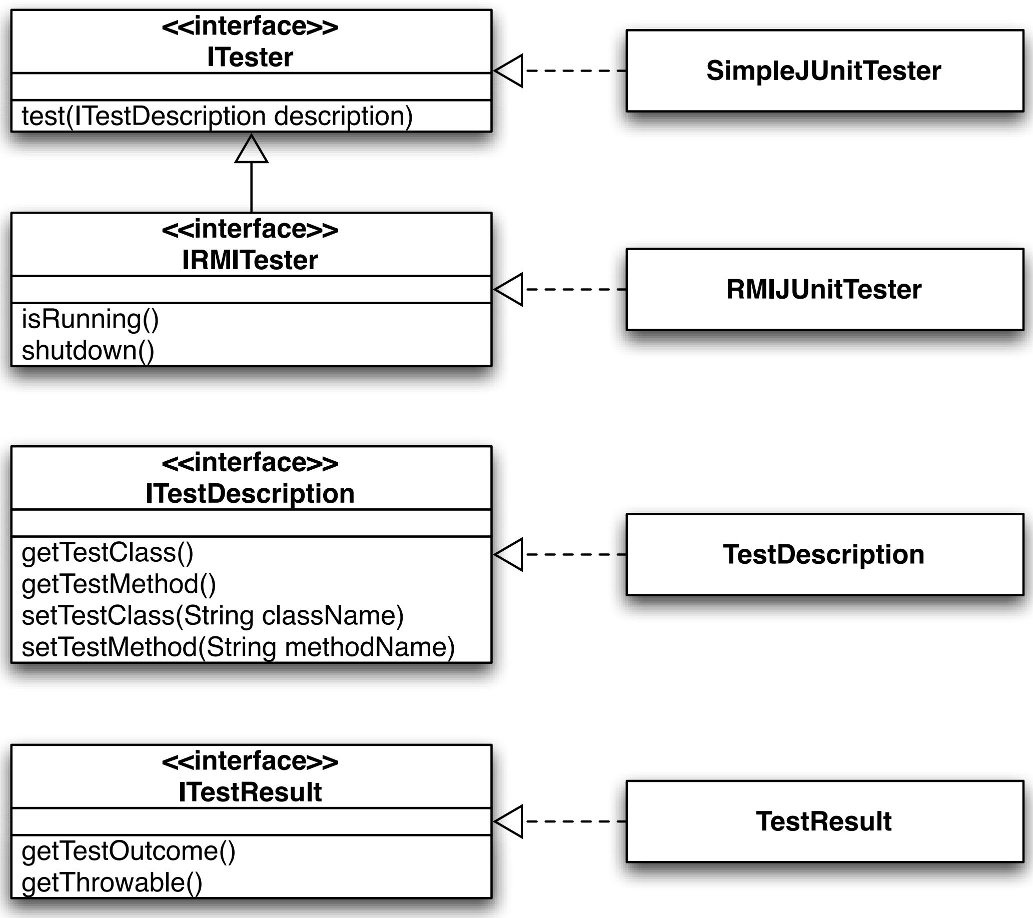


Figure 4.4: UML class diagram of the classes and interfaces responsible for conducting tests. DDCHANGE is capable of running JUnit tests in a separate VM. The diagram shows only the most important methods.

Reconstruction

After all changes have been applied successfully, we have to reconstruct the program. DDCHANGE provides the generic interface `IBuilder`, its only method builds something. For instance, a compiler would compile Java source files if that method is called. DDCHANGE has three implementing classes. The `NullBuilder` will be useful, if DDCHANGE should not build by itself (the debugger requires an instance of that interface, see below). The `MultiBuilder` uses multiple builders when it builds; it calls all given builders in the order of their addition. Because DDCHANGE uses exactly one builder to reconstruct the program, you will have to use that builder, if you need to reconstruct in several steps (for example to create J2EE home and remote interfaces, and to compile all the source code afterwards).

Ant Compiler The most interesting class implementing the interface may be the `AntCompiler`, a builder that compiles Java sources. For these purposes, that builder uses the Ant Javac task [dt05b] programmatically. The compiler is simple to use and powerful at the same time. The initialisation requires two steps, you have to set the source folder, the location of the java files, and the destination folder, the location to store the class files. If the build method is called, it compiles the specified java files using the default configuration values of the Javac task. However, you can set all the optional parameters of the Javac task by calling the appropriate method of the compiler. See [dt05b] for more information about the possible parameters to configure the compiler.

Scrub or Depend DDCHANGE provides two modes that specify which classes will be reconstructed. The first mode called *scrub* removes all class files (it “scrubs” the destination folder); thus, the compiler will compile all Java files. That mode is something like brute force, even if only one Java file out of several dozens has changed, all files will we compiled.

Therefore, DDCHANGE provides a second mode, called *depend*. That mode uses the Ant Depend task [dt05a]. It determines the class dependencies in order to find out which classes have to be compiled. The task does not parse the source files; it uses the class references encoded into the class files by the compiler. That method is generally faster than parsing the sources. The dependency tree is cached, the task re-analyses only the classes that have changed. DDCHANGE uses Depend to detect direct class-class relationships. The task can determine transitive, indirect relationships. However, considering transitive relationships will often result in recompiling all the classes. In that case, the depend mode is much slower than the scrub mode. The performance dependents on factors such as class relationship complexity and how many class files have changed. Thus, using the scrub method can be faster than the depend mode. Furthermore, the Depend task has some limitations; it cannot detect all types of dependencies. For instance, the Java compiler may optimise away some class relationships. See [dt05a] for more information about the advantages and the drawbacks of using that task.

Because of the performance issue and the limitations, DDCHANGE uses the scrub mode as default. Obviously, that mode has not any problem with dependencies or inner classes. While testing DDCHANGE, the simple scrub mode was often faster than the depend mode; the computation of the dependency information and the determination of the relationships may be to slow. However, that depends on the actual project[16].

The Deltas

DDCHANGE uses an abstraction of the changes, called deltas (because of the name of the algorithm). The `IDelta` interface specifies such a delta. It provides methods to apply and to undo the delta. DDCHANGE uses only instances of that interface when calling the algorithm. Thus, any class that implements that interface can be used in order to debug automatically some deltas.

Changed Files In order to describe deltas on files (most notably changes), DDCHANGE uses the subinterface `IFileDelta`. Instances of that interface represent a delta that affects a file. There are several implementing classes.

- `FileDeltaAdded`. A delta that represents a file that will be added when this delta is applied and that will be removed again if this delta is undone afterwards.
- `FileDeltaChanged`. A delta that represents a file that will be patched when this delta is applied and that will be un-patched again if this delta is undone afterwards.
- `FileDeltaRemoved`. A delta that represents a file that will be removed when this delta is applied and that will be restored again if this delta is undone afterwards.
- `FileDeltaReplaced`. A delta that represents a file that will be replaced when this delta is applied and that will be restored again if this delta is undone afterwards. That delta is useful for binary files that cannot be patched.

Using these deltas, you can represent all possible changes of a file. Figure 4.5 shows an UML class diagram of the interfaces and classes used to represent deltas and changes, respectively.

Collecting and Creating Deltas

Until now, we have seen the different utilities available to create changes between files and between program versions in a version control system. DDCHANGE provides classes that help to collect and to create changes.

[16]And remember Occam's Razor: "Keep it simple".

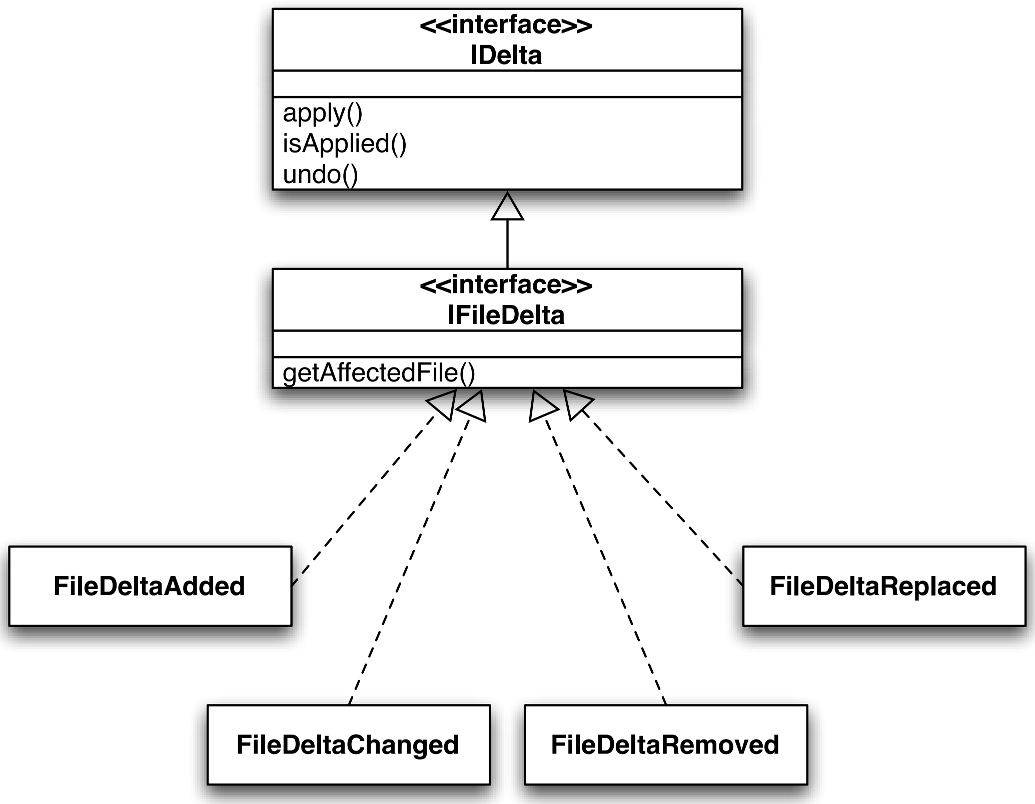


Figure 4.5: UML class diagram of deltas and changes on files. The diagram shows only the most important methods.

The most abstract interface in that context is `IDeltaCollector`. Implementations of that interface start some process. After the process has finished, the collected deltas can be obtained from the collector. The abstract class `AbstractDeltaCollector` implements that interface. It provides methods to add collected deltas and to obtain them later.

During the process of collecting deltas, the deltas have to be created in some way. The interface `IFileDeltaCreator` specifies a class that creates an array of deltas for a given file. For instance, while collecting deltas, that creator is used for considering files—it will create changes.

DDCHANGE provides an implementation of the creator interface, the `SubversionDeltaCreator`. That class creates deltas between a given date in the repository and a file in the working copy (wc). If that delta is applied, it will patch a file with the content of the revision in the repository to the content of the file in the wc. If the diff between the revision in the repository and the current wc contains more than one hunk, this creator creates as many deltas as there are hunks. So, every hunk is transformed into one individual patch. Using that creator, you can create changes between different versions of a file.

How can we create changes between program versions?

Visiting File System Trees Normally, the source code of a program is stored in a file system tree. Thus, descending that tree is an applicable method in order to process it. While visiting the files in that tree, a creator can handle every individual file.

The *Visitor Pattern* is used often to separate the structure of an object collection (here, a tree) from the operations performed on that collection (here, creating deltas). Unfortunately, the current Java specification does not support that pattern on file system trees. Therefore, DDCHANGE provides two interfaces and a class that add the Visitor Pattern to Java `File` objects. The interface `IFileVisitable` specifies the pattern on `File` objects, the interface `IFileVisitor` should be implemented by classes that visit `IFileVisitable` trees. The class `FileVisitable` implements the `IFileVisitable` interface to add the pattern to file system trees.

The class `FileDeltaCollector`, extending `AbstractDeltaCollector` and implementing `IFileVisitor`, is the connection between the process of collecting changes and their creation. That collector processes file hierarchies creating deltas using one or more `IFileDeltaCreator`. At construction time, you specify the root node of the hierarchy as `File` object. Starting at that node, it descends the file hierarchy. For every node in that hierarchy, it calls the `createDeltas(File)` method of the attached creators.

Figure 4.6 demonstrates the usage of that collector. You can use an instance of `SubversionDeltaCreator` as the creator.

Running the Delta Debugging Algorithm

Now, we have all prerequisites to determine the failure-inducing changes. Using the `SubversionDeltaCreator` and the `FileDeltaCollector` we can compute the initial set of changes. We can reconstruct the code; the changes can be applied to create different configurations. Finally, we can run JUnit tests.

Tip of the Iceberg DDCHANGE hides completely the call of the delta debugging algorithm. You have to create an instance of the class `DeltaDebugger`, passing a builder, a tester, and a description of the test. Given the set of initial changes, the debugger will determine the failure-inducing. Figure 4.7 shows briefly how to use the debugger.

Listening to the Algorithm If you want to observe the debugger and the algorithm during the debugging process, you can use an implementation of the interface `IDeltaDebuggingListener`, a listener interface for observing the execution of the delta debugging algorithm.

The debugger will notify every attached listener whenever a test starts, ends, and so on. If you want to inform the user of your application, an instance of the framework, about the progress of the determination, this mechanism is probably the only choice. However, listeners are notified only about the events of individual test runs. Because we do not know how many test runs are required before having the result, DDCHANGE is not able to give a prediction about the runtime of the whole process.

Compile Flow Statistics on the Algorithm DDCHANGE allows analysing the complete process and the individual run tests by providing a flow statistic. Using the listener concept as described above, DDCHANGE collects information about the conducted tests. That information contains the runtime of the tests, their outcome, the number of deltas, and more. After the determination, the debugger can return an instance of `IDeltaDebuggingStatistic`, a statistic of a single delta debugging run. This statistic can return an array of instances of `ISingleTestStatistic`, the statistic for a single test run. The elements in the array are sorted by the order of the run tests.

To ease the further processing of the statistic, both interfaces specify a method that returns an XML element representing the statistic. For instance, you can easily write the statistic in the XML format to your hard disc. One of the tools uses those methods to create an HTML report that shows the final report containing the failure-inducing changes (and the statistic).

```java
class DeltaCreator implements IFileDeltaCreator {

    public IFileDelta[] createDeltas(File file) {
        // your code here return new IFileDelta[0];
    }

}

// the root node for collecting deltas in file system
File root = ...;

// collect deltas in file system
FileDeltaCollector collector = new FileDeltaCollector(root);
collector.addCreator(new DeltaCreator());
collector.collect();

// obtain collected deltas
IDelta[] collectedDeltas = collector.getCollectedDeltas();
```

Figure 4.6: Listing: How to collect changes.

```java
// reconstructs changed code
IBuilder builder = ...;

// runs the test, for instance unit test
ITester tester = ...;

// specifies the test to run
ITestDescription description = ...;

// the initial set of changes
IDelta[] allChanges = ...;

// constructs new debugger
DeltaDebugger debugger =
    DeltaDebugger(builder, tester, description);

// determines failure-inducing changes
IDelta[] failureInducingChanges = debugger.debug(allChanges);
```

Figure 4.7: Listing: Tip of the Iceberg. How to determine failure-inducing changes.

As we have seen, using the Debugger component along with the other, we can accomplish the general plan. Thus, DDCHANGE enables us to determine the failure-inducing changes. The component described above provides methods to reconstruct the program, to determine the initial set of changes by visiting a file system tree, and to call the delta debugging algorithm via a well-defined abstraction, the debugger. That class of the framework is *the tip of the iceberg*, it pulls the strings. In addition, the debugger allows to observe the process and to compile some statistics about the process.

4.4 Ensure Quality

As seen before, DDCHANGE can accomplish the tasks defined in the general plan. Because instances of the framework depend on the reliability of DDCHANGE, the framework should assure some *quality management policy*. However, quality in computer software is a controversial subject. For some people, quality is a practical or even aesthetic issue, for example the programming style. For other people, quality is defined as strict compliance to requirements and non-appearance of bugs [Wik05e]. An advanced discussion of that topic goes beyond the scope of this work. Therefore, in the following we briefly discuss some *indications* for quality.

- *Using Maven*. The DDCHANGE projects are managed by Maven, a software project management and comprehension tool. This ensures a coherent project documentation and a well-defined release process. Maven runs all unit tests before a new version is released. Thus, versions that do not pass all the unit tests are prevented. See Section 5.1.1 for a more detailed description of Maven. The included CD (see Appendix A.1) contains the complete documentation generated by Maven.
- *Unit Tests*. Creating a test case is the first step before a program can be debugged (see Section 2.2). DDCHANGE uses JUnit tests to ease the debugging of failures.[17] For instance, when you change the source code adding new features, you will probably be warned if you introduced a failure in the existing code. The unit tests and the high test coverage (see next item) help to avoid regressions while changing and extending the current implementation.
- *Code Coverage*. Code coverage is a metrics that reflects the degree to which the source code of a program has been tested. DDCHANGE uses Clover [Ltd05a], a code coverage analysis tool, to discover sections of code that are not being adequately tested (using unit tests). Figure 4.8 shows the coverage for conditionals, statements, methods, and the total percentage coverage[18].

[17] There are currently 396 unit tests. Four of them are failing because of bugs in third party libraries. Thus, the failing tests do not show failures of the framework, but of the used libraries. See DDCHANGE's Bugzilla bug database for more information about these bugs.

[18] The Total Percentage Coverage (TPC) is calculated using the formula: $TPC = (CT + CF + SC + MC)/(2*C + S + M)$ where CT - conditionals that evaluated to "true" at least once, CF - conditionals that evaluated to "false" at least once, SC - statements covered, MC - methods entered, C - total number of conditionals, S - total number of statements, and M - total number of methods.

Based on the assumption that the unit tests are implemented correctly, the total percentage coverage of 84% indicates an almost faultless implementation of the framework. Achieving a coverage of more than 90%—or actually achieving 100%—is very hard and time-consuming. If you look at the Clover coverage report (see Appendix A.1), you will see that most of the uncovered code sections are `catch` blocks of error-handling code. Writing tests covering that type of code is particularly costly[19].

Clover is run by Maven. Thus, it is transparent to the developer; she does not have to run that tool explicitly. The following two tools are also run by Maven while generating the project documentation.

- *Checkstyle*. Checkstyle [Tea05a] implements a static code analysis in order to help programmers write Java code that adheres to a defined programming style. Code conventions improve readability of the software, allowing you to understand the code more quickly and thoroughly.
- *Findbugs*. FindBugs [Hov05] is a tool to find bugs in Java programs. It looks for instances of so-called *bug patterns*—code instances that are likely to be errors. Using FindBugs helps to avoid code idioms that are often an error. For example, if you forget to close a database connection on all exception paths out of a method, FindBugs will warn you. Failure to close database resources on all paths may result in poor performance.
- *JavaDoc*. The API documentation of DDCHANGE is nearly complete; almost every public, protected, and private package, class, interface, field, and method is documented. The documentation contributes to a developer's understanding and helps a developer write reliable tools based on DDCHANGE more quickly.
- *Logging*. DDCHANGE uses log4j [Gül05a], a popular logging framework for the Java programming language. The log messages can help you to observe the program run in order to gather facts about concrete runs of the framework, so that a problem can be located easier and faster.
- *Subversion*. A Subversion repository stores all the source files of the different DDCHANGE projects. The repository remembers all individual changes ever written to it. This allows you for recovering older versions of the source code, or for examining the history of how it changed. In this regard, you may think of a version control system as a sort of *time machine*. The repository is able to answer questions like "What was the content if that file last Wednesday, and who changed it since the last release?".

In conjunction with the unit tests, the tools of the platform can be used to debug failure-inducing changes on the projects itself. Appendix A.2 describes all projects stored in the repository.

[19]One technique to test such code sections is to use *Mock Objects*—a "double agent" used to replay the behaviour of objects. Some tests of DDCHANGE (for example, the unit tests for the Database component) use Mock Objects to increase the code coverage.

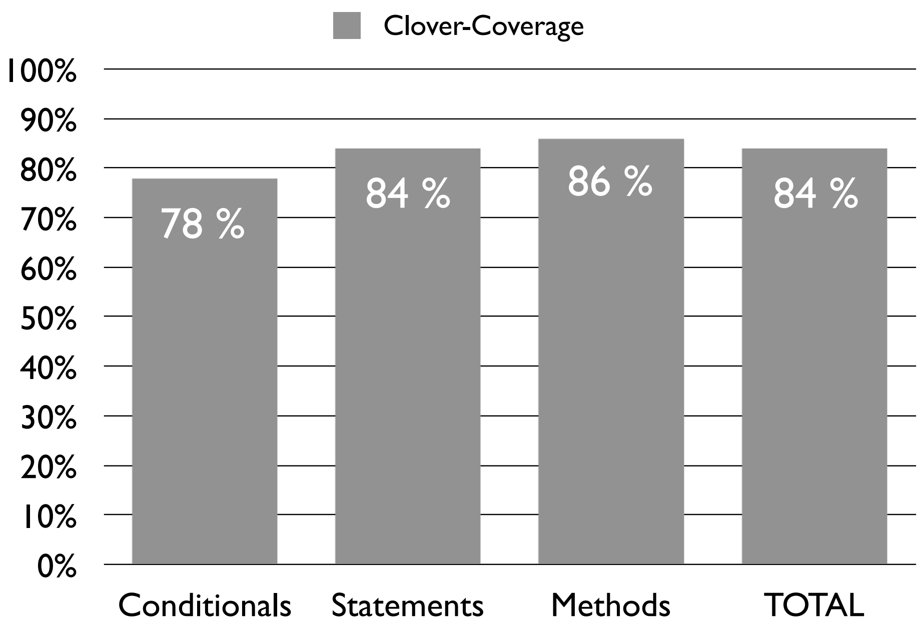


Figure 4.8: Bar chart: Code coverage of DDCHANGE as computed by Clover. Note: Coverage measurement is not a replacement for good code review and good programming practices.

- *Bugtracker*. The Bugzilla bug-tracking tool is used to collect, manage, and document issues of the DDCHANGE projects. The public tracker allows developers at WEB.DE—as well as all other users—for submitting failures, feature requests, and other tasks to a central system. Bugzilla defines a *Bug's Life Cycle*, that life cycle provides a well defined workflow how to handle all requests. Using a bug tracker avoids that submitted issues are forgotten, and improves communication between developers of DDCHANGE and its users, just as among the developers.

No items prove the quality of DDCHANGE. However, they are an indication of the project's grade.

4.5 Extending the Framework

As described in Section 4.1, one of the main intentions of using a framework is to enhance reusability. Thus, using DDCHANGE should leverage the domain knowledge and should improve the programmer's productivity. In the following sections, we will see how we can extend the framework. You will not see any example code. However, we will sketch all required contributions out.

4.5.1 Other Version Control Systems

Currently, DDCHANGE supports the Subversion version control system as a source of changes. Because many existing and especially long-standing projects use CVS instead of Subversion, adding support for CVS could be a worthwhile contribution. DDCHANGE provides some classes that ease the collection and creation of changes that can be applied in order to construct different program versions (see Section 4.3.5).

The main class responsible for the creation of the changes is the `SubversionDeltaCreator`. Thus, implementing an analogue class that uses CVS instead of Subversion should be adequate. In fact, you have to provide an implementation of `IFileDeltaCreator`, for example named `CVSDeltaCreator`, that creates the differences between two versions (or, dates). Using the provided classes `DiffUtilities` (see Section 4.3.2) and `FileDeltaChanged` (see Section 4.3.5), that task can be done without any changes on the framework. That is exactly how the Subversion related implementation creates the changes: The diff utility class creates a patch from a computed unified diff, the delta is constructed by passing that patch and a reference to the affected file.

There is a Java based CVS Client, called jCVS [End05]. Using that client library and the classes provided by DDCHANGE that task should be realisable in about one day. Afterwards, DDCHANGE would be able to create the initial set of changes using a CVS repository.

4.5.2 Other Tests

The general plan (see Section 3.4) uses unit tests in order to determine the failure-inducing changes; the non-working unit test is the failure. Other testing techniques may be used to observe other types of failures that can be debugged with the delta debugging algorithm.

Memory Leaks

Typical heap management problems in C/C++ are *memory leaks*[20] and *dangling pointers*[21]. Because of the built-in garbage collection (a form of automatic memory management that reclaims the memory used by objects that will never be accessed again) dangling pointers are history in the Java programming language. Unfortunately, memory leaks can still occur in Java programs, and this type of failure is hard to debug [Pat01]. The developer is still responsible for cleaning up references after use. For instance, adding objects to a vector and forgetting their index may

[20] A memory leak will occur if an allocated part of the memory is not freed after that part is not used anymore. Memory leaks can cause to stop the entire system from working correctly.

[21] In C and many other languages, removing an object from memory does not alter pointers that point to that object; the pointer will point to the location in memory even though the memory now is used for other purposes. Using such a *dangling pointer* and assuming it is still valid can cause unpredictable behaviour.

result in an `OutOfMemoryException`. If such a failure occurs after you have changed your program, DDCHANGE may help you.

One possibility to detect a memory leak is to write an unit test that fails throwing the exception mentioned before. However, writing that unit test may be not sufficient. Memory leaks could slowly fill the memory; your program gets slower by-and-by. To increase the intricacy of that issue, imagine the failure will occur only if you use your application in conjunction with another product. For instance, your web application gets slower using a particular web container. In that case, you have to implement a builder (see Section 4.3.5) that deploys the application to the web server (after compiling the changed sources) during the reconstruction of the program. Furthermore, you will implement a unit test that fails whenever a request to the web application overruns its time.

Compared to adding the access to CVS, this task may take a long time. However, think about debugging such a failure by hand. You would have to bring in different changes, afterwards you will deploy the application, and you will use your browser and a stopwatch in order to determine time-outs—a tedious and costly task.

Acceptance Tests

In Section 4.3.4, we discussed the way DDCHANGE uses JUnit tests in order to reproduce a failure, in the form of a failed unit test. Other tests are possible, for instance, so-called acceptance tests. These tests are created from user stories (a description written by the customers as things that the system needs to do for them); an acceptance test represents some result expected by the user from the system. Thus, these tests verify application functionality and user acceptance. They are also used as regression tests before a production release.

One tool used for creating acceptance tests is *Selenium* [HGW05], a test tool for web applications. Selenium tests run directly in a browser, just as real users do. It deploys automatically its “Browser Bot”, the JavaScript automation engine, to a browser when you point it at the installation on your web server. This way, the web browser is controlled by Selenium doing the steps that are defined in a user story. If the web application satisfies the requirements defined by the user (or the customer), the test passes. Otherwise, it will fail.

Adding acceptance tests to DDCHANGE requires more effort than the other tasks seen before. You have to implement a builder that deploys your web application to the server that runs Silenium and a tester that gets its result from Silenium. However, using the hot spots provided by DDCHANGE would be practicable. Using Selenium is an improvement compared with doing acceptance tests with the help of some manually checked lists. It automates acceptance testing. In addition, DDCHANGE could help you to automate the debugging of failed acceptance tests.

DDCHANGE enables you to write tools that determine failure-inducing changes. As delineated exemplarily above, contributing new features to DDCHANGE should not be too difficult. That is not a proof for DDCHANGE's enhancement of reusability. However, it should at least bring forward new ideas as to how you could extend the framework.

In Chapter 5, we will regard two instances of the framework. The first one uses DDCHANGE out-of-the-box, the second one contributes to the framework. Furthermore, we will see that the amount of code required to implement these tools tends to be small, compared to the framework.

4.6 Optimisations

As seen in Section 3.5, there is room for different optimisations. In the context of this work, Bouillon's implementation of the delta debugging algorithm was extended with a working cache (see Section 4.2). Thus, the issue about the multi-tested configuration is solved. The one about the incremental reconstruction is solved partly by the provided Ant compiler and the depend mode (see Section 4.3.5). Further, we could use Eclipse's *batch compiler* [Fou05d], a very fast incremental compiler. That compiler is probably the fastest compiler available for the Java programming language that includes full support for the new features of J2SE 5.0. In Section 5.2, we will see an Eclipse plug-in that uses DDCHANGE in conjunction with Eclipse's compiler. Compared to the Maven plug-in (described in Section 5.1) that uses the Ant compiler, the Eclipse plug-in compiles the changed sources much faster. Thus, doing an incremental reconstruction does not pose a big challenge—at least compiling Java sources can be done very fast.

In the following sections, we will discuss possible optimisations that concern history, grouping, and primarily the restriction of the search scope. Most of the optimisations are based on other work.

4.6.1 Restrict Search Scope

One possibility to speed up the process is the restriction of the search scope. Obviously, if we can reduce the amount of changes in the initial configuration, the number of tests required by the algorithm will be smaller, too.

Continuous Testing

The programmer is responsible for the initiation of the unit tests. In large projects, the programmer does not run always all the unit tests after she made a change. Most of the projects at WEB.DE have a test coverage up to 100%, running all the unit tests is hence a time-consuming process. Even on a very fast (compared to the developer's workstation) server, running these tests lasts at least several minutes. Thus, if the programmer would run the unit tests two times per hour, he would spent nearly half of this working day for running (and waiting for) unit tests.

Continuous Testing [SE04b] uses the idle time on a developer's workstation to continuously run unit tests in the background. Regressions can be detected faster than running the unit tests by hand [SE04a]. As a consequence, the number of changes between a passing and a failing version is smaller—restricting the search scope.

An implementation of continuous testing is available as Eclipse plug-in. That plug-in could be used together with the plug-in described in this work, speeding up the process to determine failure-inducing changes.

Change Impact Analysis

A simple method to determine the initial set of changes is to compute all the changes between the passing and the failing program (unit test) version. However, that method collects all changes, even changes that do not affect the run unit test. We could distinguish between changes that affect or do not affect the run of the unit test. *Chianti* [RST+04] is such a tool that analyses the *change impact* of Java programs.

Chianti reports the change impact in terms of affected unit tests whose execution behaviour may have been mutated by the changes. Furthermore, it determines a set of affecting changes that were responsible for the test's mutated behaviour—Chianti distinguishes between changes that result and that do not result in another behaviour of the test run.

Chianti is implemented as an Eclipse plug-in. Just like the plug-in for continuous testing, we could use Chianti together with the Eclipse instance of DDCHANGE in order to analyse whether the combination of the change impact analysis with DDCHANGE could speed up the automated debugging process. Using Chianti to restrict the search scope—we can ignore changes that do not affect the unit test—could optimise the process. However, the change impact analysis seems to be a time-consuming process. Therefore, the restriction of the search scope could be connected with too high costs, annihilating the gain of time.

Change Classification

DDCHANGE currently uses only a small subset of all the information contained in the database: the date of the passing and the date of the failing run. Because DDCHANGE (or rather the two existing instances) will store all the test outcomes in the database if a complete test suite (collection of test cases) is run, we could use that information, too. For instance, if only one out of dozens tests fails and all other pass, there could be introduced "good" and "bad" changes since all the tests were passing. We could use again the change impact analysis in order to classify these changes.

In fact, there is a method called *Change Classification* [SRRT05]. That work presents several analyses that classify changes into three groups. Each group indicates the likelihood that the contained changes contributed to a test's failure; the groups are named *red*, *yellow*, and *green*. The *green* changes are unlikely to be failure-inducing, the *red* ones are likely to be failure-inducing, and the *yellow* are in-between. They use the change impact analysis in conjunction with the property of the individual tests whether their outcome has not changed, changed from passing to failing, or vice versa.

Again, that method is implemented as an Eclipse plug-in; we could study whether the comprehension of the changes' group membership can speed up the debugging process. However, it seems that the classification is a long-winded process—annihilating the gain of time.

4.6.2 Group Changes

As discussed before, restricting the search scope is one possibility to speed up the debugging process. Another method is the grouping of the changes according to their scope.

Change Impact Analysis

Basically, the change impact analysis obtains a set of interdependent atomic changes, and determines the changes' effect on the call graph of the tests. Further, the analysis determines dependencies between the atomic changes. For instance, if you add a method m in class C, the change that introduces a call of m in another class depends on the method's addition.

We could use that information in order to group related changes and to avoid unresolved test outcomes. Introducing the method call without the method's definition would result in a syntactically incorrect program—an unresolved test outcome (remember: if nearly all tests are unresolved the number of tests can be quadratic with respect to the number of changes). However, the change impact analysis may be connected with too high costs.

Another method to group changes is the analysis of the abstract syntax tree. Regarding that tree, we can group the changes according to their scope, whether they change the same package, the same class, or the same method (see Section 3.5).

4.6.3 History

The change classification uses information provided by the history of the run tests. Therefore, we could also classify that method in the group of methods regarding the history. As described in Section 3.5, another way regarding the history is the ordering of the changes according to their chronological order. We could extend DDCHANGE with operations that order the changes this way. Because of time constraints, that optimisation was not implemented in the context of this work.

4.6.4 Distributed or Parallel Debugging

The last optimisation described here is the distributed or parallel execution of the delta debugging algorithm. The algorithm splits the initial set of changes into two, four, eight, and so on subsets. Depending on the test outcome of a tested subset, the other subsets have to be tested. These tests could be conducted in parallel. The algorithm would test the subsets of the actual granularity at the same time on different processor units or even machines.

DDCHANGE provides some basic implementation required for this task. The remote tester (see Section 4.3.4) could be used in order to execute the tests on different machines. We could implement a pooled tester, similar to pooling for database connections. Different testers would be running and waiting for incoming jobs on different machines.

Some existing libraries would ease the implementation of such a mechanism. For instance, the Jakarta Commons Pool component [dt05d] provides a generic object-pooling API.

Using only remote testers would not be sufficient; the reconstruction has to be done in parallel, too. The support for remote builders is currently missing. With view on the implementation of the remote tester, remote builders should be realisable.

A detailed description of all tasks required to realise distributed debugging goes beyond the scope of this work. However, DDCHANGE provides some basic implementations that are required to enable distributed debugging.

4.7 Results

DDCHANGE, the platform's *core*, is a framework that enables you to determine failure-inducing changes. Contrary to a simple class library, DDCHANGE forces well-defined application architecture. An instance of DDCHANGE can hand most of the control flow, needed to determine failure-inducing changes, over to the framework. Thus, in addition to reusability of code, DDCHANGE supports the reusability of design—allowing programmers to concentrate on the details of their application. DDCHANGE tends to be a Black-Box-Framework; it provides interfaces for components that can be plugged into the framework. That type of framework can be easier extended than White-Box-Frameworks.

Being a framework, DDCHANGE should reduce cost and improve quality. The DDCHANGE project uses some tools and techniques that indicate a good grade of quality. As we will see in the next chapter, DDCHANGE reduces the source code required implementing a concrete instance, an application that determines failure-inducing changes.

In order to determine failure-inducing changes, we have to implement the general plan. As seen in this chapter, DDCHANGE provides components that enable you to accomplish all the different tasks defined in that plan. Moreover, DDCHANGE's implementation is flexible and expandable. For instance, you can implement your own methods to reconstruct the program and to conduct the test that detects the failure. Using the default implementation of DDCHANGE, you are able to implement a tool that determines failure-inducing changes in a Java program that is under test by unit tests—without the needs to extend the framework.

DDCHANGE can be used to debug other type of failures, too. As described in this chapter, DDCHANGE could be used in order to debug memory leaks and acceptance tests. Other types are imaginable, that chapter provides all the basic information that is required to implement an appropriate instance of the framework.

There is room for optional optimisations. You could restrict the search scope or group changes according to their scope. All these optimisations could result in a speed-up of the process. However, the accuracy of the result would not be improved-on that score, DDCHANGE is complete.

The next chapter describes two tools using the framework described afore.

5 The Tools

This chapter introduces the practical application of the framework in the form of two concrete instances, more precisely: two different *plug-ins*. These plug-ins allow the developer to determine failure-inducing changes using hers familiar tool.

The chosen tools are well known to Java developers. The first one is *Apache Maven* (in the following just Maven), a software project management and comprehension tool. WEB.DE uses Maven in order to manage the complete life cycle of Java projects, from the initialisation (for example checkout from a version control system) via the build to the point of the deployment to different servers. In combination with *CruiseControl*, you can use DDCHANGE in the continuous build process. The second tool is *Eclipse* and its highly regarded Java IDE. The Java developers of WEB.DE use this tool in order to develop J2EE applications.

Moreover, many Java developers and open-source projects use Maven or Eclipse, or even both. Thus, using these tools to realise concrete applications on top of the framework, we can reach many Java developers. Just like the framework, the plug-ins are released under the Eclipse Public License, an open-source license.

The following descriptions are written at first from the perspective of a developer that uses the plug-ins. Succeeding sections discuss the architecture of the plug-ins and the customisations of the framework. As we will see, the Maven plug-in uses DDCHANGE out-of-the-box, thus without the need to add any extension to the framework. In contrast, the Eclipse plug-in contributes a new builder to compile changed Java sources and another source to obtain the initial set of changes from. However, you do not need much effort to implement these contributions that use the framework to determine failure-inducing changes in Eclipse.

The included CD (see Appendix A.1) contains three movies showing the plug-ins in action. Therefore, you can get a picture of them without even installing. However, if you have installed Maven or Eclipse, it is very easy to install the plug-ins using the Maven repository and the Eclipse local update site, respectively. As well as the movies, the repository and the update site are located on the CD.

5.1 Maven Plug-In

The first plug-in is called DDCHANGE MAVEN, the plug-in for the tool of the same name. DDCHANGE MAVEN aims to integrate delta debugging on changes *transparently* with the software project. CruiseControl, a framework for a continuous build process, will call DDCHANGE MAVEN to start the debug process as soon as a test fails.

Before we will discuss DDCHANGE MAVEN, we describe Maven and CruiseControl at a glance.

5.1.1 About Maven

Maven is a software project management tool for the Java programming language. Based on the concept of a project object model (POM), Maven can manage the build process, as well as reporting and documentation of a project.

In the capacity of automated software build tool, Maven is comparable with the widely used Ant tool[1]. However, Maven is more than just a slightly enhanced Ant. In fact, the two tools have different objectives. Using Ant, the developer must understand how that tool applies to her development and environment. For instance, if she wants to compile some Java code, she will have to write a complex build file describing that task. With Maven, that *build process knowledge* is captured in *plug-ins*. A Maven plug-in is a small operation out of the whole process that relies on the POM. Thus, instead of writing a task with a specific set of parameters, the developer provides some information about the project in general and the plug-in responsible for the compilation uses that information; several plug-ins can share the same information.

Another difference in the objectives is the advancing of *best practices*. Ant is a very flexible tool, you break down the build process down to many targets, and you can combine them in many ways to build the project. That flexibility results in some uncontrolled growth because every individual programmer uses her own targets to accomplish the same task. Maven attempts to enforce the use of best practices by providing sensible default metadata—while it remains flexible. These defaults contain standard locations for sources, documentation, and output, a common layout for the project documentation and the possibility to fetch project dependencies (for example other JARs) from shared repositories.

Maven goes further than Ant; it encapsulates the *project knowledge* in one area, including the source code and the documentation. Maven handles more than the pure building. It handles the release process, different testing techniques, the generation of reports, and more. Using Maven's plug-in mechanism, other products such as issue trackers, application servers and source control systems can be integrated within Maven and consequently within the project's uniform process.

Plug-ins provide their functionality through so-called *goals*. They use the POM's metadata to complete their tasks. In some sense, they are comparable to pre-defined Ant targets. For instance, one common plug-in is the `jar` plug-in. Its goal `jar:deploy` publishes a library by building

[1]Ant is a software tool for automating software build processes. It is similar to `make` but is primarily intended for use with Java. It is written in the Java language and uses a file in XML format to describe the build process and its dependencies.

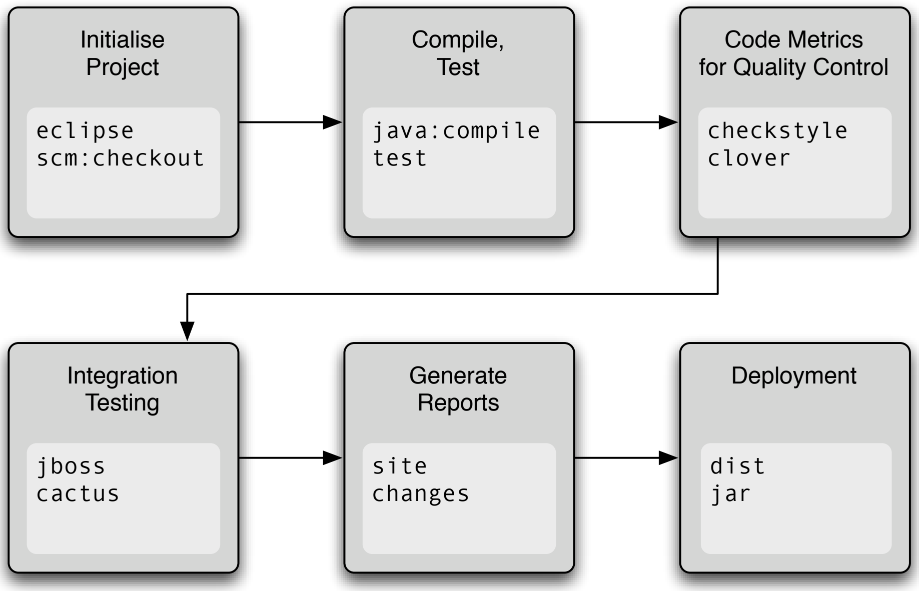


Figure 5.1: Typical lifecycle of a Maven project. That lifecycle covers the whole process from the initialisation until the deployment.

the sources and running the unit tests. After all unit tests have passed, that plug-in packages the class files into a JAR file and afterwards it deploys that JAR to a remote repository. To accomplish that task, the goal reuses the `java` and the `test` plug-ins (among others).

Figure 5.1 illustrates the typical lifecycle of a Maven project; Maven covers the whole process—from the initialisation until the deployment.

The integration of DDCHANGE within Maven would enrich the project's life cycle and its knowledge with automated debugging. The developer could continue to use Maven to manage hers process. As soon as a unit test failed (unit tests are an inherent part of a project managed by Maven), the new plug-in could start the determination of the failure-inducing changes.

5.1.2 About CruiseControl

CruiseControl is a framework for a continuous build process [FJK05]. That framework facilitates the *continuous integration* in software engineering. Continuous integration is a process that rebuilds and tests an application periodical and frequently.

Generally, every time there is a change in a revision control system or in a monitored file system, the framework obtains the most recent version of the source code. Another mode is a periodical update of the sources, for instance every night about midnight for so-called nightly builds. After the update, the framework runs the unit tests, builds the complete application, and tests the application (other types of tests are possible, for example automated acceptance tests). Because the framework integrates the software, that test is known as *integration testing*—a test to expose failures in the interfaces and in the interaction between integrated components; the so-called *big bang integration*[2] can be avoided.

By the use of continuous integration of the changes of all developers, possible failures are discovered earlier. The developers use unit tests to test small and self-contained parts of the application. That way, only failures in these isolated parts are detected early, failures as consequences of the interplay in the application are missed. The aim of integration testing is to test the application en bloc: different, interdependent components of a complex system in interaction.

Compared to unit tests, integration testing is a long-winded process. If the developer builds and tests the application after every change (even regarding not only the changes made by herself), she would not be able to develop the source code further on. Instead, after the developer has published her changes, she can concentrate on further development while CruiseControl conducts the integration test on a dedicated server in the background. Only if there is a failure, the framework interrupts the developer.

The online article of Martin Fowler and Matthew Foemmel [FF05] features a general survey on that topic. Figure 5.2 illustrates the architecture and the lifecycle of the CruiseControl framework. In principle, a developer commits her changes to a repository or a file system, CruiseControl periodically checks that place. When CruiseControls detects a new version, it builds and tests that version. The results (for instance, failed or passed build, failed or passed tests) are published in some way. The developer can obtain the published results via Email, a web page, or some other communication media.

Continuous integration and integration testing result in periodic and frequent automated tests. Currently, the developer is notified about the failure if the integration fails. We will utilise DDCHANGE in order to get the automatically determined failure-inducing changes in addition to the pure failure notification. As we will se, we can insert DDCHANGE MAVEN into CruiseControl's architecture and process, thus adding automated debugging to automated testing.

5.1.3 Usage

The first of the following sections describe exemplary the usage of DDCHANGE MAVEN. It covers the complete lifecycle, from the first passing run of the unit tests until the report generated by DDCHANGE MAVEN, an HTML page included in the project's documentation site. The different steps in the debugging process are initiated by calling certain Maven goals. The following section summarises the most interesting configuration properties as provided by DDCHANGE MAVEN. The last section outlines the integration with CruiseControl. We will describe the way that is used by CruiseControl in order to call DDCHANGE MAVEN.

[2] That term is derived from the scientific theory that the universe emerged from an enormously dense and hot state. Imagine that explosion and think about a last minute integration about to release an application.

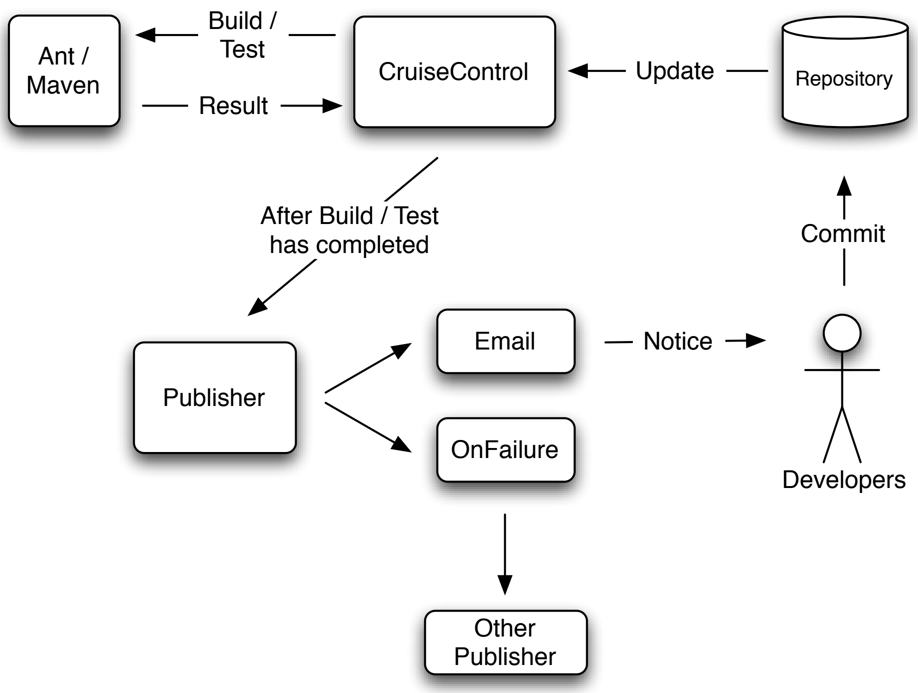


Figure 5.2: CruiseControl's architecture. A commit to the repository by a developer starts CruiseControl. After updating its sources, CruiseControl builds and tests using Ant, Maven, or other tools. The results are processed using different publishers. For instance, the developer responsible for the commit receives an email that contains a build report.

Run manually

In our example, we will use the "Apache Jakarta Commons Lang" [Tea05b] component, a well-known class library for the Java programming language. That project uses Maven; thus, we need not to write the meta data files, we can call the Maven goals without bigger effort. Further more, the project utilises JUnit in order to test the individual classes in the library. We can modify the sources and use the unit tests in order to check for introduced failures.

We will see how DDCHANGE MAVEN works based on running goals and the sample output displayed by Maven on the console while running these goals. Because Maven outputs all the run tests to the console, I removed most of the test classes for demonstration purposes. However, that does not influence the result computed by DDCHANGE MAVEN, the output is shorter. As a side effect, collecting of deltas is much faster because DDCHANGE MAVEN regards fewer class files in the file system. The modified version of the library was committed to the Subversion repository that hosts all the DDCHANGE projects. While the automatic debug process is running, DDCHANGE MAVEN will obtain the initial set of changes from this repository.

After we have checked out the sources of the Commons Lang project, we make sure that all the unit tests are passing. Thus, we run the `test` goal provided by Maven. That goal calls different other goals before the unit tests are actually run. Among other things, some directories are created. Because we have not compiled any sources so far (the class files are not contained in the repository), the called `java:compile` goal compiles the main Java classes[3], and the called `test:compile` goal compiles the test classes. Finally, the unit tests are run. As you can see in Figure 5.3, Maven runs 16 unit tests, all are passing.

Now, we know there is no failure (at least, the code covered by the run unit tests seems to be ok) in the unchanged project. In order to establish the history that contains the test results, we have to call the goal `ddchange:store-reports`. Running this goal, DDCHANGE MAVEN parses the unit test reports generated by the `test` goal. These XML reports contain all the required information about the run tests, including the test outcome and a possible stack trace. After parsing the reports, DDCHANGE MAVEN stores the results in a database. See Figure 5.4 for the complete output of Maven running `ddchange:store-reports`. Note the line saying "Using internal database.". Because we have not specified an external database, DDCHANGE MAVEN launches an instance of HSQLDB (see Section 4.3.1). The database files are located in the *target folder* of the project. That folder contains all producible files, including the class files. Because the goal `clean` removes the target directory, you should at least specify another directory for the database files.

After we have made some changes on the Java source files, we run the unit tests again in order to check whether we have introduced some failure. Figure 5.5 shows the output of the second run of the goal `test`. Now, one of the 16 unit test is failing, the `SystemUtilsTest`. We have apparently modified the source code in the wrong way—introducing a failure. Instead of

[3]Maven encourages the use of a single source directory, but a separate directory hosts by default the unit tests. That procedure eases the packaging of JAR files, which should not contain the test classes.

```
mburger@jonagold:~/ddchange> maven test
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.1-beta-1

build:start:

java:prepare-filesystem:
    [mkdir] Created dir: /Users/mburger/ddchange/target/classes

java:compile:
    [echo] Compiling to /Users/mburger/ddchange/target/classes
    [javac] Compiling 70 source files to /Users/mburger/ddchange/
target/classes
    [javac] Note: Some input files use or override a deprecated API.
    [javac] Note: Recompile with -deprecation for details.

java:jar-resources:

test:prepare-filesystem:
    [mkdir] Created dir: /Users/mburger/ddchange/target/test-classes
    [mkdir] Created dir: /Users/mburger/ddchange/target/test-reports

test:test-resources:

test:compile:
    [javac] Compiling 1 source file to /Users/mburger/ddchange/
target/test-classes
    [javac] Note: /Users/mburger/ddchange/src/test/org/apache/
commons/lang/SystemUtilsTest.java uses or overrides a deprecated API.
    [javac] Note: Recompile with -deprecation for details.

test:test:
    [junit] Running org.apache.commons.lang.SystemUtilsTest
    [junit] Tests run: 16, Failures: 0, Errors: 0, Time elapsed:
0,885 sec
BUILD SUCCESSFUL
Total time   : 16 seconds
Finished at  : Freitag, 25. November 2005 14:38:55 CET

mburger@jonagold:~/ddchange>
```

Figure 5.3: Maven's output running `test` goal: Running the 16 unit tests the first time, all tests are passing.

```
mburger@jonagold:~/ddchange> maven ddchange:store-reports
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.1-beta-1

build:start:

ddchange:prepare-filesystem:
    [delete] /Users/mburger/ddchange/target/ddchange not found.
    [mkdir] Created dir: /Users/mburger/ddchange/target/ddchange
    [mkdir] Created dir: /Users/mburger/ddchange/target/ddchange/
classes
    [mkdir] Created dir: /Users/mburger/ddchange/target/ddchange/
database
    [mkdir] Created dir: /Users/mburger/ddchange/target/ddchange/
reports
    [mkdir] Created dir: /Users/mburger/ddchange/target/ddchange/
sources
    [mkdir] Created dir: /Users/mburger/ddchange/target/ddchange/
test-classes
    [mkdir] Created dir: /Users/mburger/ddchange/target/ddchange/
test-sources
    [echo] Copy the test resources...

ddchange:store-reports:
    [echo] Running test result collector.
    [echo] Name of project in database: commons-lang-2.1:commons-
lang-2.1
Using internal database.
Saved 16 results to database.
BUILD SUCCESSFUL
Total time   : 21 seconds
Finished at  : Freitag, 25. November 2005 14:41:00 CET

mburger@jonagold:~/ddchange>
```

Figure 5.4: Maven's output running `ddchange:store-reports` goal: DDCHANGE MAVEN stores the test results of the previously run tests in an internal database.

debugging this failure by hand, we will use DDCHANGE MAVEN to automatically debug the failure.

Running the goal `ddchange:diff`[4] DDCHANGE MAVEN determines the failure-inducing changes. It initialises the builder (more precisely, the Java compiler), launches the internal database, starts the RMI registry, parses the new unit test reports, collects the changes between the passing and the current version of the project, and starts the remote tester. Having completed the initialisation phase, DDCHANGE MAVEN processes the initial set of 20 changes. After running the delta debugging algorithm, DDCHANGE MAVEN has determined exactly one failure-inducing change. Finally, it stops the launched services. Figure 5.6 shows the output of that process. If there would be more than only one failing test, DDCHANGE MAVEN will debug all of them one after another.

While generating the project's documentation site, DDCHANGE MAVEN integrates its report on failure-inducing changes; the goal `site` generates the whole site. See Figure 5.7 for the index generated by DDCHANGE MAVEN. Because there is one failing unit test, DDCHANGE MAVEN included one report—one individual report for each processed failing unit test. The individual reports are stored in the XML format to ease further processing. The report includes XSLT[5] and CSS[6] files—using a browser that supports the processing of these style sheets, you can view the reports as a nicely formatted XHTML page. That report contains a description of the failed test and the date of the last passing run, a statistic of the delta debugging run, a list of all changes (the difference between the passing and the current version), and the failure-inducing changes.

The screenshot on Figure 5.8 shows an extract of the report transformed by a web browser. DDCHANGE MAVEN required seven tests to determine the failure-inducing change. The first test verifies the empty set of changes. Applying no changes on the passing version, the test has to pass. The second test verifies the complete set, applying all changes the test has to fail. The following five tests check different subsets. In addition to the test outcomes, the statistic contains information about the runtime of the different phases (applying the deltas, building, running the unit test, and so on). For instance, with the help of that information, you can analyse the time-consumption of different builders.

The most important part of the report is the failure-inducing change, as shown in Figure 5.9 (again, an extraction of the transformed report). As we can see, that change changes the constructor of class `SystemUtils`. As noted in the constructor's documentation, its visibility has to be public. We made it private, a best practise when writing utility classes[7]. Fortunately, the

[4] In earlier publications, the variant of the delta debugging algorithm that isolates was called *diff*. Because the book [Zel05] was published while the end game of this work, DDCHANGE uses still the old names of the different variants.

[5] *XSLT* is an XML-based language used for the transformation of XML documents. DDCHANGE MAVEN uses that transformation to create an XHTML page from the XML report.

[6] Cascading Style Sheets (CSS) is a language used to describe the presentation of a document written in a markup language. The transformed XHTML report is styled in this way.

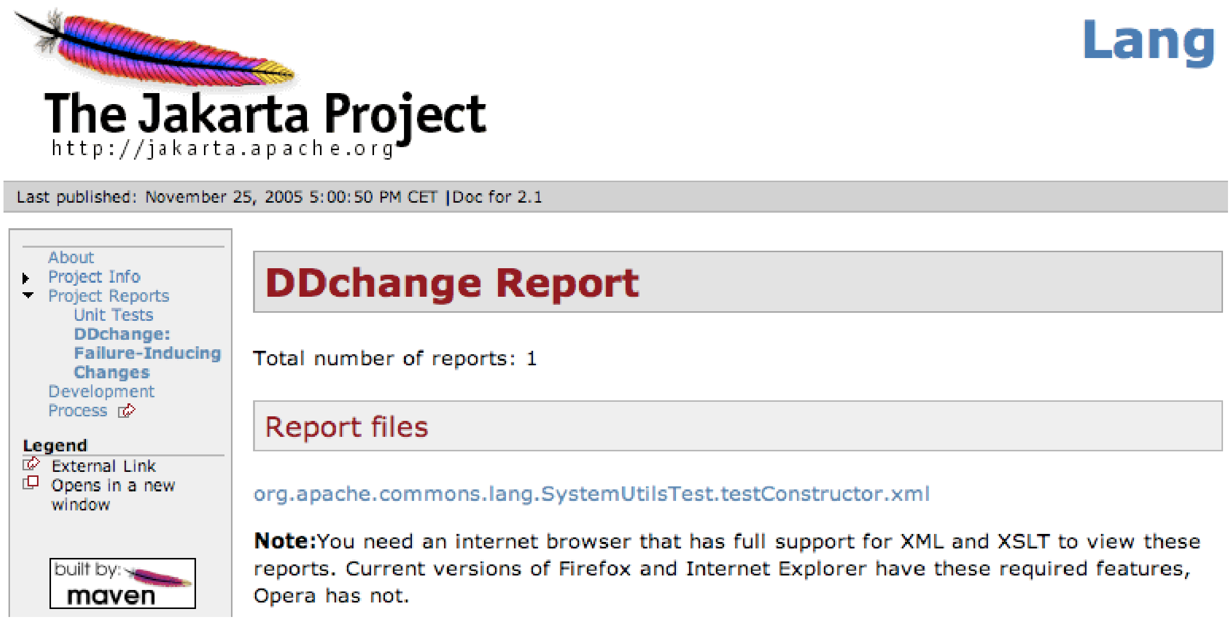
[7] An *utility class* typically defines only static methods. Thus, an instantiation is (often) not desired by the programmer.

```
mburger@jonagold:~/ddchange> maven test
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.1-beta-1

build:start:

java:prepare-filesystem:

java:compile:
    [echo] Compiling to /Users/mburger/ddchange/target/classes

java:jar-resources:

test:prepare-filesystem:

test:test-resources:

test:compile:

test:test:
    [junit] Running org.apache.commons.lang.SystemUtilsTest
    [junit] Tests run: 16, Failures: 1, Errors: 0, Time elapsed:
1,208 sec
    [junit] [ERROR] Test org.apache.commons.lang.SystemUtilsTest
FAILED

BUILD FAILED
File...... /Users/mburger/.maven/cache/maven-test-plugin-1.6.2/
plugin.jelly
Element... fail
Line...... 181
Column.... -1
There were test failures.
Total time   : 11 seconds
Finished at  : Freitag, 25. November 2005 14:45:31 CET

mburger@jonagold:~/ddchange>
```

Figure 5.5: Maven's output running `test` goal a second time: Running all the unit tests after we have applied some changes, one of the tests is failing.

```
mburger@jonagold:~/ddchange> maven ddchange:diff
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.1-beta-1

build:start:

ddchange:diff:
ddchange:prepare-filesystem:
    [echo] Copy the test resources...

ddchange:debug-internal:
    [echo] Running Delta Debugging algorithm 'diff'
    [echo] Name of project in database: commons-lang-2.1:commons-
lang-2.1
Going to initialize builder...
Using internal database.
Going to start rmiregistry...
Going to debug test case: org.apache.commons.lang.SystemUtilsTest
Going to debug test: testConstructor
Latest passing run in database: Fri Nov 25 14:40:32 CET 2005
Going to collect deltas on sources...
Found 19 deltas in /Users/mburger/ddchange/target/ddchange/sources
Going to collect deltas on test sources...
Found 1 deltas in /Users/mburger/ddchange/target/ddchange/test-
sources
Going to revert sources to last passing version...
Going to revert test sources to last passing version...
Going to start remote tester...
Appending to the classpath: junit-3.8.1.jar
Appending to the classpath: commons-lang-2.1.jar
Appending to the classpath: ddchange-framework-test-0.1.4.jar
Going to debug 20 deltas...
Failure-inducing deltas: 1
Going to stop remote tester...
Going to stop internal database...
Going to stop rmiregistry...

BUILD SUCCESSFUL
Total time   : 2 minutes 15 seconds
Finished at  : Freitag, 25. November 2005 14:49:41 CET

mburger@jonagold:~/ddchange>
```

Figure 5.6: Maven's output running `ddchange:diff` goal: DDCHANGE MAVEN determines one failure-inducing change out of 20.


The Jakarta Project
http://jakarta.apache.org

Lang

Last published: November 25, 2005 5:00:50 PM CET |Doc for 2.1

DDchange Report

Total number of reports: 1

Report files

org.apache.commons.lang.SystemUtilsTest.testConstructor.xml

Note: You need an internet browser that has full support for XML and XSLT to view these reports. Current versions of Firefox and Internet Explorer have these required features, Opera has not.

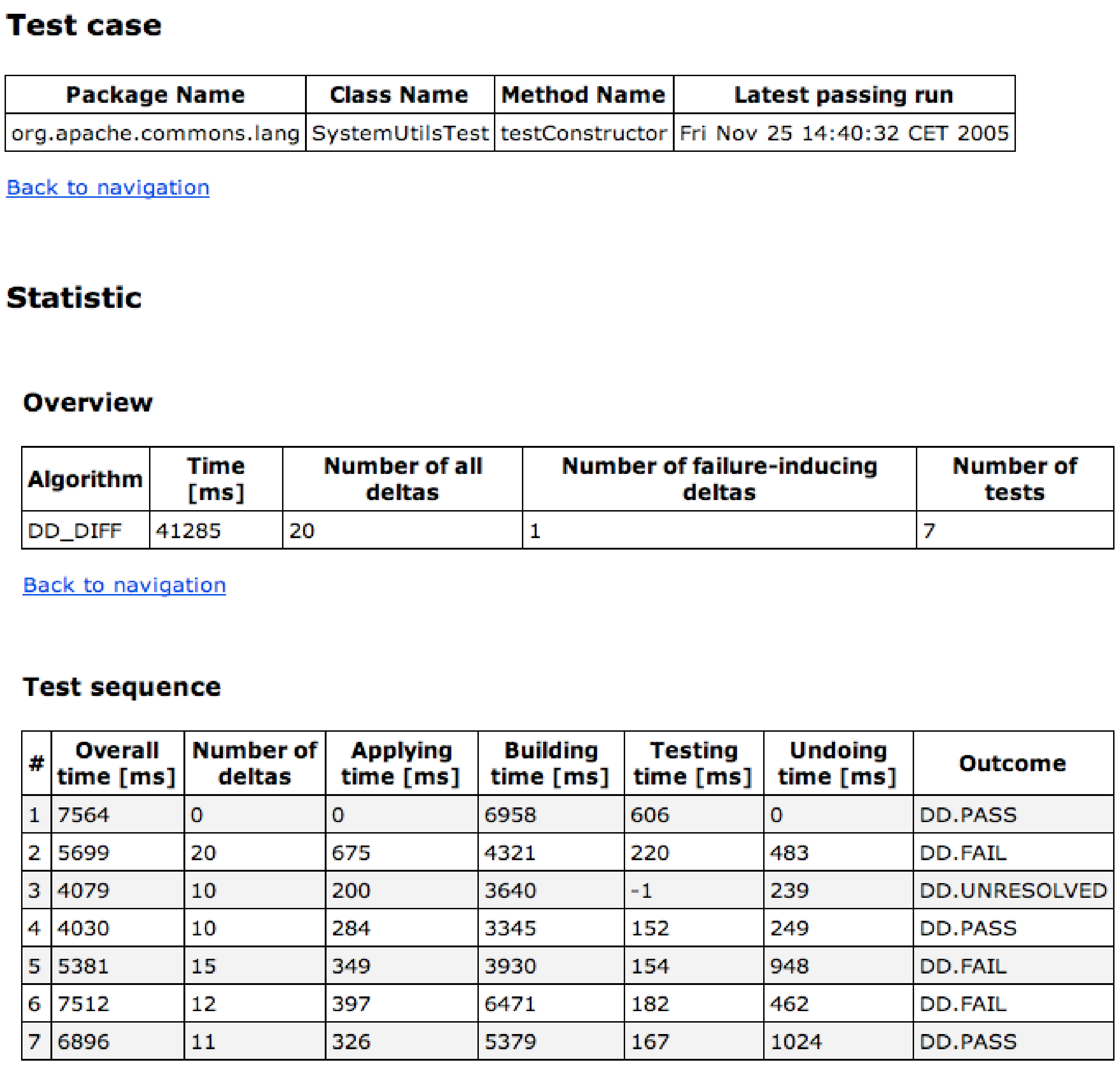
Figure 5.7: Screenshot: Index of documentation site. The project reports contain the one generated by DDCHANGE MAVEN.

unit test `SystemUtilsTest` checks the visibility.

The example described above is a relatively small and simple one. First, the difference between the passing and the failing program version consists of only 20 changes. Debugging manually this small number of changes is not too hard. Second, the Commons Lang library is a pure class library. Thus, the individual classes do not interact a lot with each other; the failing unit test points out quite well the possible location of the defect. However, we can demonstrate the usage and the capability of DDCHANGE MAVEN. If we would increase the number of changes, the algorithm would compute the same result, requiring a longer time—we would not gain more insight.

Configuration

You can use DDCHANGE MAVEN out-of-the-box; it provides sensible default configuration values. If you do not specify an external database, DDCHANGE MAVEN will use an instance of the HSQLDB database, the database files will be stored in the target folder of your project. DDCHANGE MAVEN uses the username and the password stored on your computer and auto-detects the client type while accessing the Subversion repository. Thus, unless you want to use an external database and you do not want to store your Subversion password on your computer, you need not to configure any property of DDCHANGE MAVEN—all settings are optional.



Test case

Package Name	Class Name	Method Name	Latest passing run
org.apache.commons.lang	SystemUtilsTest	testConstructor	Fri Nov 25 14:40:32 CET 2005

Back to navigation

Statistic

Overview

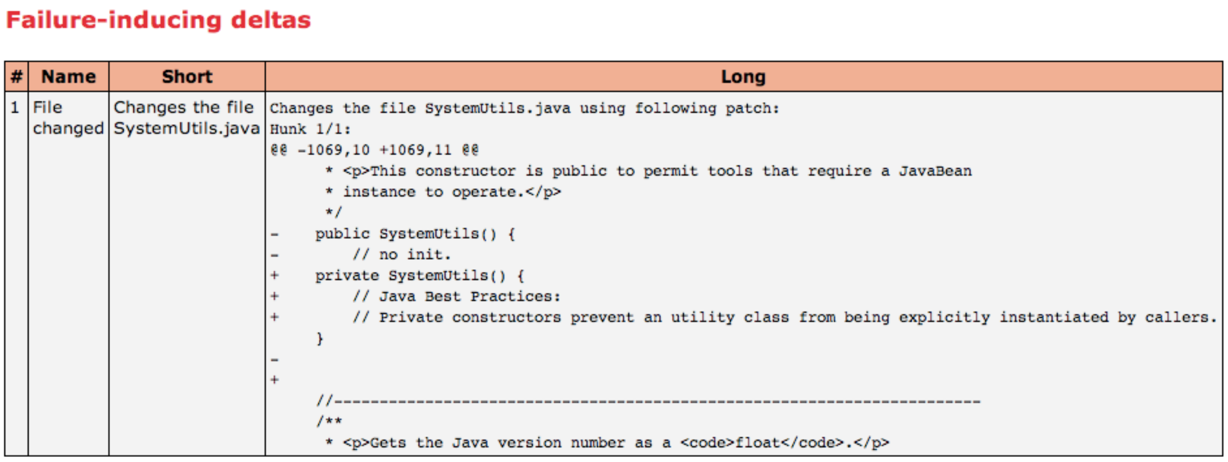
Algorithm	Time [ms]	Number of all deltas	Number of failure-inducing deltas	Number of tests
DD_DIFF	41285	20	1	7

Back to navigation

Test sequence

#	Overall time [ms]	Number of deltas	Applying time [ms]	Building time [ms]	Testing time [ms]	Undoing time [ms]	Outcome
1	7564	0	0	6958	606	0	DD.PASS
2	5699	20	675	4321	220	483	DD.FAIL
3	4079	10	200	3640	-1	239	DD.UNRESOLVED
4	4030	10	284	3345	152	249	DD.PASS
5	5381	15	349	3930	154	948	DD.FAIL
6	7512	12	397	6471	182	462	DD.FAIL
7	6896	11	326	5379	167	1024	DD.PASS

Figure 5.8: Screenshot: Report on statistics. The algorithm required seven tests in order to determine the failure-inducing changes.



Failure-inducing deltas

#	Name	Short	Long
1	File changed	Changes the file SystemUtils.java	Changes the file SystemUtils.java using following patch:

```
Hunk 1/1:
@@ -1069,10 +1069,11 @@
      * <p>This constructor is public to permit tools that require a JavaBean
      * instance to operate.</p>
      */
-    public SystemUtils() {
-        // no init.
+    private SystemUtils() {
+        // Java Best Practices:
+        // Private constructors prevent an utility class from being explicitly instantiated by callers.
     }
-
+
    //-----------------------------------------------------------------------
    /**
     * <p>Gets the Java version number as a <code>float</code>.</p>
```

Figure 5.9: Screenshot: Report on changes. The failure-inducing change modifies the visibility of the constructor.

However, DDCHANGE MAVEN provides a number of configurable properties. Amongst others, you can configure the database access (username, password, type of database, hostname, and so on), the Subversion client adapter type (auto-detect, javahl, and JavaSVN), and whether the compiler scrubs the destination folder or not.

As well as any other Maven plug-in, you can configure DDCHANGE MAVEN using the `project.properties` file. A complete list of all settings is contained in the DDCHANGE MAVEN project documentation site on the included CD (see Appendix A.1). The CD also contains a list of all goals provided by the plug-in. For instance, you need not create any tables in your database. The goal `ddchange:export-schema` exports the required schema to the specified database (creates tables).

If you want to include the DDCHANGE MAVEN report in you project documentation site, you have to add a `report` element with the text content `ddchange-maven-plugin` to your `project.xml` file. Running the `site` goal, Maven will include all reports on failure-inducing changes.

Run by CruiseControl

As described above, DDCHANGE MAVEN can be run manually in order to determine failure-inducing changes automatically. If a test fails, the developer will have to execute the appropriate Maven goals. In Section 5.1.2, we introduced CruiseControl; that framework enables us to build and test the software project as soon as a developer commits some changes to the repository. Such a build can be costly; some projects require more than one hour in order to build and test the complete product. Therefore, CruiseControl can be run dependent on time instead of dependent on individual commits. Large projects, for example the Eclipse project, use so-called nightly builds for their continuous build process.

The integration process provided by CruiseControl is suitable for running the delta debugging process as soon as a failure occurs. You can start the process using the `<onfailure>` publisher (see Figure 5.2). This publisher can execute the `<execute>` publisher, a publisher that executes again a command as part of the publishing phase. In this way, DDCHANGE MAVEN will run completely on its own.

Once launched, DDCHANGE MAVEN will run the delta debugging algorithm on the server that processes the integration build. Finally, it will write the determined failure-inducing changes to report files in the XML format. CruiseControl provides other publishers that can transform the XML files to HTML (`<xsltlogpublisher>`) and that can send the reports to the developers (`<email>`).

The debugging process can be a long-running task, depending on the number of failed tests and the number of changes that have to be analysed by DDCHANGE MAVEN. Especially if you run CruiseConrol on every commit, you probably want to avoid starting the debugging process a second time while another process is running. For these purposes, DDCHANGE MAVEN provides a special configuration property. You can specify a lock file that will be deleted as soon as the debugging process is finished.

Describing all aspects of the CruiseControl configuration in order to initiate the debugging process goes beyond the scope of this work. See [FJK05] for more information about CruiseControl and for a configuration reference. Using the publisher mentioned above, you will have the ability to automatically start the automatic debugging.

5.1.4 Architecture

Compared to the framework (that consists of about 100 classes) the Maven plug-in is very small. Most of the functionality is defined in 17 classes, and most of them are smaller utility classes. In the following sections, we will discuss the main concepts at a glance.

Integration with Maven

The *heart* of DDCHANGE MAVEN is the *plugin.jelly* file, a file written using Jelly, an XML-based scripting language [Fou05a]. That file defines the plug-in's goals and is required by every Maven plug-in. To minimise the amount of Jelly code, the goals run directly JavaBeans[8]. Jelly uses the methods defined in the Bean to set the configuration values and to run the different goals. The default configuration values are defined in the file *plugin.properties* and can be overwritten by the user.

[8] A *JavaBean* is a special class that follows certain conventions about method naming, constructors, and so on. These conventions enable tools or other classes to access the fields in a well-defined way. A JavaBean has to be in accordance with the JavaBeans API Specification [Ham97].

Furthermore, the Jelly script file registers the report with Maven for inclusion on the project's documentation site. The Jelly Stylesheet Library (JSL), a tag library that implements an XSLT-like declarative XML based processing engine, is used in order to include the report files.

The JavaBean Classes

DDCHANGE MAVEN contains three concrete JavaBean classes. The class `SchemaExporter` is run by the goal that exports the schema, the class `ReportCollector` by the goal that stores the unit test results in the database, and the class `DebuggingEntryPoint` by the goals that initiate the debugging process. The three classes extend a hierarchy of two other Beans, named `PluginPropertiesBean` and `PluginPropertiesBeanExtended`. The first one defines the setters and getters for the configuration values. The default value for all properties is null and their type is String. The second one extends the first one and adds convenience methods to deal with the properties. These methods allow getting a property as another type than `String`, for example as `File`. The methods further check for valid configuration values. If the method returns a File object, it checks whether the file exists; if not, it will throw an exception. Thus, the first Bean in the hierarchy adds unwise setters and getters; the second one adds checks for the concrete properties and their values. If you want to add a property, you should first add the getter and the setter to the first Bean. If you require some checks on the values, you can extend the second one.

Because the three Beans run by the goals extend the Beans responsible for all settings, they can use all properties defined in the upper class hierarchy. You need not to add the methods for new properties to the different Beans, and all Beans can profit directly from the new properties and their implemented checks. For instance, the methods that set and returns the properties for the database connection are defined once and can be used in all of the concrete Beans.

The class `DebuggingEntryPoint` does not extend directly the class `PluginPropertiesBeanExtended`. Instead, it extends the class `AbstractEntryPoint`, that class extends again the Bean responsible for the properties. See Figure 5.10 for a UML diagram of the class hierarchy. The class `SchemaExporter` extends directly the first Bean in the hierarchy, because it requires only simple string values.

The class `AbstractEntryPoint` parses the test reports and checks them for failed tests. If there is any failed test, it will initialise two compilers, one that compiles the main sources and another one that compiles the unit test sources. Using a MultiBuilder, DDCHANGE will compile all required classes. Finally, that Bean will initialise a DAO and will launch the internal database instance, if required. The initialised builder and DAO can be obtained via provided getter methods.

Because Maven uses the Ant compile task in order to compile the Java source files of the project and the DDCHANGE framework provides an Ant compiler that can be used programmatically, the Bean only needs to call the associated setters to transfer the Maven compile settings to the

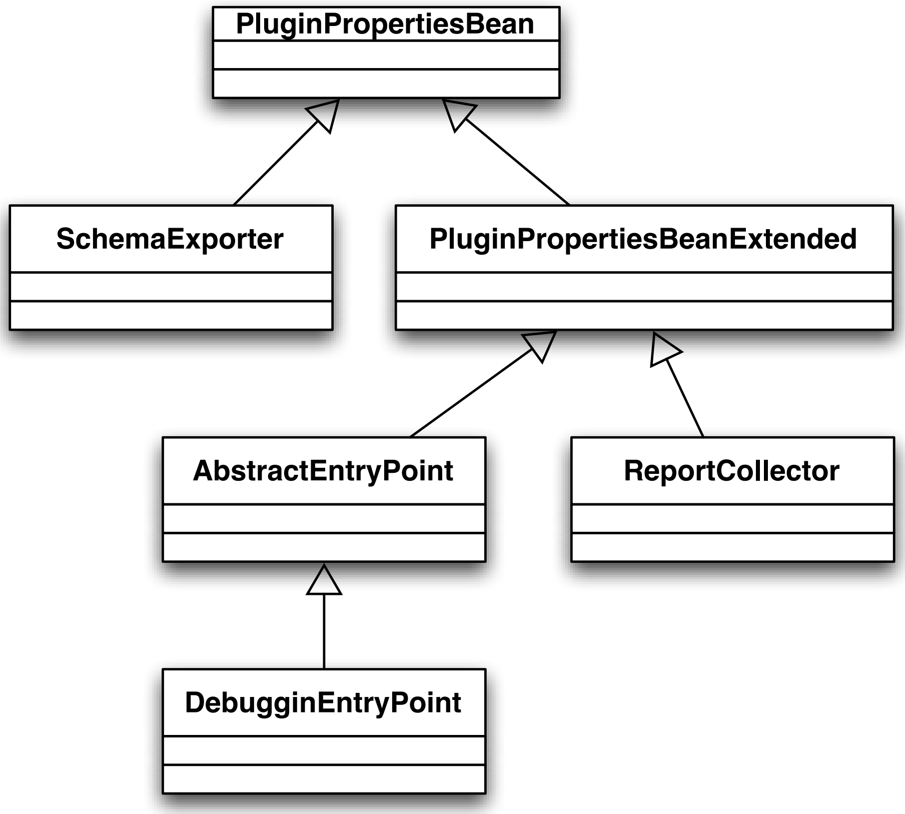


Figure 5.10: UML class diagram of the JavaBean class hierarchy. These Beans defines the functionality of DDCHANGE MAVEN. The diagram shows no methods.

Ant compiler. Thus, all compile settings provided by Maven can be used with DDCHANGE MAVEN.

The class `DebuggingEntryPoint` iterates over the failed tests, collects the changes using the Subversion component of DDCHANGE, and runs the selected algorithm, for instance isolation or minimisation—DDCHANGE MAVEN provides different goals for the different variations of the delta debugging algorithm. Finally, it writes the computed results including the statistics to the XML report files.

Most of the plug-in's code consists of the Bean classes and several utility classes that provides methods to launch a database instance, to parse the test reports, and so on. As you have seen, the architecture of DDCHANGE MAVEN is rather uncomplicated and compact.

5.1.5 Adaptations to the Framework

The Maven plug-in is an example for the *out-of-the-box* feature of DDCHANGE. No adaptations to the framework were required in order to implement DDCHANGE MAVEN. As described above, this plug-in uses the provided Ant compiler. That fact has two advantages. First, we need not to implement a compiler. Second, all Maven settings relating to the compilation of the Java sources are supported by the plug-in.

In principle, DDCHANGE MAVEN consists of the Jelly script file that builds the bridge between the Maven goals and the executed code, and different JavaBeans that implement the functionality of the provided goals. The Beans revert to small utility classes that encapsulate the calls to the framework. The main Bean that runs the algorithm instantiates and initialises all required classes and finally calls the tip of the iceberg (see Section 4.3.5).

5.2 Eclipse Plug-In

The platform's second concrete application instantiated from the framework is a plug-in for Eclipse, called consequently *DDCHANGE ECLIPSE* . In contrast to the Maven plug-in, this plug-in integrates the delta debugging algorithm directly with the developer's IDE for the Java programming language.[9]

In the following section, we will introduce Eclipse at a glance. The succeeding sections describe the usage, the architecture and finally the adaptations to the framework, by analogy with the description of the Maven plug-in.

[9]DDCHANGE MAVEN can be used of course on the workstation by calling the goals manually. However, it was developed in order to be run during the continuous integration.

5.2.1 About Eclipse

In the context of this work, the Eclipse's nature being an SDK (software developer kit) for the Java programming language is the most relevant aspect of Eclipse. However, Eclipse provides much more than only being an IDE:

> "Eclipse is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. It is composed of three projects, the Eclipse Project, the Eclipse Tools Project and the EclipseTechnology Project[...]" [Fou05c]

Eclipse is an open-source, platform-independent software framework. Based on that platform you can deliver so-called *rich-client* applications (as opposed to *thin-client* browser-based applications). So far (until version 2.x), that platform was generally used to develop IDEs (Integrated Development Environments). The main representative was the Java IDE called Java Development Toolkit (JDT). The JDT is also used to develop Eclipse itself. Starting with version 3.0, the concept of rich-client applications was introduced, now Eclipse can be used for other types of client application as well.

The *Eclipse Project* mentioned in the quote above is the interesting one for us:

> "The Eclipse Project is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. It is composed of three subprojects, Platform, JDT - Java development tools, and PDE - Plug-in development environment.[...]" [Fou05c]

The Eclipse SDK is the combination of the three Eclipse Project subprojects into a single download. Together, these projects provide a development environment that allows the developer to create tools that integrate into the Eclipse Platform.

That Platform can be extended by writing and contributing *plug-ins* to it. A plug-in is the smallest unit of Platform function that can be developed and delivered separately. All of the Platform's functionality is provided by such plug-ins, except for the Platform Runtime component (a relative small kernel). The Eclipse white paper [OTI01] provides a good overview on the architecture, the technical articles on the Eclipse Corneer [EC05] provides further information that go into detail.

The main advantage of this lightweight software component framework is the expandability and the reusability of new and existing plug-ins, respectively. For instance, the JDT provides plug-ins to launch Java programs, to run JUnit tests and to use Eclipse's incremental Java compiler. All these provided features support some of the steps of the general plan (see Section 3.4). Thus, Eclipse is a second tool that is suited for a tool that determines failure-inducing changes.

5.2.2 Usage

The Java Development Toolkit (JDT) integrates the JUnit testing framework very tightly. You can run your unit tests at the push of a button; Eclipse runs them and provides direct feedback via its user interface (UI). You can inspect the test results in the so-called JUnit view[10] (see the left side of Figure 5.11 for an example of this view showing a failure).

The JDT provides a mechanism that enables us to observe the running JUnit tests. DDCHANGE ECLIPSE uses that mechanism and starts the debugging process as soon as a unit test fails. In the following, we exemplarily describe the usage of the plug-ing. The following sections will describe the architecture, including the mechanism mentioned above.

Example Run

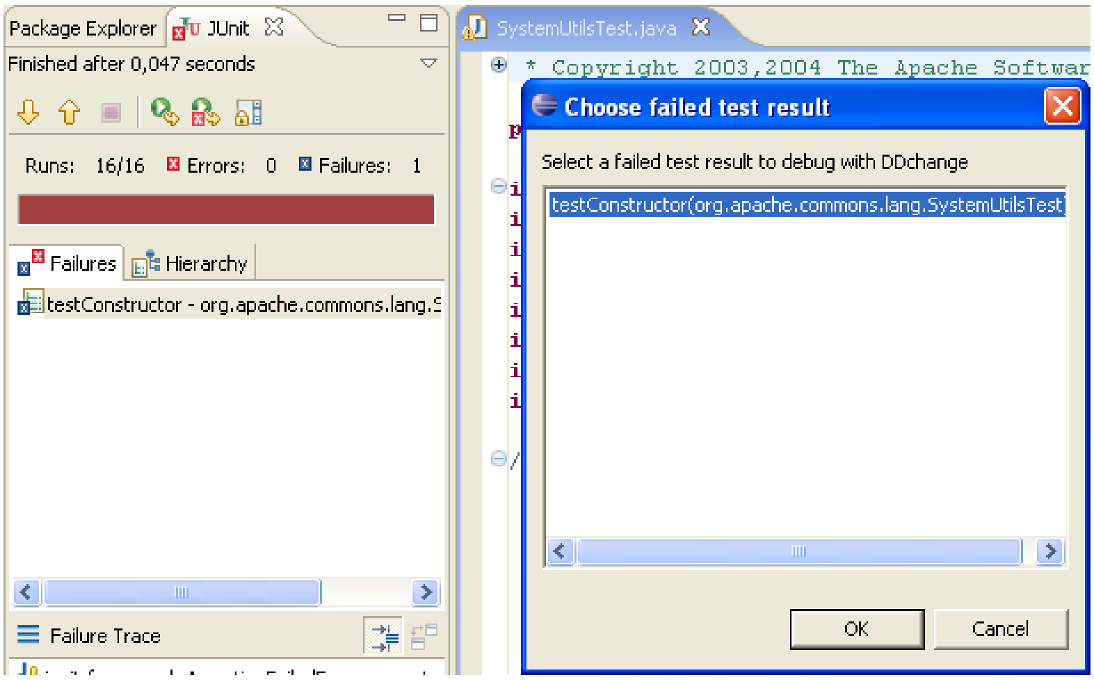
In this example, we will use again the “Apache Jakarta Commons Lang” project and the same failure as in Section 5.1.3. Thus, you should be familiar with the basic conditions. Using the Maven goal `eclipse` you can automatically generate the Eclipse project files. Afterwards, you can import that project into Eclipse without further effort.

If one or more unit tests run in the JDT fail, DDCHANGE ECLIPSE will show a dialog that lists all failed tests. You can choose one test to debug or cancel the process. Figure 5.11 shows a detail of a screenshot showing the described dialog. The test method testConstructor in unit test SystemUtilsTest failed. Therefore, DDCHANGE ECLIPSE opened the dialog.

After we have selected and confirmed the failed test, DDCHANGE ECLIPSE immediately starts the debugging process. At first, the plug-in makes a copy of your project; all changes are applied on that copy. If DDCHANGE ECLIPSE would crash or would not shut down properly, only the copy of your project may be corrupt, not your original project. Afterwards, it collects all the changes between the passing and the current version of your project. Instead of using a version control system, DDCHANGE ECLIPSE utilises the *local history* of Eclipse.

Each time you edit and save a file, Eclipse saves a local copy of that file. You can replace the current file with a previous edit or even restore deleted files. Each entry in the local history is uniquely represented by the date and time the file was saved. The local history lacks many features of a version control system; however, it provides all features required for our purposes—we can get the content of former versions. In contrast to typical version control systems, the local history is restricted in space. By default, the local history maintains changes not older than seven days, it keep not more than 50 entries per file, and the maximum size of individual states must not exceed one megabyte. Therefore, the local history is appropriate for a relative small number of changes.

[10] A *view* is one of the main visual entities that appear in the *workbench*; the term workbench refers to the desktop development environment. The views provide some context of the shown *editor*, another one of the main visual entities.



Finished after 0,047 seconds

Runs: 16/16 Errors: 0 Failures: 1

testConstructor - org.apache.commons.lang.S

* Copyright 2003,2004 The Apache Softwar

Choose failed test result

Select a failed test result to debug with DDchange

testConstructor(org.apache.commons.lang.SystemUtilsTest)

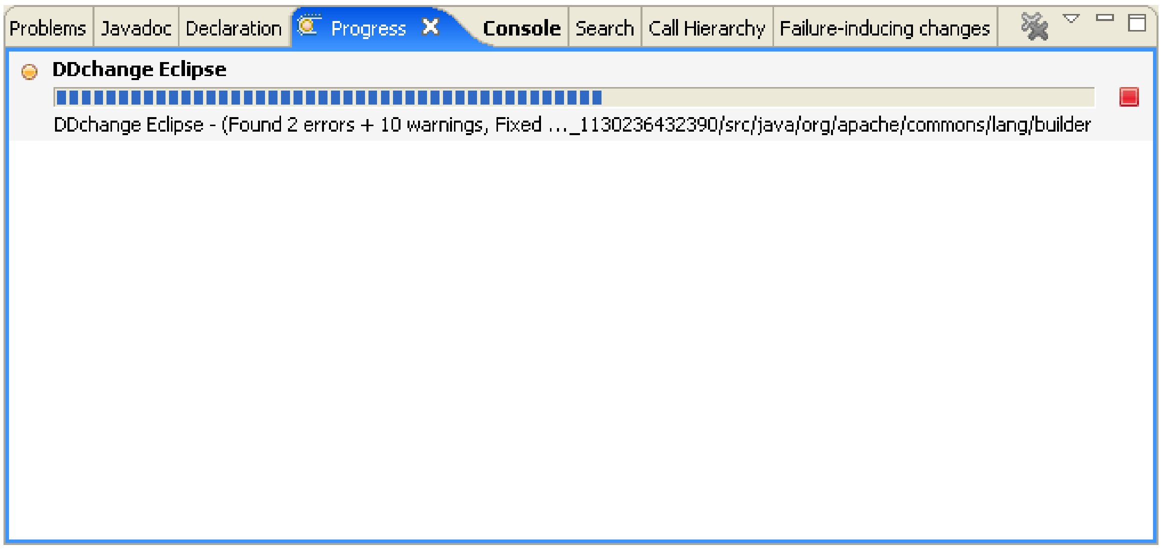
Figure 5.11: Screenshot: As soon as a test failed, DDCHANGE ECLIPSE prompts a test to debug. You can cancel that prompt, if you do not want to start the determination of the failure-inducing changes.

After DDCHANGE ECLIPSE has computed the initial set of changes, it launches the remote tester and starts the delta debugging algorithm. As you can see in Figure 5.12, the whole debugging process *runs in the background*. You can continue your work while waiting for the determined failure-inducing changes.

DDCHANGE ECLIPSE cannot report the actual progress of the debugging process, because we do not know how many tests are required ahead of the computed result. The plug-in provides a *text console* that indicates some progress information about the process. Figure 5.13 shows a snapshot of this information.

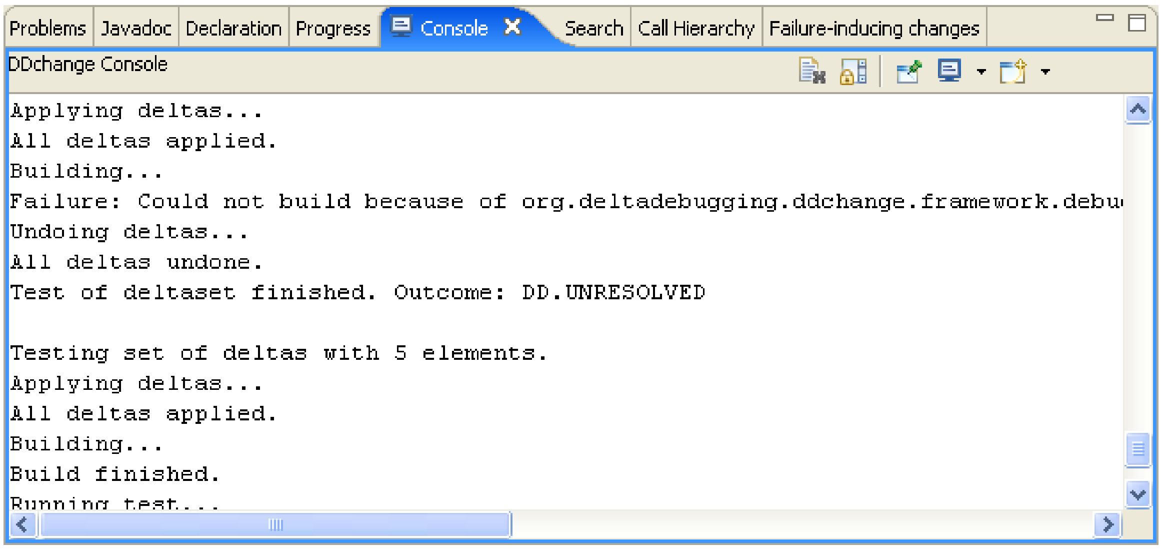
As soon as the debugging process is completed, DDCHANGE ECLIPSE shows the result in its own view. The view shows two trees, the first one contains the initial set if changes, the second one the failure-inducing ones. The view allows you to browse all changes by expanding the individual nodes in the tree; the changes are organised according to the affected files. Figure 5.14 shows the failure-inducing changes determined while debugging the same failure as in Section 5.1.3. This time, the algorithm determined two instead of one change, because we have used simplification instead of isolation. If you double-click the tree containing the failure-inducing changes, DDCHANGE ECLIPSE will revert all appendant changes. In this case, you can re-run the failed test and it will pass—we have found the defect in the code.

You can inspect the individual changes by double-clicking. DDCHANGE ECLIPSE will open a dialog that shows the content of the change (See Figure 5.15). A minus sign indicates a removed



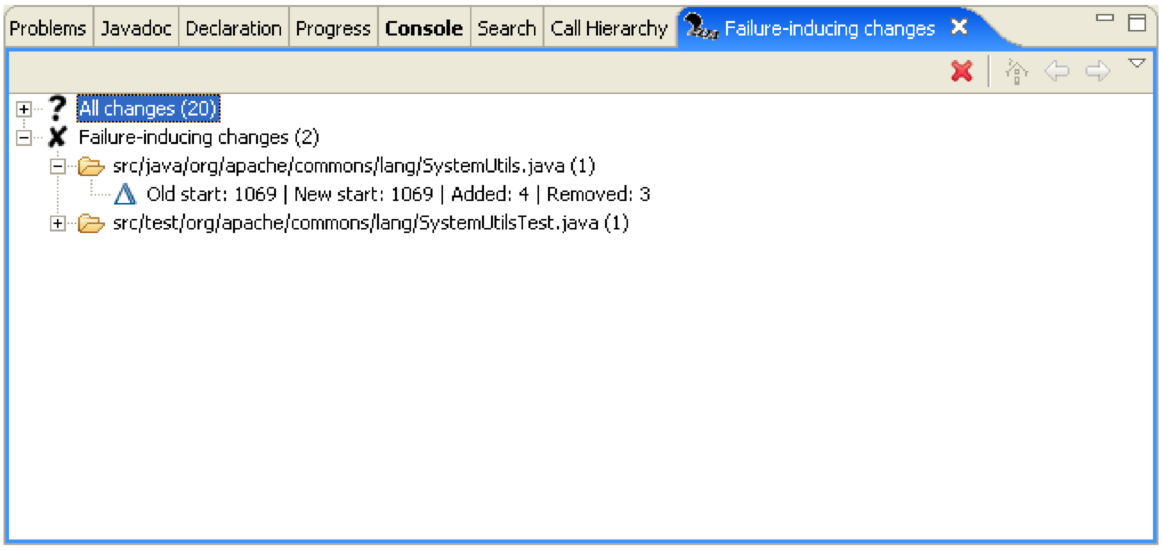
DDchange Eclipse - (Found 2 errors + 10 warnings, Fixed ..._1130236432390/src/java/org/apache/commons/lang/builder

Figure 5.12: Screenshot: DDCHANGE ECLIPSE runs in the background, you can continue with your work while the plug-in determines the failure-inducing changes.



```
Applying deltas...
All deltas applied.
Building...
Failure: Could not build because of org.deltadebugging.ddchange.framework.debu
Undoing deltas...
All deltas undone.
Test of deltaset finished. Outcome: DD.UNRESOLVED

Testing set of deltas with 5 elements.
Applying deltas...
All deltas applied.
Building...
Build finished.
Running test...
```

Figure 5.13: Screenshot: DDCHANGE ECLIPSE provides a text console that shows the progress of the algorithm.



All changes (20)
Failure-inducing changes (2)
src/java/org/apache/commons/lang/SystemUtils.java (1)
Old start: 1069 | New start: 1069 | Added: 4 | Removed: 3
src/test/org/apache/commons/lang/SystemUtilsTest.java (1)

Figure 5.14: Screenshot: The view “Failure-inducing changes” shows both the initial set of changes and the failure-inducing ones. The tree contains two changes because the *simplification* was used. If you double-click the tree labeled “Failure-inducing changes”, DDCHANGE ECLIPSE will revert all appendant changes.

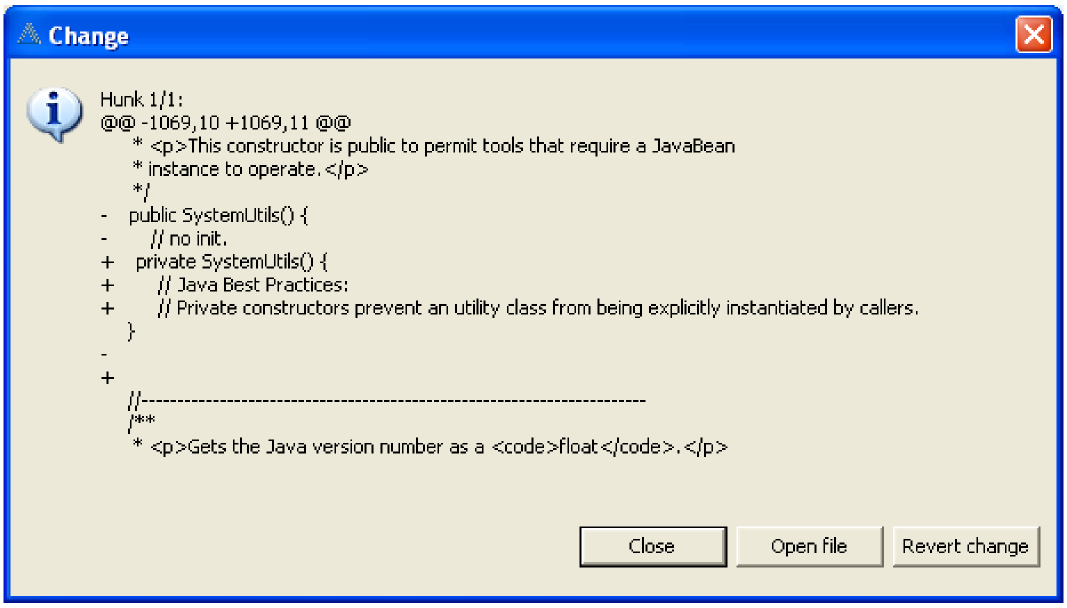
line, a plus sign an added line. You can open the file in its associated editor (the editor will jump to the location of the change), or you can revert the shown change.

DDCHANGE ECLIPSE integrates tightly the delta debugging algorithm with the Java Development Toolkit. As soon as a unit test fails, you can start the debugging process. The whole process runs in the background, you can continue with your work.

5.2.3 Architecture

In the following, we will discuss the architecture of DDCHANGE ECLIPSE at a glance. The tool consists of several plug-ins, these plug-ins form a so-called feature. Thus, DDCHANGE ECLIPSE is not a single plug-in, it is in fact a feature. In the context of this work, the most interesting plug-ins are the *core* and the *ui* plug-ins. The first one implements the core functionality, the second one the user interface. This allows running the *headless* core functionality without loading the user interface classes. Furthermore, the feature consists of two additional plug-ins and one more project. We will describe the two plug-ins mentioned at first in more detail than the other ones.

Before we start with DDCHANGE ECLIPSE’s architecture in detail, we will touch on the way of contributing new functionality to the Eclipse platform.



```
Hunk 1/1:
@@ -1069,10 +1069,11 @@
     * <p>This constructor is public to permit tools that require a JavaBean
     * instance to operate.</p>
     */
-   public SystemUtils() {
-       // no init.
+   private SystemUtils() {
+       // Java Best Practices:
+       // Private constructors prevent an utility class from being explicitly instantiated by callers.
    }
-
+
    //-----------------------------------------------------------------------
    /**
     * <p>Gets the Java version number as a <code>float</code>.</p>
```

Figure 5.15: Screenshot: You can examine carefully the individual changes. In this dialog, you can open the file in the associated editor (DDCHANGE ECLIPSE will jump to the changed lines), you can revert the shown change, or you can simple close the dialog.

Contributing to Eclipse

The basic concepts of Eclipse plug-ins are discussed not in detail here, various books including [GB03], [CR04], and [AL04] explicate the fundamental principles and patterns used in order to develop plug-ins and features for the Eclipse platform. However, in order to understand the architecture of DDCHANGE ECLIPSE, we will look at the most important terms and concepts.

As mentioned above, you will contribute to Eclipse using *plug-ins*. A plug-in, comparable to an object, is an encapsulation of behaviour and data. It interacts with other plug-ins to form a running program. The most important part of a plug-in is the *manifest file*. That XML file defines at least the basic description (name, id, and version number). The manifest may also define what Java code it provides (one way how the plug-in may implement its functionality, another way is HTML code to provide some documentation) and what other plug-ins it requires. The manifest allows *lazy loading*; the implementation code is loaded when the functionality is required. For instance, the Java classes of DDCHANGE ECLIPSE are loaded not before a unit test failed, about to the dialog that prompts a unit test to debug.

Another important aspect of plug-ins are their (optional) *extension points*. The mechanism of extensions and extension points allows loose coupling of the plug-ins. A metaphor used often to explain that mechanism is “electrical outlets”. The power point is the extension point; the plug the extension. The extension point declares a contract, other plug-ins must accept that contract in order to use or extend the provided functionality. The contract consists mostly of XML declarations and Java interfaces.

The important term explained at last here is a *feature*; it groups and describes different functionality (plug-ins) that makes up a product. That mechanism allows the feature to be installed and updated using the *update sites*. Eclipse's update manager locates and installs a new feature from such a site.

In addition to the books mentioned above, [GBM03] and [SHNM04] provides useful information about developing user interfaces for the Eclipse platform.

The Core Plug-In

The key tasks of the *core plug-in* is observing of run JUnit tests and starting the automatic debugging process. Furthermore, it uses other mechanisms that collect changes and build the changed Java source files. In this section, we will focus on aspects that mainly concern Eclipse in general. In Section 5.2.4, we will discuss the adaptations to the framework.

The JDT contains a JUnit plug-in that encapsulates the JUnit testing framework. That plug-in runs the unit tests and defines an extension point that allows you to observe the runs. The interface `ITestRunListener` defines a reporting scheme; this listener is notified when test runs are started and ended. DDCHANGE ECLIPSE uses three implementations of that interface. The first one, the `AbstractTestRunListener`, is an abstract listener that collects all individual test results while a unit test or a test suite is run. After the complete test run has ended, it calls the abstract method `testRunEnded()`. The concrete subclasses can obtain all collected test results using the `getJUnitTestResults()` method, or all failed test results calling the method `getJUnitFailedTestResults()`.

The class `DatabaseTestRunListener` extends `AbstractTestRunListener`; it persists all collected test results as soon as the complete test run has finished. It schedules the job (see below) `TestResultPersistenceJob`, a job that uses the DAO provided by DDCHANGE to store all collected results in an internal database (see Section 4.3.1). DDCHANGE ECLIPSE uses an instance of HSQLDB, the database files are stored in the plug-in's state location. Because the plug-in uses only the local history, an internal database is sufficient.

The last one extending `AbstractTestRunListener` is the class `TestRunListener`. After the complete test run has finished, it obtains all failed test results and will schedule the job `DDchangeJob` if there is at least one failed test; this job starts the automatic debugging process. The job class is comparable to the class `DebuggingEntryPoint` of DDCHANGE MAVEN (see Section 5.1.4). It copies the project, obtains the initial set of changes from the local history, initialises the builders, launches the remote tester, and initiates the algorithm. Afterwards, it reports the determined failure-inducing changes.

In contrast to DDCHANGE MAVEN (it debugs successive all failed tests), DDCHANGE ECLIPSE debugs only one failed test[11]. The plug-in defines an extension point; part of it is the interface `IDDchangeStartCallBack`. That interface defines one method. The method's single argument is an array of failed test results; its implementation has to return one of the contained results to specify the test to be debugged.

DDCHANGE ECLIPSE defines a second extension point in order to allow clients for observing the debugging process and for obtaining the final result. The contract of this extension point is mainly defined (besides the XML declaration in the manifest) in the interface `IDDchangeRunListener`. It extends `IDeltaDebuggingListener`, DDCHANGE's listener interface for observing the execution of the delta debugging algorithm (see Section 4.3.5). The extended interface defines methods to communicate additional events to the clients. For instance, if DDCHANGE ECLIPSE could not find a passing test result in the database, it will notify all listeners.

The Eclipse platform supplies a concurrency infrastructure, as support for scheduling and interacting with background activity. DDCHANGE ECLIPSE uses that infrastructure to process possible long-running tasks; for instance, the run of the algorithm. It schedules these tasks and the infrastructure executes the tasks (in terms of the API so-called jobs) in a well-defined manner. Because DDCHANGE ECLIPSE uses that background tasks-jobs API, the whole debugging process can be run without seizing the user interface—the developer can continue with her work while the tool labours in the background.

The database and the RMI registry are also launched via jobs. Because the core plug-in is loaded when, and only when, its functionality is required the first time (a unit test is run), the database and the registry are not launched during the start-up of the Eclipse platform. DDCHANGE ECLIPSE's plug-in runtime class[12] schedules several jobs that are responsible for launching the services.

The UI Plug-In

The *UI plug-in* supplies the user interface of DDCHANGE ECLIPSE. It uses several concepts and patterns that are common practice when implementing user interfaces for Eclipse products and plug-ins, respectively. They include views, actions, toolbars, tree viewers, content providers, decorators, and more. See [GBM03] and [SHNM04] for more information about these concepts and patterns. Describing these terms and the related implementations does not really fit in the context of this work.

[11] That restriction was caused by time limitations. Displaying only one computed result in the UI can be accomplished with less effort than showing the results of several debugged tests. However, you could extend the plug-in to debug more than one test.

[12] The class `org.eclipse.core.runtime.Plugin` is the abstract superclass of all plug-in runtime class implementations; it defines life cycle methods in order to react to the life cycle requests automatically issued by the platform.

The most interesting aspect of the user interface is the communication with the core. The UI plug-in solely uses the extension points defined by the core; thus, it contributes different extensions to the core plug-in. First, its implementation of the `IDDchangeStartCallBack` provides the dialog that prompts a test to debug (see Figure 5.11). That callback returns the chosen test to the core, which starts debugging that test afterwards. The second extension to the core consists of two implementations of the `IDDchangeRunListener`. The implementing class `DDChangeConsole` provides the text console that shows some information about the running process (see Figure 5.13). The implementing content provider `DDchangeContentProvider` feeds the tree view of the changes with the initial set and the determined result (see Figure 5.14).

That is a short description of the second main plug-in of DDCHANGE ECLIPSE. However, the UI is certainly not the most interesting part of the platform.

Other Plug-Ins and Projects

The DDCHANGE ECLIPSE feature is made up of more than the core and the UI plug-in—additional plug-ins and projects form the feature. The following list describes these other parts briefly.

- *Logging plug-in.* The logging plug-in adds a configurable logging facility using Apache's Log4j. It contains several implementations of Log4j's AppenderSkeleton, an abstract superclass with common functionality that is required by classes that output log statements in some way. Using that plug-in, you can use the Log4j logging mechanism while also redirecting the log statements to Eclipse's plug-in logging framework. See [Mar04] for the basic idea. This way, the log statements (especially warnings and errors) of the DDCHANGE framework are redirected to the workbench user interface.

- *Feature plug-in.* The feature plug-in is required to *brand* the DDCHANGE ECLIPSE feature. In this way, the "About Eclipse SDK Features" dialog lists the DDCHANGE ECLIPSE feature and its plug-ins along with version information.

- *Feature project.* That project hosts the definition of the DDCHANGE ECLIPSE feature. Its feature.xml file describes the feature and composes the core, the UI, the logging, and the feature plug-in to a downloadable tool.

- *Update site project.* The update site project hosts the DDCHANGE ECLIPSE feature. You can export and upload all required files to a webserver. The Eclipse update manager can be used do install and update the feature in a comfortable and user-friendly way from a website.

5.2.4 Adaptations to the Framework

As already mentioned, DDCHANGE ECLIPSE makes some adaptations to the DDCHANGE framework. It does not use the Subversion component to compute the initial set of changes, and it uses another builder to compile the changed Java source files. Eclipse's *local history* is the source for the changes, and Eclipse's *incremental compiler* compiles the changed sources. The following two sections describe these contributions in more detail.

Local History

DDCHANGE ECLIPSE contributes the access to Eclipse's local history (see Section 5.2.2 for a short introduction). That contribution consists of different classes that enable the core plug-in to collect and create deltas (see Section 4.3.5 for a detailed description of that concept).

The class `AbstractResourceDeltaCollector` extends the framework's `AbstractDeltaCollector` and implements Eclipse's `IResourceVisitor` interface. That interface defines a visitor that descends resource trees. Resources are Eclipse's abstraction of projects, folders, and files that a user is working with. Thus, the new abstract collector is comparable to the framework's `FileDeltaCollector`.

The concrete collector class `LocalHistoryDeltaCollector` extends the `AbstractResourceDeltaCollector` and collects changes by visiting a resource tree and calling its `LocalHistoryDeltaCreator` for every visited resource. That creator accesses Eclipse's local history to create the changes for a given resource. It implements the new `IResourceDeltaCreator` interface; an interface comparable to the `IFileDeltaCreator`.

The core plug-in provides further an own abstraction for deltas on resources, an interface named `IResourceDelta`. That interface is comparable to the framework's `IFileDelta`, it is an representation of a change that alters a resource. The implementation `LocalHistoryDelta` defines a change that obtains its patch from the local history. It contains a patch computed by the framework's Difference component and can be applied to resources.

The architecture of the classes that enable DDCHANGE ECLIPSE to collect and to create changes from the local history follows closely the architecture as discussed in Section 4.3.5. The main differences: they descend a tree of resources instead of a tree of files, and they obtain the former version of the resources from the local history instead from a Subversion repository. As you can see, the implementation of that new feature is quite straightforward.

Using the local history instead of a Subversion repository has several advantages. First, every developer can use DDCHANGE ECLIPSE without the need to install or setup a (remote) repository. Second, the access to the local history is much faster than the access to the repository over a network. Even a remote repository in the local network (LAN) is slower than the local history. That feature speeds up the computing of the initial set of changes noticeably.

Incremental Compiler

DDCHANGE ECLIPSE does not use the framework's Ant compiler (see Section 4.3.5). The core plug-in implements an `IBuilder` that uses Eclipse's incremental compiler. That compiler is probably the fastest compiler available for the Java programming language that includes full support for the new features of J2SE 5.0 (see Section 4.6).

The new compiler has two main advantages. First, it is fast. Second, we can retain the user's settings for the project. These settings include source and target directories, JARs and class folders on the build path, compiler compliance level, and much more. If we would use the Ant compiler, we would have to translate all these settings—and some of them are unique to the Eclipse compiler. Thus, using Eclipse's compiler is probably the best, or even only, choice. Fortunately, the implementation of the new builder did not cost much effort. The class contains about 130 lines of source code including JavaDoc.

5.3 Results

One possible measurement for the effort that will be (or, was) required to develop a program is the SLOC (source lines of code). If we compare the SLOC required to develop the framework and the two plug-ins, it will tend to result in the conclusion that developing tools (in general, instances of the framework) on top of the framework requires much less costs than developing the framework—or even developing the different tools from scratch. The SLOC value (including comment lines, especially JavaDoc) of the DDCHANGE framework is about 34.000, of DDCHANGE MAVEN and DDCHANGE ECLIPSE in each case about 8.000. Thus, the framework consists of about 4 times more SLOC than the two individual plug-ins (see Figure 5.16).

As we have seen in this chapter, we can concentrate on the concrete tool while developing it. In the case of DDCHANGE MAVEN, the framework supplies the whole infrastructure in order to determine failure-inducing changes. The Maven plug-in uses the framework out-of-the-box, at the same time it provides various configuration parameters and remains therefore flexible. On the other hand, DDCHANGE ECLIPSE uses only the *core* of the framework and contributes a new builder and a new source for the changes. Moreover, it is delivered with a user interface; nevertheless, not much more SLOC were required to develop that plug-in. Using the *tip of the iceberg*, you can determine failure-inducing changes while supplying your own builder, tester, and the initial set of changes.

The two plug-ins complement one another. DDCHANGE ECLIPSE runs on the developer's machine, as an extension of her familiar IDE. As soon as a unit test fails, she can debug that failure using the new plug-in. DDCHANGE MAVEN can be integrated with the continuous build process. As soon as a failure occurs, the developer is notified about the failure *and* a possible cause.

Compared to the Maven plug-in, the Eclipse plug-in has some advantages. It is tightly integrated within the IDE, and because it uses the local history and a fast incremental compiler, it computes

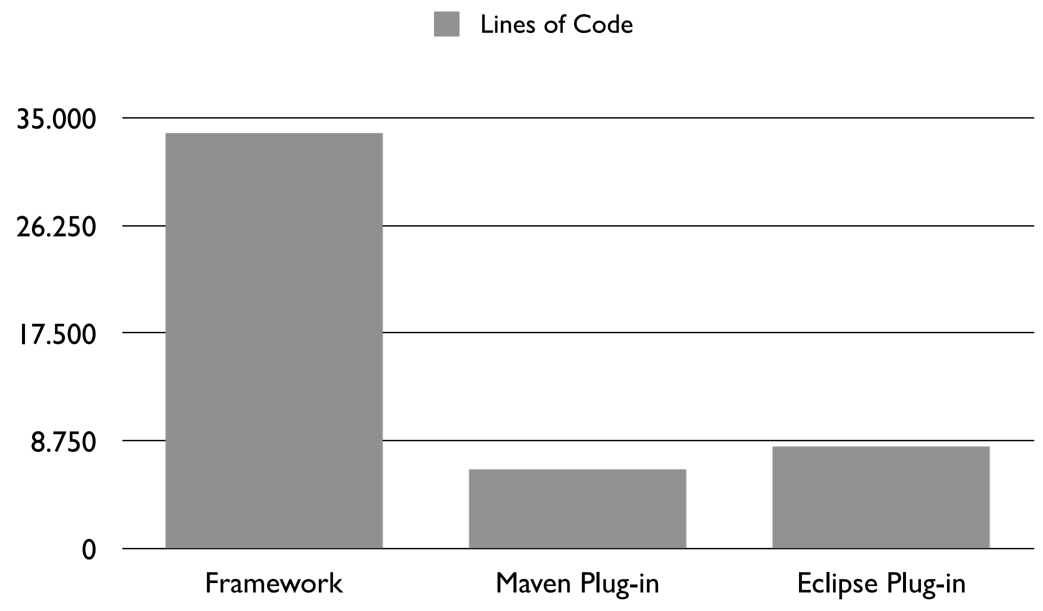


Figure 5.16: Bar chart: Comparison of the source lines of code required to develop the framework, the Maven plug-in, and the Eclipse plug-in.

the same result much faster. Another advantage of the usage of the local history is the simplicity related to the effort in order to install and setup the tool. The developer needs not to use a repository. He need just to install the plug-in, afterwards he can start using it. Using Eclipse's update manager and the provided update site, the installation process can be done with a few mouse clicks.

However, DDCHANGE ECLIPSE analyses mainly local changes unless the developer updates frequently hers local working copy. Once installed in the continuous build process (for example using CruiseControl), DDCHANGE MAVEN determines the failure-inducing changes regarding the committed changes of all developers, without the need for interaction with one of the developers. And because CruiseControl runs much faster on a server than on a workstation, the speed issue carries little weight. To make a long story short: DDCHANGE MAVEN can enrich the project's life cycle and its knowledge with automated debugging. Furthermore, it can add automated debugging to automated testing.

6 Conclusion

This chapter draws a general conclusion about the platform. The different chapters on the framework and on the tools contain more specific results as well as possible topics for further work.

The study has shown that automation of determining failure-inducing code changes at different stages in common development processes is enabled by using (1) a framework, (2) the delta debugging algorithm as main part of the strategy, (3) and two instances in form of plug-ins. Thus, developers can have a recourse in automated debugging without the need to adapt their familiar process—startup costs are low.

Since debugging of changes that introduced a failure is exhausting and time-consuming (as well as often frustrating), automating that debugging process may increase efficiency and improve productivity—the platform developed and presented in this work may save both time and money.

You can use the platform's core, the DDCHANGE framework, to develop with a modicum of effort new tools that enables to determine failure-inducing changes automatically. Furthermore, the platform contains innately two concrete tools. The Eclipse plug-in, DDCHANGE ECLIPSE, is able to determine these changes on local workstations, integrated with the programmer's familiar IDE. Complementary, the Maven plug-in, DDCHANGE MAVEN, can be integrated seamlessly with the continuous build process on a central server; it enriches automated testing with automated debugging. Using these tools, you can debug completely automatically many failures that occur in day-to-day business.

Because the two plug-ins are available for prominent development tools, a great many developers can use them. Furthermore, the plug-ins, as well as the framework, are released under an open-source license, allowing everyone to study, change, and improve the design.

Just as any other study, this one leaves some open issues and areas that lead to future work.

- We *act on the assumption* that the platform can save both time and money. However, we cannot currently prove that assumption. A evaluation could negate or confirm it. For instance, we could use WEB.DE's issue tracker and the Subversion repository in order to acquire information about fixed bugs. If we know the amount of time spent for the correction of a bug, we can use DDCHANGE to determine the failure-inducing changes and check whether the computed result pinpoints the bug faster, or not.

- The platform determines the failure-inducing changes. We could optimise that process in different ways in order to speed it up. First, the restriction of the search space promises a reduction of the runtime. Second, we could conduct the tests in parallel on several machines.
- We could use other tests than unit tests to debug other types of failures. For example, we could use acceptance tests or debug memory leaks. Memory leaks are often especially hard to debug, automated debugging of such failures would improve further the debugging process.

This work can lead to intense simplifications in the common development process—remember Occam's Razor: "Keep it simple".

A Appendix

A.1 Included CD

Enclosed you will find a CD that contains additional documents. The most interesting parts may be the JavaDoc API documentation of the framework, the complete source code of the framework and the tools, the movies that show the tools in action, and, last but not least, the framework and tools ready for use.

- *Documentation site for the framework.* The project information was generated by Maven from the project's sources and the metadata about the project. That information is provided as a web site. It is browsable off-line direct from the CD. The following reports are contained in the site:
  - *Changes.* A changes report showing the differences between different releases of the framework.
  - *Checkstyle.* A Checkstyle report so code violations can easily be found and corrected. Checkstyle implements a static code analysis in order to help programmers write Java code that adheres to a coding standard. See the website of Checkstyle [Tea05a] for more information about that tool.
  - *Clover.* Report of Clover, a code coverage analysis tool. It discovers sections of code that are not being adequately exercised by the unit tests. See the website of Clover [Ltd05a] for more information about that tool.
  - *FindBugs.* FindBugs is a tool to find bugs in Java programs. It looks for instances of so-called *bug patterns*—code instances that are likely to be errors. See the website of FindBugs [Hov05] for more information about that tool.
  - *JavaDocs.* The JavaDoc API documentation in HTML format. That comprehensive documentation of the framework contains public, protected, and private packages, classes, interfaces, fields, and methods.
  - *JavaDoc Report.* Report on the generation of JavaDoc.
  - *JavaDoc Warnings Report.* Formatted report of JavaDoc warnings.
  - *Unit Tests.* JUnit report based on the framework's unit tests. See the website of JUnit [GB05] for more information about that tool.

  - *Source Xref.* A set of browsable cross-referenced sources.
  - *Test Xref.* A set of browsable cross-referenced test sources.
  - *Project License.* Displays the primary license for the project.
  - *Simian Report.* Simian (Similarity Analyser) identifies duplication in Java source code. See the website of Simian [Ltd05b] for more information about that tool.

  The directory `maven-sites` contains the sites.

- *Tutorials on How to Use the Software.* Animated Macromedia Flash demonstrations that show DDchange Eclipse and DDchange Maven in action. See how the tools work even without installing them. Can be viewed with a Flash enabled browser.

  The directory `flash-movies` contains the movies embedded in simple HTML files.

- *Source Code.* All files required to build the framework and the tools.

  The directory `source-code` contains all sources, including the project metadata files and the Java sources.

- *Maven Repository.* A structured storage of the project artefacts. The local repository contained in the CD hosts all JARs and can be used by Maven to build the sources. Includes the Maven plug-in, too.

  The directory `maven-repository` hosts the local Maven repository.

- *Eclipse Update Site.* DDchange releases for Eclipse 3.1.x are hosted at the update site. Add this local update site as “Local Site” in Eclipse’s update manager (which you can find in the Help menu).

  The directory `eclipse-site` hosts the local Eclipse update site.

- *PDF Version of Thesis.* Electronic version of this work as Portable Document Format (PDF).

  The directory `thesis` contains the PDF file.

A.2 Subversion Repository

The repository located on `https://devel.netbeyond.de/svn/mburger/university/DDchange/` stores the different projects. In the following you will find a short description of these projects.

- *ddalgorithm.* The implementation of the delta debugging algorithm as described in Section 4.2.

- *ddchange-bughunting.* A project used for bug hunting. Contains code that reproduces reported bugs.
- *ddchange-eclipse-core.* The core of DDCHANGE ECLIPSE. Provides the main functionality via so-called extension points. The user interface uses these extension points in order to interact with the core.
- *ddchange-eclipse-feature.* The DDCHANGE ECLIPSE feature. A feature bundles several plug-ins into one downloadable package. This feature is hosted by the DDCHANGE ECLIPSE update site (see below).
- *ddchange-eclipse-feature-plugin.* A plug-in used solely in order to brand the DDCHANGE ECLIPSE feature.
- *ddchange-eclipse-logging.* Eclipse plug-in based on [Mar04]. Adds configurable logging facility using Log4j.
- *ddchange-eclipse-ui.* The user interface of DDCHANGE ECLIPSE.
- *ddchange-eclipse-updatesite.* Update site project. The update site hosts the DDCHANGE ECLIPSE feature.
- *ddchange-framework-core.* DDCHANGE, the framework as described in Chapter 4. The Test component is hosted in the following project.
- *ddchange-framework-test.* The Test component of DDCHANGE as described in Section 4.3.4. This component is separated from the core project, because it contains the remote tester. These classes have to be in the classpath of the program to be debugged. To minimise the amount of additional classes, this component was separated.
- *ddchange-maven-plugin.* DDCHANGE MAVEN, the instance of DDCHANGE that utilise Maven.
- *ddchange-sample-commons-lang-2.1.* The slightly modified version of the "Apache Jakarta Commons Lang" project as used for the examples in Chapter 5.
- *ddchange-sample-project.* Contains small examples that were used during the development process of this work.

List of Figures

Bibliography

[AL04] John Arthorne and Chris Laffra. *Official Eclipse 3.0 FAQs*. Addison-Wesley Professional, 2004.

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung und Unternehmensmodellierung*. Lehrbücher der Informatik. Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.

[Bal01] Helmut Balzert. *Lehrbuch der Software-Technik - Software-Entwicklung*. Lehrbücher der Informatik. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2001.

[Bec02] Kent Beck. *"Test Driven Development: By Example"*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2002.

[Bou04] Philipp Bouillon. A Framework for Delta Debugging in Eclipse. Diploma thesis, Universität des Saarlandes, July 2004.

[Céd05] Chabanois Cédric. svnClientAdapter. `http://subclipse.tigris.org/svnClientAdapter.html`, 2005. [Online; accessed 2005-11-23].

[Com05a] The Professional Open Source Company. Hibernate. `http://www.hibernate.org/`, 2005. [Online; accessed 2005-11-23].

[Com05b] The Professional Open Source Company. Hibernate Documentation Overview. `http://www.hibernate.org/5.html`, 2005. [Online; accessed 2005-11-23].

[CR04] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins (Eclipse Series)*. Addison-Wesley Professional, 2004.

[dev05a] The Subversion developers. JavaHL. `http://svn.collab.net/repos/svn/trunk/subversion/bindings/java/javahl/`, 2005. [Online; accessed 2005-11-23].

[dev05b] The Subversion developers. Subversion. `http://subversion.tigris.org/`, 2005. [Online; accessed 2005-11-23].

[dt05a] Ant development team. Ant's Depend Task. http://ant.apache.org/manual/OptionalTasks/depend.html, 2005. [Online; accessed 2005-11-23].

[dt05b] Ant development team. Ant's Javac Task. http://ant.apache.org/manual/CoreTasks/javac.html, 2005. [Online; accessed 2005-11-23].

[dt05c] Ant development team. Apache Ant. http://ant.apache.org/, 2005. [Online; accessed 2005-11-23].

[dt05d] Jakarta Commons Pool development team. Jakarta Commons Pool Component. http://jakarta.apache.org/commons/pool/, 2005. [Online; accessed 2005-11-23].

[EC05] Technical Articles Eclipse Corner. Eclipse Public License (EPL) Frequently Asked Questions. http://eclipse.org/articles/index.html, 2005. [Online; accessed 2005-11-14].

[EHH$^+$05] Paul Eggert, Mike Haertel, David Hayes, Richard Stallman, and Len Tower. Diffutils - GNU Project. http://www.gnu.org/software/diffutils/diffutils.html, 2005. [Online; accessed 2005-11-23].

[End05] Tim Endres. jCVS. http://www.jcvs.org/, 2005. [Online; accessed 2005-11-23].

[FF05] Martin Fowler and Matthew Foemmel. *Continuous Integration*, 2005. [Online; accessed 2005-11-15].

[FJK05] Jeffrey Fredrick, Paul Julius, and Joris Kuipers. CruiseControl Home. http://cruisecontrol.sourceforge.net/, 2005. [Online; accessed 2005-11-15].

[Fou05a] Apache Software Foundation. Jelly: Executable XML. http://jakarta.apache.org/commons/jelly/, 2005. [Online; accessed 2005-11-28].

[Fou05b] The Eclipse Foundation. Eclipse Bug System Home Page. https://bugs.eclipse.org/bugs/, 2005. [Online; accessed 2005-11-24].

[Fou05c] The Eclipse Foundation. Eclipse project FAQ. http://www.eclipse.org/eclipse/faq/eclipse-faq.html, 2005. [Online; accessed 2005-11-14].

[Fou05d] The Eclipse Foundation. Eclipse's Batch Compiler. http://www.eclipse.org/jdt/core/howto/batch%20compile/batchCompile.html, 2005. [Online; accessed 2005-11-23].

[FS97] Mohamed Fayad and Douglas C. Schmidt. Object–Oriented Application Frameworks. *CACM*, 40(10), 1997. guest editorial.

[Gat05] Stuart D. Gathman. GNU Diff for Java. http://www.bmsi.com/java/#diff, 2005. [Online; accessed 2005-11-23].

[GB03] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley Professional, 2003.

[GB05] Erich Gamma and Kent Beck. JUnit, Testing Resources for Extreme Programming. `http://www.junit.org/`, 2005. [Online; accessed 2005-11-12].

[GBM03] David Gallardo, Ed Burnette, and Robert McGovern. *Eclipse in Action*. Manning, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[Gül05a] Ceki Gülcü. Log4j project. `http://logging.apache.org/log4j/docs/`, February 2005. [Online; accessed 2005-11-23].

[Gül05b] Ceki Gülcü. Taxonomy of class loader problems encountered when using Jakarta Commons Logging. `http://www.qos.ch/logging/classloader.jsp`, February 2005. [Online; accessed 2005-11-23].

[Ham97] Graham Hamilton. JavaBeans API Specification. `http://java.sun.com/products/javabeans/docs/spec.html`, 1997. [Online; accessed 2005-11-28].

[hDG05a] The hsqldb Development Group. HSQLDB. `http://hsqldb.org/`, 2005. [Online; accessed 2005-11-23].

[hDG05b] The hsqldb Development Group. HSQLDB Documentation. `http://hsqldb.org/web/hsqlDocsFrame.html`, 2005. [Online; accessed 2005-11-23].

[HGW05] Jason Huggins, Paul Gross, and Jie Tina Wang. Selenium. `http://selenium.thoughtworks.com/index.html`, 2005. [Online; accessed 2005-11-23].

[Hov05] David Hovemeyer. FindBugs, Find Bugs in Java Programs. `http://findbugs.sourceforge.net/`, 2005. [Online; accessed 2005-11-12].

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Joh02] Rod Johnson. *Expert One-on-One J2EE Design and Development*. Expert One-on-One. Wrox Press, Chicago, Illinois, USA, 2002.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[LDK05] Greg Luck, Brett Dargan, and Gavin King. ehcache. `http://ehcache.sourceforge.net/`, 2005. [Online; accessed 2005-11-23].

[Lew73] David K. Lewis. *Counterfactuals*. Blackwell, 1973.

[Lew86] David K. Lewis. *On the Plurality of Worlds*. Blackwell, Oxford, 1986.

[Ltd05a] Cenqua Pty Ltd. Cenqua Clover Code Coverage for Java. http://www.cenqua.com/clover/, 2005. [Online; accessed 2005-11-12].

[Ltd05b] RedHill Consulting Pty. Ltd. Simian: Similarity Analyser. http://www.redhillconsulting.com.au/products/simian/, 2005. [Online; accessed 2005-11-12].

[Mar04] Manoel Marques. Plugging in a logging framework for Eclipse plug–ins. http://www-128.ibm.com/developerworks/library/os-eclog/, September 2004.

[MM85] Webb Miller and Eugene W. Myers. A File Comparison Program. *Software—Practice and Experience*, 15(11):1025–1040, 1985.

[Mye86] Eugene W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1(2):251–266, 1986.

[Off04] Eclipse Management Office. Eclipse Public License (EPL) Frequently Asked Questions. http://www.eclipse.org/legal/eplfaq.html, 2004. [Online; accessed 2005-11-13].

[OTI01] Inc. Object Technology International. Eclipse Platform Technical Overview. http://eclipse.org/whitepapers/eclipse-overview.pdf, 2001. updated for Eclipse 2.1 in Feb. 2003.

[Pat01] Jim Patrick. Handling memory leaks in Java programs. http://www-128.ibm.com/developerworks/java/library/j-leaks/, February 2001.

[Pre96] Wolfgang Pree. *Framework Patterns*. SIGS Bks &, 1996.

[Pre99] Wolfgang Pree. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, chapter 16. John Wiley & Sons, 1999.

[RST$^+$04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[SA96] Raimund Seidel and Cecilia R. Aragon. Randomized Search Trees. *Algorithmica*, 16(4/5):464–497, 1996.

[SE04a] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 76–85, Boston, MA, USA, July 12–14, 2004.

[SE04b] David Saff and Michael D. Ernst. Continuous testing in Eclipse. In *2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, Spain, March 30, 2004.

[SHNM04] Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. *SWT/J-Face in Action : GUI Design with Eclipse 3.0 (In Action series)*. Manning Publications, 2004.

[SM05] Inc. Sun Microsystems. Core J2EE Patterns - Data Access Object. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html, 2005. [Online; accessed 2005-11-23].

[Sof05] TMate Software. JavaSVN. http://tmate.org/svn/, 2005. [Online; accessed 2005-11-23].

[SRRT05] Maximilian Stoerzer, Barbara Ryder, Xiaoxia Ren, and Frank Tip. Finding Failure-Inducing Changes using Change Classification. Research Report RC 23729, IBM, September 2005.

[Tea05a] Checkstyle Development Team. Checkstyle. http://checkstyle.sourceforge.net/, 2005. [Online; accessed 2005-11-12].

[Tea05b] Commons Documentation Team. Commons Lang. http://jakarta.apache.org/commons/lang/, 2005. [Online; accessed 2005-11-13].

[WE05] Larry Wall and Paul Eggert. patch - GNU Project. http://www.gnu.org/software/patch/patch.html, 2005. [Online; accessed 2005-11-23].

[Wik05a] Wikipedia. Counterfactual conditional—Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Counterfactual_conditional&oldid=26619523, 2005. [Online; accessed 2005-11-07; revision as of 13:59, 27 October 2005].

[Wik05b] Wikipedia. Diff—Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Diff&oldid=28914884, 2005. [Online; accessed 2005-11-23; revision as of 16:30, 21 November 2005].

[Wik05c] Wikipedia. Inversion of Control—Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Inversion_of_Control&oldid=28157452, 2005. [Online; accessed 2005-11-13; revision as of 00:00, 13 November 2005].

[Wik05d] Wikipedia. Occam's Razor—Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Occam%27s_Razor&oldid=27118139, 2005. [Online; accessed 2005-11-07; revision as of 00:12, 2 November 2005].

[Wik05e] Wikipedia. Software quality—Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Software_quality&oldid=27498022, 2005. [Online; accessed 2005-11-23; revision as of 02:58, 6 November 2005].

[Zel01] Andreas Zeller. *Using Delta Debugging, A short tutorial*, November 2001. [Online; accessed 2005-11-13].

[Zel05] Andreas Zeller. *"Why Programs Fail: A Guide to Systematic Debugging"*. Morgan-Kaufmann Publishers, Inc., San Francisco, California, USA, 2005.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure–Inducing Input. *Software Engineering*, 28(2):183–200, 2002.