JExample: Exploiting Dependencies Between Tests to Improve Defect Localization

Adrian Kuhn[1], Bart Van Rompaey[2], Lea Haensenberger[1], Oscar Nierstrasz[1], Serge Demeyer[2], Markus Gaelli[1], and Koenraad Van Leemput[2]

[1] Software Composition Group, University of Bern,
Neubrückstrasse 10, 3012 Bern, Switzerland
{akuhn,gaelli,oscar}@iam.unibe.ch,
lhaensenberger@students.unibe.ch
[2] University of Antwerp, Middelheimlaan 1, 2020 Antwerpen, Belgium
{bart.vanrompaey2,serge.demeyer,koen.vanleemput}@ua.ac.be

Abstract. To quickly localize defects, we want our attention to be focussed on relevant failing tests. We propose to improve defect localization by *exploiting dependencies between tests*, using a JUNIT extension called JEXAMPLE. In a case study, a monolithic white-box test suite for a complex algorithm is refactored into two traditional JUNIT style tests and to JEXAMPLE. Of the three refactorings, JEXAMPLE reports five times fewer defect locations and slightly better performance (-8-12%), while having similar maintenance characteristics. Compared to the original implementation, JEXAMPLE greatly improves maintainability due the improved factorization following the accepted test quality guidelines. As such, JEXAMPLE combines the benefits of test chains with test quality aspects of JUNIT style testing.

1 Introduction

A well-designed test suite should exhibit high coverage to improve our chances of identifying any defects. When tests fail, we want to quickly localize defects, so we want our attention to be focussed on the relevant failing tests to identify the root cause of the defect. However, when some part of the base-code gets changed, a small defect can cause a domino effect of multiple failing unit tests. This is a problem, because the person changing the code has no other option than to browse all failing unit tests to try and deduce a single root cause. This task can prove to be quite difficult when that person is unfamiliar with the test code that fails.

Dependencies between unit tests, the cause of this domino effect, have generated considerable controversy [6,12,4]. Common wisdom states that defect localization is improved by avoiding dependencies between tests, yet empirical evidence shows that latent dependencies exist anyway even in well-designed test suites [9]. This suggests that, despite the guidelines, dependencies between tests are inevitable.

In this paper we propose to improve defect localization by making dependencies between tests explicit. For example, a developer can declare that a `testRemove` test depends on the successful outcome of a `testAdd` test. Based on these depedencies, a testing framework can automatically determine a suitable order to run the tests, and to skip tests that depend on other failed tests. This setup prevents the domino effect of failing tests. We test this hypothesis by means of a case study in which we compare four implementations of the same test suite: a JEXAMPLE-based implementation and three alternative JUNIT-based implementations.

The contributions of this paper can be summarized as:

- We propose explicit test dependencies as solution for the domino effect in defect localization (e.g. in Chained tests),
- We introduce JEXAMPLE, an extension of JUNIT that uses annotations to declare explicit dependencies between test methods,
- We present empirical evidence that JEXAMPLE provides five times better defect localization than traditional JUNIT, without considerable degradation in performance, code size or duplication.

The remainder of the paper is structured as follows: after related work in Section 2, Section 3 introduces JEXAMPLE, illustrating the difference between chained test methods and conventional JUNIT test methods. Section 4 covers the case study, with Section 5 discussing the results and stating some concluding remarks.

2 Related Work

Many authors have been studying techniques to prioritize test cases, selectively execute regression tests or reduce test suites.

Kung *et al.* discuss a cost-effective selective regression testing approach after changes in an object-oriented program, by (i) determining the set of affected classes and (ii) prioritizing the testing of classes to minimize test stub construction [10]. A similar approach is followed by Wong *et al.*, first applying a modification-based selection technique followed by test set minimization (minimal selection preserving a coverage criterion) and prioritization (increasing cost per additional coverage) [19].

Rothermel *et al.* propose several techniques for prioritizing test cases with the goal of improving the rate of fault detection. Coverage and fault-detection ability, in various forms, are used to determine the test cases execution order [14]. Results show that all techniques improve the rate of fault detection compared to the standard, randomly ordered suite.

Stoerzer et al. automatically classify changes depending on the likelihood that they contribute to a test's failure [16], by monitoring test execution and addressing the local change history in Eclipse.

Gaelli *et al.* infer a partial order of unit tests corresponding to the coverage hierarchy of their sets of covered method signatures [8]. Their work shows that most tests either cover a superset of another test method's coverage or cover themselves a subset of another test, concluding that most tests implicitly depend on other tests. In case of test case failures, the developer is guided to tests which were found to be smallest in a previously stored hierarchy of a successful test run. In our *a priori* approach, letting developers explicitly link tests, we do not need a green running suite in the first place.

There exists a consensus about the following quality aspects of test code. When supporting fast and frequent code-test cycles, not only should a test run take minimal time [15,5], but detected defects should be communicated to the developer in an informative manner [6]. This implies that the link between the test error and the responsible unit under test is made explicit. In an evolving system, test code needs to be understood, reviewed and extended by team members. Moreover, knowing that (refactoring) operations to the production system potentially invalidate the corresponding tests, a test suite's code should be easy to understand and change [17]. As test code is typically not verified beyond reviewing (at 150 to 200 lines of code per hour [2]), tests are advised to be short and simple. *Test smells* are described as maintenance prone constructs that are specific for software test code [4,12], in addition to regular deficiencies such as code duplication. As such, they are to be avoided.

Dependencies between unit tests have generated considerable controversy [4,6,12]. As a motivating example to illustrate a test dependency, consider the test code in Listing 1: the unit under test is a simple `Stack` class, for which two test methods are given, `testPush` and `testPop`, both implemented to run independently of each other. However, as each of the two test methods *must* cover `Stack`'s `push` method (there is no pop without push), an implicit dependency between `testPop` and `testPush` is introduced: whenever `testPush` fails, `testPop` is likely to fail as well.

Listing 1. Implicit dependency between test methods

```
public class StackTest {
   private Stack stack;

   @Before
   public void setup() {
      stack = new Stack();
   }

   @Test
   public void testPush() {
      stack.push("Foo");
      assertEquals(false, stack.isEmpty());
      assertEquals("Foo", stack.top());
   }

   @Test
   public void testPop() {
      stack.push("Foo");
      Object top = stack.pop();
      assertEquals(true, stack.isEmpty());
      assertEquals("Foo", top);
   }
}
```

On one side of the controversy, detractors consider dependencies to be a form of "bad smell" in testing code. Van Deursen *et al.* use the term *Eager Test* to refer to a test method checking several methods of the object to be tested [4]. They say that dependencies between the enclosed tests make such tests harder to understand and maintain. Van Rompaey *et al.* provide empirical evidence to support this claim [18]. Fewster and Graham state that the efficiency benefit of long tests (where setup and tear-down is only performed once) is far outweighed by the inefficiency of identifying the single point of failure [6]. The xUnit family of testing frameworks, as exemplified by JUNIT, advises its users to avoid dependencies between tests. Test methods are supposed to be independent artifacts, sharing at most a (re-initialized) fixture constituting the unit under test. Meszaros uses the term *ChainedTests* to point to this test design [12], motivating that it may be a valid strategy for overly long, incremental tests. On the down side, the implicit fixture initialized by previous tests may impede the understandability of a single test. On the supporting side of test dependencies, we find testing frameworks such as TESTNG that provide support to define explicit dependencies between test methods and or test cases [3].

3 JExample in a Nutshell

In order to facilitate chained tests, JEXAMPLE extends JUNIT as follows: (i) test methods may return values; (ii) test methods may take arguments; and (iii) test methods may declare dependencies.

When using JEXAMPLE there is no need for fixtures or setup methods. Any test method M_0 may be used as a setup method, using its return value x as the fixture for its dependents. That is, JEXAMPLE takes the return value x of M_0 and passes it on as an argument to all methods that depend on M_0. Chained tests are related to the idea of example-driven testing, which states that fixture instances are valuable objects, and hence, to be reused and treated first-order first order by a testing framework [7].

When executing a standard JUnit test case (*e.g.,* Listing 1), the JUNIT framework executes `setup` before each test method, using a field to pass the fixture instance from setup to test methods. Considering this, we may say that `setup` creates an example instance, and that all other tests depend on this instance. Hence, we promote `setup` to become a test method with return value:

Listing 2. Promote fixture to test with return value

```
@Test
public Stack testEmpty() {
   Stack empty = new Stack();
   assertTrue(empty.isEmpty());
   assertEquals(null, empty.top()));
   return empty;
}
```

Note the assertions in the method body. As `setup` is now a proper test method we may even test the fixture before passing it on. Next, we rewrite `testPush` to depend on the result of `setup` using a `@Depends` annotation as follows:

Listing 3. Take another test's result as input value

```
@Test
@Depends("testEmpty")
public Stack testPush(Stack stack) {
   stack.push("Foo");
   assertFalse(empty.isEmpty());
   assert("Foo", empty.top());
   return stack;
}
```

When executing the test in Listing 3, the JEXAMPLE framework will first call `setup` in order to fetch its return value and then pass the return value as an argument to `testPush`. As this method might possible modify its argument, we must either clone the example instance before passing it on or call `setup` twice. The current implementation of JEXAMPLE does the former (of course only if `setup` succeeds, otherwise all dependents of `setup` are skipped anyway).

Next, we readdress Listing 1 to find deeper levels of dependencies, turning the test case into a graph of chained test methods. And indeed, there is a test method that implicitly depends upon `testPush`'s outcome: `testPop` cannot be exercised without pushing some element first. Hence we implement `testPop` as follows, avoiding the duplicate call to `push` by depending on `testPush`'s return value:

Listing 4. Avoid code duplication using dependencies

```
@Test
@Depends("testPush")
public Stack testPop(Stack stack) {
   Object top = stack.pop();
   assertEquals(true, empty.isEmpty());
   assertEquals("Foo", top);
   return stack;
}
```

Given a defect in `Stack`'s `push` method, the `pop` test is ignored by JEXAMPLE, thereby pointing precisely to the defect location.

4 Case Study

In this section we report on a case study that compares four different implementations of the same unit test suite. The goal is to check how Chained tests improve defect localization, and how JEXAMPLE promotes quality criteria related to performance, size and code duplication.

The (pre-existing) JUnit test suite under study exercises an implementation of the *Ullmann subgraph isomorphism* algorithm — *i.e.,* an algorithm to compare the structure of graphs. This set of rigorous, white box tests was written to verify the core of a research tool as well as the interaction with a third party graph library.

Ullman Original is the original implementation of the case-study. Figure 1 illustrates how the test suite is implemented as a single test case consisting of six very long test methods. The test methods concentrate on a *growing unit under test* and are hence implemented as an alternating series of initialization and assertion code, entangling fixture and test code.

We refactored this original test suite implementation to three alternatives (i) best practice JUNIT tests; (ii) JUNIT using test case inheritance; and (iii) Chained tests using JEXAMPLE. The goal of this refactoring was to obtain equivalent implementations of the same test suite using different test design styles.

The *Ullmann JUnit-style (UJ)* implementation follows the original JUnit test guidelines as described by Beck and Gamma [1]. Tests on the same unit under test are grouped together, by sharing fixture objects and a setup method. To apply this style

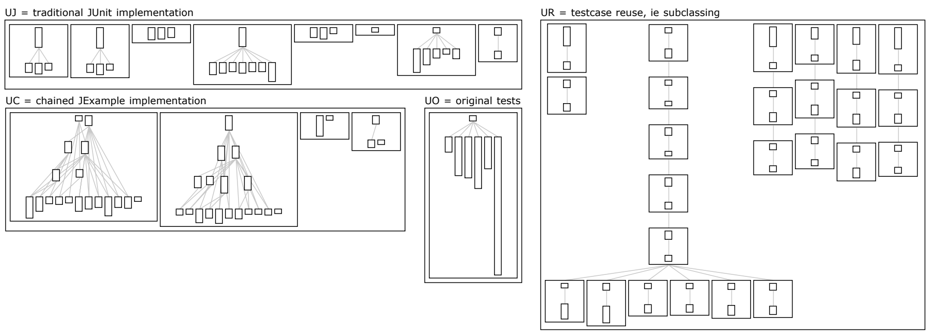
UJ = traditional JUnit implementation
UR = testcase reuse, ie subclassing
UC = chained JExample implementation
UO = original tests

Fig. 1. Polymetric view of the four alternative test suite implementations: The innermost boxes represent test methods (including setup methods); the height of the boxes shows the method's LOC (lines of code) metric; edges show test dependencies. The enclosing boxes represent test cases; edges show test case inheritance.

to the Ullmann case-study, the original test suite is split into eight test cases that each focuses on a different snapshot of the growing unit under test.

The *Ullman test case Reuse (UR)* implementation relies on test case subclassing to build a set of chained yet isolated test cases. This implementation uses a specific subclassing pattern, turning each iterative initialization step of the original test into a subclass that calls `super.setUp()` in its `setup` method to reuse previous initialization code. As such, test dependencies are specified by the test case inheritance hierarchy.

Ullmann Chained JExample-Style (UC), finally, introduces explicit dependencies using JExample. Mirroring the iterative initialization code of the original test methods, and using the dependency mechanism presented in Section 3, a root test method creates an example instance of an empty graph object, and passes the instance on to dependent methods. The dependent methods extend the example instance a bit, check some assertions, and eventually pass the instance on to another level of dependent instances, and so on. Dependencies between methods are declared by the developer using `@Depend` annotations, whereas passing on a method's return values to its dependents is done by the framework while running the test suite.

Figure 1 presents the test design of the four implementations using polymetric views [11]. The innermost boxes represent test methods (including setup methods). The height of these boxes show the method's SLOC (source lines of code) metric, while edges show test dependencies. The enclosing boxes represent test cases; edges show test case inheritance.

4.1 Evaluation Procedure

To evaluate *defect localization*, we measure the number of reported test case failures after randomly introducing defects in the system's code with a mutation testing tool (Jester [13]). Using the explicit dependencies between tests, JExample ignores tests that depend on a failed test. As such, we expect the UC implementation to report fewer failures than the other three implementations.

Besides defect localization, we select *test suite run-time performance*, *test size* and *code duplication* from the set of common test quality criteria. To quantify the adherence to these criteria, we use the following set of metrics:

– *Performance*. We measure the execution time of each implementation. To measure execution of test suites with failures, we use the created mutations.
– *Size*. We calculate the overall size (source lines of code) of the test suite as well as the number and size of test methods. While the former tells us something about the code base as a whole that needs to be maintained, the latter identifies how well the test suite is factorized.
– *Duplication*. We measure the amount of duplication[1] in each implementation as a result of the presence or absence of reuse possibilitities.

For comparison reasons, we control a couple of test suite equality factors. First, we ensured that all four implementations exhibit the same coverage, being 96.9% (Java) instruction coverage (measured using Emma[2]). Secondly, we aimed to keep the same number of assertions. Ultimately, slight differences appeared (between 81 and 85 asserts) due to varying reuse opportunities.

4.2 Results

Defect localization. In order to quantify the traceability quality of the four implementations, we created eight scenarios, named MUT1-MUT8, where a single mutation in the Ullmann code causes the tests to fail. Jester changes constants in the code and adds clauses in boolean conditions to test the defect detection strength of a test suite. For each mutation scenario, we then measure the number of failures JUnit reports. Knowing that only one mutation has been introduced at a single location, ideally only a single failure should be reported.

Table 1. Number of failures: absolute number/ignored tests (relative number)

	UO	**UJ**	**UR**	**UC**
MUT1	4 (66%)	12 (46%)	14 (52%)	2/12 (6%)
MUT2	2 (33%)	2 (8%)	2 (7%)	2/0 (4%)
MUT3	1 (17%)	10 (38%)	9 (37%)	1/12 (3%)
MUT4	1 (17%)	1 (4%)	1 (4%)	1/0 (3%)
MUT5	1 (17%)	10 (38%)	9 (37%)	1/12 (3%)
MUT6	1 (17%)	1 (4%)	1 (4%)	1/1 (3%)
MUT7	1 (17%)	10 (38%)	9 (37%)	1/12 (3%)
MUT8	4 (66%)	11 (42%)	9 (37%)	2/14 (4%)

Table 2. Average execution time (in seconds) of 30 test runs

	UO	**UJ**	**UR**	**UC**
SUCC	0.512	0.600	0.747	0.554
MUT1	0.506	0.603	0.753	0.539
MUT2	0.516	0.609	0.753	0.554
MUT3	0.500	0.597	0.748	0.534
MUT4	0.516	0.600	0.751	0.556
MUT5	0.499	0.594	0.752	0.533
MUT6	0.513	0.610	0.747	0.547
MUT7	0.502	0.600	0.755	0.537
MUT8	0.515	0.613	0.754	0.546

Table 1 presents the failures reported during test runs on the mutated Ullmann code in absolute numbers as well as a percentage of the number of tests. For UC, we add the number of tests ignored by JEXAMPLE. The results show that for a single mutation,

[1] Using CCFinderX — http://www.ccfinder.net
[2] http://emma.sourceforge.net

typically multiple tests fail. In the case of the original implementation, the number of failures varies between 1 and 4 (out of 6 test methods). For UJ and UR however, up to 14 tests fail as a result of the factorization of test methods. Due to the ordered test execution of the UC implementation, at most 2 tests fail during a test run, while up to 14 are skipped. Overall, test runs on the UJ and UR implementations report five times more defect locations than the JEXAMPLE tests. The original implementation, however, only reports about 36% more defect locations.

Performance. Table 2 lists the average execution time for the four test suite implementations, and for 5 scenarios. In the first scenario called SUCC(ESS), we execute the tests on the original Ullmann implementation. In the four cases we reuse the mutations of Ullmann created earlier. The results are collected as the average execution time of 30 test runs, measured with the UNIX *time* command on an Intel Pentium 4, 3 Ghz, 1 Gb Ram, Sun JDK 1.6.0, JUnit 4.3.1.

For the successful test run on the original implementation of the Ullmann algorithm, we observe that the original test suite implementation has the fastest average, followed by UC, UJ and UR. We apply Student's t-test to verify whether there exists a significant difference between the test execution time sample sets. As a result, we can indeed conclude — with a confidence of 95% — that UC is 7-8% slower than UO, yet 8-12% faster than UJ and 35-41% faster than the UR approach. The results do only indicate significant performance increases – compared to the SUCCESS scenario – for mutations where 12 to 14 tests are skipped in the UC implementation.

Size. The original Ullmann test implementation, lacking any form of encapsulation for individual tests or set-up, is the most concise one. As a consequence, the UJ and UC implementations are 77% and 87% larger. UR is even 2.4 times as large.

Due to explicit set-ups and fixtures for the multiple test cases the original Ullmann test case (NOTC — number of test cases) has been refactored into, and the method header code for many test commands, the alternative implementations are better factored, as Table 3 shows. The average test method length has dropped from close to 40 to around 10, while the number of assertions per test method decreased from 14 to below 4.

Duplication. To evaluate the level of code duplication, we used CCFinderX to calculate and report code clones. Table 4 summarizes the results. The original implementation contains the least duplication, with 266 tokens (out of 3446) involved in any code clone. The alternative implementations contain between 4200 and 4500 tokens, 13 to 22% of

Table 3. Size of the four implementations expressed in Source Lines Of Code (SLOC), Number Of Test Cases (NOTC), Number Of Test Setups (NOTS) and Number Of Test Methods (NOTM)

	UO	**UJ**	**UR**	**UC**
SLOC	311	551	735	582
NOTC	1	8	26	4
NOTS	1	8	25	0
NOTM	6	52	54	34

Table 4. Code clone results, configured with a minimum clone token size of 50, soft shaper and p-match in CCFinderX

	#tokens	**% tokens**
UO	266	7.7%
UJ	597	13%
UR	938	22%
UC	780	18%

which is listed in code clones. UC contains about 5% more code covered by clones than UJ.

5 Discussion and Conclusion

In this paper we introduced the idea of making dependencies between tests explicit to improve defect localization. We proposed JEXAMPLE, an extension of JUNIT that allows the tester to annotate test methods with its dependencies. In a case study, JEXAMPLE is compared to more traditional JUNIT-style tests.

The case study showed that compared to alternative test suite implementations, JEXAMPLE tests indeed exhibit an improved defect localization. Moreover, such test suites execute faster and contain less code than traditional JUNIT tests. Compared to a test design style consisting of monolithic test methods entailing long chains of tests, JEXAMPLE tests run somewhat slower and contain some more source code, but rely upon good unit testing practices of encapsulated, concise test methods to ensure maintainability. JEXAMPLE thus combines the best of both worlds: it exhibits the benefits of test chains with the test quality aspects of JUNIT style testing.

There exist a couple of open challenges to consider regarding maintenance of chained tests. First, the dependencies between tests have to be indicated by the developers. Forgetting to do so, or introducing wrong dependencies leads to potentially more failures for a single defect. In the future, automated support to track such dependencies might alleviate this effort. Secondly, with JEXAMPLE, the concepts of a fixture and a set-up become implicit, rendering their identification harder.

Being an extension of JUNIT, JEXAMPLE tests can co-exist with regular tests. Moreover, the migration process merely consists in adding dependency and parameter-passing annotations, as well as cloning the passed-on objects. Our evaluation showed that JEXAMPLE-style tests are especially useful for expressing long test chains as well as for unit tests with obvious dependencies in test suites.

Reproducible Results Statement: A prototype of JEXAMPLE, as well as all four scenarios of the case-study are available for download at: `http://scg.unibe.ch/Resources/JExample`

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Analyzing, Capturing and Taming Software Change” (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008), and IWT Flanders, in the context of the ITEA project if04032 entitled “SERIOUS: Software Evolution, Refactoring, Improvement of Operational & Usable Systems” (Eureka $\sum$ 2023 Programme).

References

1. Beck, K., Gamma, E.: Test infected: Programmers love writing tests. Java Report 3(7), 51–56 (1998)
2. Belli, F., Crisan, R.: Empirical performance analysis of computer-supported code-reviews. In: Proceedings of the 8th International Symposium on Software Reliability Engineering, pp. 245–255. IEEE Computer Society, Los Alamitos (1997)

3. Beust, C., Suleiman, H.: Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley, Reading (2007)
4. Deursen, A., Moonen, L., Bergh, A., Kok, G.: Refactoring test code. In: Marchesi, M. (ed.) Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP 2001), University of Cagliari, pp. 92–95 (2001)
5. Feathers, M.C.: Working Effectively with Legacy Code. Prentice-Hall, Englewood Cliffs (2005)
6. Fewster, M., Graham, D.: Building maintainable tests. In: Software Test Automation. Ch. 7. ACM Press, New York (1999)
7. Gaelli, M.: Modeling Examples to Test and Understand Software. PhD thesis, University of Berne (November 2006)
8. Gaelli, M., Lanza, M., Nierstrasz, O., Wuyts, R.: Ordering broken unit tests for focused debugging. In: 20th International Conference on Software Maintenance (ICSM 2004), pp. 114–123 (2004)
9. Gaelli, M., Nierstrasz, O., Ducasse, S.: One-method commands: Linking methods and their tests. In: OOPSLA Workshop on Revival of Dynamic Languages (October 2004)
10. Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., Kim, Y.-S., Song, Y.-K.: Developing and oject-oriented software testing and maintenance environment. Communications of the ACM 38(10), 75–86 (1995)
11. Lanza, M., Ducasse, S.: Polymetric views—a lightweight visual approach to reverse engineering. Transactions on Software Engineering (TSE) 29(9), 782–795 (2003)
12. Meszaros, G.: XUnit Test Patterns - Refactoring Test Code. Addison-Wesley, Reading (2007)
13. Moore, I.: Jester — a JUnit test tester. In: Marchesi, M. (ed.) Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP 2001), University of Cagliari (2001)
14. Rothermel, G., Untch, R., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. Transactions on Software Engineering 27(10), 929–948 (2001)
15. Smith, S., Meszaros, G.: Increasing the effectiveness of automated testing. In: Proceedings of the Third XP and Second Agile Universe Conference, pp. 88–91 (2001)
16. Stoerzer, M., Ryder, B.G., Ren, X., Tip, F.: Finding failure-inducing changes in java programs using change classification. In: Proceedings of the 14th SIGSOFT Conference on the Foundations of Software Engineering (FSE 2006) (November 2006)
17. Deursen, A.V., Moonen, L., Zaidman, A.: On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. In: Software Evolution. Ch. 8. Springer, Heidelberg (2008)
18. Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M.: On the detection of test smells: A metrics-based approach for general fixture and eager test. Transactions on Software Engineering 33(12), 800–817 (2007)
19. Wong, W.E., Horgan, J.R., London, S., Agrawal, H.: A study of effective regression testing in practice. In: Proceedings of the Eighth International Symposium on Software Reliability Engineering, November 1997, pp. 230–238 (1997)