Delft University of Technology
Software Engineering Research Group
Technical Report Series

Measuring Test Case Similarity to Support Test Suite Understanding

Michaela Greiler, Arie van Deursen and Andy Zaidman

Report TUD-SERG-2012-008

TUD-SERG-2012-008

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS), 2012, Springer.

Measuring Test Case Similarity to Support Test Suite Understanding

Michaela Greiler, Arie van Deursen and Andy Zaidman

Delft University of Technology, The Netherlands
{m.s.greiler, arie.vandeursen, a.e.zaidman}@tudelft.nl

Abstract. In order to support test suite understanding, we investigate whether we can automatically derive relations between test cases. In particular, we search for trace-based similarities between (high-level) end-to-end tests on the one hand and fine grained unit tests on the other. Our approach uses the shared word count metric to determine similarity. We evaluate our approach in two case studies and show which relations between end-to-end and unit tests are found by our approach, and how this information can be used to support test suite understanding.

1 Introduction

Modern software development practice dictates early and frequent (automated) testing. While automated test suites are helpful from a (continuous) integration and regression testing perspective, they lead to a substantial amount of test code [16]. Like production code, test code needs to be maintained, understood, and adjusted upon changes to production code or requirements [8, 10, 13].

In light of the necessity of understanding and maintaining test suites, which can become very costly due to the large amounts of test code, it is our stance that tool support can reduce the burden put on the software and test engineers. The V-model from Figure 1 shows that different levels of tests validate different types of software artifacts, with each level contributing to the large amount of test code. Figure 1 also shows that, ideally, requirements can be *traced* all the way to source code, making it easier to perform *impact analysis*, i.e., determining what the impact of a changing requirement is on the source code. The right side of the V-model however, the test side, does not have similar tool support.

In this paper we propose to support engineers by helping them to understand relationships between different types of test suites. As an example, an automated test suite can include "end-to-end" tests, exercising an application from the user-interface down to the database, covering functionality that is meaningful to the end user.[1] The test suite will typically also include dedicated unit tests, aimed at exercising a very specific piece of behavior of a particular class. Suppose now a requirement changes, entailing a modification to the end-to-end test, which unit tests should the software engineer change as well? And vice-versa, if a unit test is changed, should this be reflected in an end-to-end test as well?

[1] We deliberately did not use the term acceptance test, as it is commonly associated with tests executed by the customers/users.

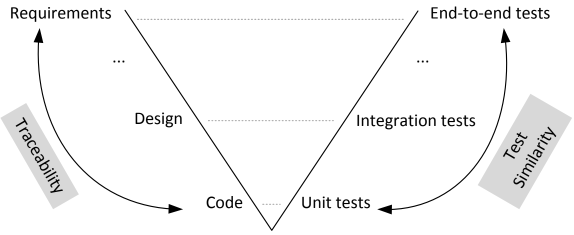

Fig. 1. The V-model for testing

Our goal is to develop an automated technique for establishing relations between test cases, in order to assist developers in their (test suite) maintenance activities. To reach this goal, we employ *dynamic analysis* [4]. We collect call traces from test executions, and use these to compute a *similarity* value based on the shared word count metric. The resulting technique, which we call *test connection mining*, can be used to establish connections between test cases at different levels. An implementation of our technique is available via a framework called the *Test Similarity Correlator*.

We evaluate the approach in two case studies, by elaborating on the usefulness of the approach to improve the understanding. We analyze how measuring similarity based test relations can help to (1) find relevant tests by showing test relationships, (2) understand the functionality of a test by describing high-level test cases with related unit test cases and (3) reveal blank spots in the investigated unit test suites.

This paper is structured as follows: in Section 2, we discuss our test execution tracing approach. In Section 3, we describe the similarity metrics we use to compare traces. Subsequently, we describe our approach and its implementation (Section 3) as well as the set-up of our case studies (Section 4). The two case studies are covered in Sections 5 and 6. We conclude with discussion, related work, and a summary of our contributions in Sections 7–9.

2 Tracing and Trace Reduction

Test connection mining first of all requires obtaining execution traces with relevant information of manageable size. This section describes the specific execution trace we use and the trace reduction techniques we apply.

2.1 Tracing Test Executions

Before the test run, the production and test code are instrumented. Subsequently, during the test run we obtain an execution trace comprised of various types of *events*: (1) *test execution* events represent the execution of a test method, (2) *set-up and tear-down* events mark the execution of a method which is either used for test set-up or tear-down, (3) *method execution* events signalling the execution of a public method within the code under test and (4) *exception thrown* events indicating that an exception has been thrown.

For the similarity measurements it is important to be able to distinguish between production and test code. Otherwise, executions of test helper methods will appear in the trace and render the test cases as less related. Due to the common practice to put the test code in a separate package (e.g., test.jpacman), we simply filter executions of methods belonging to test code out during instrumentation. If test and production code are within the same packages, test classes can be annotated and correctly addressed during instrumentation.

2.2 Handling mocks and stubs

When mocks or stubs are used, care has to be taken to correctly trace calls to mocked classes and map these calls to the corresponding original class.

A first issue is that using mocking frameworks can have the effect that an automatically created mock-object is not part of the instrumented package. For example, by using the JMock[2] library interfaces and classes defined to be mocked are automatically wrapped in a proxy class which is located in the same package as the class directing the mocking operation, which will usually be the test class or a helper class within the test package. Because we do not trace executions of methods defined in the test package, these classes have to be addressed specifically. We do so by keeping track of a list of mocked types.

Mocking also plays a role for tracing the production code, as the mocked and unmocked classes have to be mapped to allow identifying their similarity. Therefore, we have to indicate that a method on a mockable object has been invoked. To that end, we check whether the runtime type of the target is contained in the list of mocked classes. If yes, we further investigate whether the method intercepted is part of the mockable type, since a class implementing a mockable interface can have additional methods. Therefore, we derive recursively, via reflection, the set of methods belonging to this type including all methods defined by it and its (potential) super-types. Only if the method intercepted is an actual method of the mockable type, we discovered a mockery execution. As such, we add it to the trace and mark it with a *mockery mark*.

Finally, we need to neutralize those mock and stub calls. As illustrated in Listing 1.1, an execution of a method of a mocked type can be traced as the execution of an inner class within the (test) class defining the mock operation. As this differs from the trace events left behind by the execution of a method of the actual type, we render them as similar, by using the *mockery marks* set during tracing. Note also the actual type might differ from the mocked type by being the implementation of a type or extending a common type. We inspect the trace and replace all executions of methods of an actual type, as well as the executions of the mocked type by their common (super) type. For example, the traces in Listing 1.1 would be mapped to “void Sniper.join()”.

Listing 1.1. Trace differences with or without mocking

```
//Execution of method join of the mocked interface Sniper
void TestClass.$Proxy1.join()

//Execution of method join of class AuctionSniper implementing Sniper
void AuctionSniper.join()
```

[2] http://www.jmock.org

2.3 Trace reduction

Trace reduction techniques are important in order to reduce the size of traces to improve performance and scalability, and to help reveal the key functionality of the test case, e.g., by reducing common functionality or noise [3]. We adopt the following five reduction techniques before starting our analysis:

Language based Reduction. The traces are reduced to only public method executions and do not comprise any private or protected methods. Furthermore, only the production code is fully traced; for the test code only test method execution events are traced to be able to separate individual test cases from an entire test run.

Set-up and Tear-Down Reduction. As set-up and tear-down methods do not explicitly contribute to the specific focus of a unit test, and are usually shared by each test case within a test class, all method executions taking place during set-up or tear-down are removed.

Shared Word Reduction. This trace reduction focuses on helping identify the core functionality of a test case, by removing trace events that are omnipresent in almost all test traces (defined by a variable threshold).

Complement Reduction. This reduction focuses on reducing the trace size by removing calls within the trace of interest that are not existing in any of the test traces to compare to. Although, after such a reduction target traces will be calculated as more similar to the source trace, the reduction itself does not influence the information perceived useful for ranking the target traces with respect to each other.

Unique Set of Calls. This technique reduces all trace events within a trace to a unique set of events. Because information such as order and position of events are not preserved this reduction is only useful for similarity measurements that do not take such information into account.

3 Determining Similarity Measurements

The second step involved in test connection mining consists of computing trace similarities. In particular, we compute the similarity between a source trace te (e.g., from an end-to-end test) and a target trace tu (e.g., from a unit test).

As similarity metrics we compared (1) shared word count [14], (2) Levenshtein distance [12] and (3) pattern matching based on the Knuth-Morris-Pratt algorithm [12]. From an initial experiment we observed that all three metrics provided similar results, which is why we continue with the shared word count in the remainder of this paper.

The shared word count measurement [14] assesses the number of tracing events that two test execution traces have in common. The similarity between a source trace and a target trace is calculated as the number of tracing events comprised in both test traces.

3.1 Relevancy support based on occurrence

Some tests are related to other tests, because they test common functionality. Using this piece of knowledge, we can improve our results, by marking these more general tests as less important. Vice versa, by giving a test appearing less often a high impact, results with more specific functionality are ranked higher. We do so by multiplying the similarity measurement for a specific trace tu with the total number of test cases it has been compared to, and dividing this by the number of times the trace appeared as a result. We also use the average similarity of test case tu_i to rank similar results. For example, if target test cases tu_1 and tu_2 have the same similarity with te, than the test case with the smaller average similarity among all te_j is ranked first.

3.2 Implementation

We implemented the various trace reduction techniques and similarity measurements presented in this paper in a Java based framework called *Test Similarity Correlator*.[3] Our tool offers an API to steer customized test similarity measurements, varying in trace reduction, thresholds and similarity calculations.

To instrument the test execution we use the AspectJ[4] framework. We offer three different annotations to facilitate tracing of execution of test-methods, set-up and tear-down methods. *Test Similarity Correlator* comprises several aspects, addressing join points to weave in our tracing advices, including the aspect to address code generated by the mocking library JMock.

4 Set-Up for Case studies

4.1 Research Questions

To evaluate the usefulness of test connection mining, we conducted an explorative study based on two case studies. In these case studies, we aim at answering the following questions:

RQ1 How do the associations found by test connection mining relate to associations a human expert would establish?
RQ2 Are the associations found useful for a typical test suite understanding task, i.e., getting familiar with the test suite of a foreign system?
RQ3 How does mocking influence the similarity measurements?
RQ4 What are the performance characteristics, both in time and in space, of the analysis conducted?

To answer these questions, we select a subject system that is shipped with both a functional test suite as well as a unit test suite. We manually compile a *conceptual mapping* between unit and functional test cases, and compare these to the mappings found through test connection mining automatically.

The first case study is used to assess RQ1 and RQ4, whereby in the second case study we focus on RQ2 and RQ3.

[3] `http://swerl.tudelft.nl/bin/view/Main/TestSimilarityCorrelator`
[4] `http://www.eclipse.org/aspectj`

4.2 Technique customization

The specific trace reduction configuration (see Section 2) we use in the case studies consists of the following steps.

Before calculating the trace similarity, traces are reduced by using the *Set-up and Tear-Down* reduction, followed by the *Shared Word*, and the *Complement* reductions. The order of the reduction is important and influences the ranking of the results. For example, if all unit test cases call a method "quit()" as part of their tear down method, but only one unit test actually uses this method during test execution, the application of first the *Shared Word* reduction and then the *Set-up and Tear-Down* reduction would eliminate this call from the trace. The *Shared Word* reduction technique can be customized by a threshold influencing how many traces must comprise a trace event before it is removed.

For similarity measurements based on shared word count, which does not take the order of events into account, the traces are further reduced to their unique set of events.

5 Case Study I: JPacman

As first subject system we use JPacman,[5] a simple game written in Java inspired by Pacman, used for educational purposes at Delft University of Technology since 2003. Key characteristics of JPacman are listed in Figure 2.

JPacman follows the model-view-controller architecture. Each class in the model package comes with a (JUnit) unit test class, which together comprise 73 unit test cases. The test suite makes use of several test patterns described by Binder [1], using state machines, decision tables, and patterns such as *polymorphic server test* (reusing superclass test suites at the subclass level). This results in a line coverage of 90% in the model package, as measured by Cobertura.[6]

The functional test suite is directly derived from a set of JPacman user scenarios written in *behavior-driven development*[7] style. These scenarios are of the form given in Listening 3. There are 14 such scenarios, each of which is translated into a JUnit test case. The resulting functional test cases exercise around 80% of the user interface code and around 90% of the model code.

5.1 Obtaining the Conceptual Mapping

JPacman's main developer created a conceptual mapping in advance. The key criterion to identify a relation between an end-to-end test t and a unit test u was

[5] Version 4.4.4, dated October 2011. JPacman can be obtained for research and educational purposes from its author, 2nd author of this paper.

[6] http://cobertura.sourceforge.net/

[7] http://dannorth.net/whats-in-a-story/

Code size (lines)	4,000
Test code size (lines)	2,000
No of classes	26
No of test classes	16
No of unit tests	73
No of functional tests	14

Fig. 2. JPacman characteristics

```
Given [context]
  And [some more context]...
When  [event]
Then  [outcome]
  And [another outcome]...
```

Fig. 3. JPacman Test Scenarios

the question whether the behavior of u is important in order to understand the behavior of t. The conceptual mapping contains both *positive* incidences (important connections to be established) and *negative* ones (unlikely connections that would be confusing). In most end-to-end (ETE — numbered from *ETE01* to *ETE14*) test cases, we had at least 5 positive and 9 negative connections.

While the mapping obtained can be considered a useful baseline, it should be noted that it is *incomplete*: it only identifies clearly connected and clearly disconnected test pairs. The remaining tests are categorized as *undecided*. Furthermore, we tried to be as specific as possible: relations to "infrastructure" unit test cases relevant to many end-to-end tests were not included.

5.2 RQ1: Comparison to Conceptual Mapping

We used a spreadsheet containing 14 × 73 matrices to study the differences between the conceptual mapping as well as the ones obtained through our automated analysis. Due to size restrictions, we can not show all results of the measurements[8]. Besides saving space, showing the top 5 results is also realistic from a practical point of view, because in practice a user of the tool would also look primarily at the highest ranked results. In Table 1 we show for each end-to-end test the 5 most similar unit tests based on the shared word count metric. A ranking is indicated as *optimal* in case it is marked as highly related in the conceptual mapping and it is ranked high (i.e. top match). Incidences marked as related by the expert which are high ranked are evaluated as *good*. Results of the category undecided are subjected to additional manual analysis: the results are indicated as *ok* only if the relation is strong enough to be justified, and labeled as *nok* otherwise. Unrelated results ranked highly, as well as (highly) related results ranked low, are also evaluated as *nok*.

The overall impression is that the automated analysis provides a useful approximation of the manually obtained mapping. Looking at all the results for each end-to-end test case, we found that:

- For all but one end-to-end test (i.e. *ETE02*), the top match is ranked as the first or second automatically calculated result.
- Within the top 10 results only one unit test case marked unrelated is listed.
- All remaining results ranked within the top 10 (i.e. from the undecided category) are sufficiently related to the end-to-end tests to justify investigation.
- No relations are missing as all test cases marked as relevant by the expert have been identified as related. Thereby, 80% of all test cases marked as related have been ranked within the upper half of the results *showing similarity* and within the top 30% of overall results.
- 92% of all tests marked as unrelated correctly map to *no similarity* by the measurements. The remaining unrelated tests revealed weak connections and have been ranked in the bottom half of the results, except for one test (14).

[8] The complete results are available at `http://swerl.tudelft.nl/twiki/pub/Main/TestSimilarityCorrelator/similarityResults.zip`

Test Case (no.)	match	Test Case (no.)	match	Test Case (no.)	match
1 Move to empty cell & undo		2 Move beyond border		3 Move to wall	
MovePlayer (23)	optimal	FoodMove (44)	ok	DxDyImpossibleMove (15)	optimal
UndoEmptyMove (17)	optimal	FoodMoveUndo (39)	ok	SimpleMove (22)	good
UndoDxDy (18)	optimal	UndoFoodMove (19)	ok	DieAndUndo (26)	optimal
UndoFoodMove (19)	ok	PlayerWins (24)	ok	DieAndRestart (25)	optimal
Apply (38)	optimal	wonSneakPaths (35)	ok	MovePlayer (23)	good
4 Eat food & undo		5 Win and restart		6 Get killed and restart	
FoodMoveUndo (39)	optimal	SetUp (12)	ok	DieAndRestart (25)	optimal
UndoFoodMove (19)	optimal	PlayerWins (24)	optimal	PlayerWins (24)	ok
UndoFood (47)	optimal	FoodMoveUndo (39)	good	wonSneakPaths (35)	ok
FoodMove (44)	optimal	FoodMove (44)	optimal	Updates (37)	ok
MovePlayer (23)	good	DxDyPossibleMove (14)	ok	UndoFoodMove (19)	ok
7 Monster to empty cell		8 Monster beyond border		9 Monster to wall	
UndoMonsterMove (16)	optimal	Wall (70)	optimal	EmptyCell (69)	optimal
MoveMonster (28)	optimal	MonsterPlayer (73)	ok	MonsterFood (72)	ok
Updates (37)	ok	MonsterFood (72)	ok	MonsterPlayer (73)	ok
OutOfBorder (68)	ok	EmptyCell (69)	optimal	Wall (70)	optimal
FoodMove (71)	ok	MonsterKillsPlayer (27)	ok	MonsterKillsPlayer (27)	ok
10 Monster to food		11 Monster to player		12 Suspend	
MoveMonster (28)	optimal	MonsterPlayer (73)	optimal	SuspendRestart (29)	optimal
Updates (37)	ok	70 Wall	ok	Start (21)	good
Apply (66)	good	MonsterFood (72)	ok	SneakPlaying (33)	ok
FoodMoveUndo (67)	optimal	EmptyCell (69)	good	SuspendUndo (30)	optimal
FoodMove (71)	ok	MonsterKillsPlayer (27)	optimal	SneakHalted (36)	good
13 Die and Undo		14 Smoke			
DieAndUndo (26)	optimal	SetUp (12)	good		
MovePlayer (23)	good	PlayerWins (24)	optimal		
wonSneakPaths (35)	ok	FoodMoveUndo (39)	ok		
SimpleMove (22)	ok	wonSneakPaths (35)	ok		
DieAndRestart (25)	optimal	FoodMove (44)	ok		

Table 1. Top 5 ranked unit tests per end-to-end test for JPacman

Correct Identifications. *Top matches.* The top two results of the measurements in most cases also contain the top match for an end-to-end test case. For example, the end-to-end test involving a keyboard event to move the player to the right and then undoing that move (*ETE01*), is connected to a unit test actually moving the player. As another example, *ETE03* attempts to move through a wall (which is impossible), and is connected to a unit test addressing the correct positioning of the Pacman's mouth after attempting an impossible move. As dying is a type of an impossible state, connections to dying are also correct.

Moving Monsters vs. Players. Some groups of test cases are moving players (i.e. 44, 45, 46), whereas other tests (72, 73, 74) are moving monsters. In the conceptual mapping, tests moving players are related to ETE tests 1-6, and marked as unrelated for ETE tests 7-11, whereby tests moving players are related the opposite way. These relations respectively non-relations are correctly identified by the measurements, except for test case 74, which we will outline below.

Surprises. *Moving Monsters.* According to the expert, a group of tests (72, 73, 74) all move monsters, and should lead to similar rankings. Surprisingly, one test (74) performs differently from the rest, and also differs from the conceptual mapping. After investigation, it became apparent that this test is not as focused as expected and desired by the expert. The test even concludes with a method which is never followed by an assertion statement. This investigation revealed a clear "test smell" and pointed the expert to a place in need of a refactoring.

Sneak paths. A surprising connection is provided for the "monster to player" test (*ETE11*), which is connected to "wonSneakPaths" (35). This relates to unit tests aimed at testing for *sneak paths* in state machines, following Binder's test approach for state machines [1]. A sneak path is an illegal transition, and the

JPacman sneak path test cases verify that illegal ⟨state, event⟩ pairs do not lead to a state change. To do so, the test case brings the application in the desired state (e.g., *Playing*, or *Died*), and then attempts to trigger a state change.

The process of bringing the application in a given state, however, may bear similarity with other test cases. For example in unit test 35, the player first wins. Then multiple steps, such as the player getting caught by a monster or the player running into a monster, are triggered which should not change the state from "won" to "lost" anymore. As this triggers the player to die or being killed, this sneak path test case shows up as being related not only to end-to-end tests triggering winning situations. A better separation of set-up logic (bringing the application in the right state) and executing the method-under-test would help reveal more focused associations.

Deviations. *Moving beyond border. ETE02* is the only test which does not map to a top match within the first 5 results. The first top matches are found from rank 7 onwards. Reasons for this behavior are that *ETE02* is one of the smallest end-to-end tests involving a move, and that testing the behavior for "beyond border" covers branches that only lead to different data, not different calls made. All 5 high ranked results correctly involve doing a move. After investigation of the results, the expert reports that the unit test cases indicated as related in the conceptual mapping do a bit more than only a move (e.g. an undo operation), which is why our approach gives these unit tests a lower rank.

Move to Wall. ETE03 contains the only unrelated connection within the top 10 results: on rank 9 is the "possible move" test. On the other hand, counterpart test "impossible move" is a top match.

Disparate test sizes. The main deviations (tests marked as unrelated being ranked higher than tests marked as related) are due to extreme size differences in unit tests. The expert easily relates narrow focused tests, whereby the automatic approach, by design of the shared word count, gives preference to broader tests (which share more events). A prime example is the "wonSneakPaths" test, which is related to many end-to-end tests as it triggers a broad range of functionality. The more equal the amount of functionality tested by the unit test cases is, the better the results revealed by the automatic approach.

Additional Lessons Learned. *API violations.* The smoke test (*ETE14*) consists of a series of events applied to JPacman. As such, it is fairly heterogeneous, doing many different things. This makes it hard to come up with a reasonable positive mapping to unit tests. On the other hand, some unit test cases are not relevant to *any* end-to-end test, including the "smoke test". As an example, tests 57, 58 and 59 deal with using the API in a wrong way, which should generate an appropriate exception. Seeing that these test cases are not related to the smoke test gives confidence that such violations are not possible at a higher level.

Human vs. automated mapping. Fine-grained deviations between tests, like state and specific object instantiations, have been used by the expert to relate tests to each other. For example, for the expert the determining events for relating unit test cases involving moving to end-to-end tests have been the actual actors (objects). The automated approach is able to differentiate similar to an expert between objects. On the other hand, the importance of states for human mappings is not equally reflected by the automated approach as it assigns every event the same importance. Identifying and prioritizing states before the similarity calculation is performed could improve the approximation to the "human" conceptual mapping. As we will see in the second case study, if tests are small and focused, the impact of state changes reflects well in the similarity measurements.

5.3 RQ4: Performance Characteristics

Since JPacman is the larger case study, we will answer RQ4 here. The traces obtained for both case studies are relatively small: the smallest one is 1kb and comprises 2 trace events, the largest being 62Kb and 700 trace events (after applying trace reduction). Similarity calculations within this dimension are computed within 10 seconds for the whole test suite. Even the results for the smoke test of JPacman, comprising approximately 60,000 trace events (4Mb) before reduction, are almost instantly ready after applying trace reduction techniques.

6 Case Study II: Auction Sniper

The second case study revolves around a system developed in strict test-driven development (TDD) manner called Auction Sniper. Its test suite also makes heavy use of mocking techniques in order to isolate unit tests. In contrast to the first case study, where we compare the test relations with a conceptual mapping of an expert, in this case study we investigate the usefulness of the technique to help an outsider understand test relations (RQ2). In addition, we investigate how our technique can cope with the influence of mocking techniques (RQ3).

Auction Sniper is an application which allows to automatically bid in auctions. Auction Sniper watches different auctions and increases the bid in case a higher bid of another bidder arrived until the auction closes or a certain limit has been reached. This system is used as an example in the book "Growing Object-Oriented Software, Guided by Tests" by Freeman et al. [6] to describe TDD techniques. The software and the related tests are explained in detail in this book and are publicly available[9]. The system comprises approximately 2,000 lines of code. The test suite has 1,240 lines of code, which constitute 37 unit tests, 6 end-to-end tests and 2 integration tests.

6.1 Obtaining an Initial Understanding

We analyzed the book and derived an initial understanding of the relations between end-to-end tests and unit tests. The authors always start with an end-to-end test, which kick-starts each development circle for a new functionality, whereby the authors explain each end-to-end test "should have just enough new requirements to force a manageable increase in functionality" [6]. Then, the

[9] `https://github.com/sf105/goos-code`

scaffold implementation of the new functionality follows. Prior to implementation of detailed functionality, the authors develop and explain the necessary unit tests.

Based on this iterative development, we map each unit test case developed within the cycle of an end-to-end (ETE) test as related to this ETE test. We refine this first mapping by identifying the differences of the ETE test cases based on their code. We mapped some unit tests not covered in the book based on their name. In the following we summarize the functionality of the six ETE tests. All unit test case names are given in Table 4 which can be helpful during comprehension of the presented results.

The end-to-end tests. ETE tests 01 to 06 are actually enhancements of each other, involving a lot of common functionality. The main steps are: 1. An auction sells an item, 2. An auction sniper joins this auction. 3. Then, the auction closes, 4. Finally, the auction sniper shows the outcome of an auction.

In addition, test cases 02 to 05 place actual bids. Only test case 06 deviates from the rest, as it does not close the auction and sends an invalid message. Another main difference between the test cases is the state in which the sniper is when the auction closes. In *ETE01* the sniper is in the state “joining” when the auction closes, which results in a lost auction. In *ETE02* the sniper makes a bid, but loses in the “bidding” state. In *ETE03* the sniper makes a higher bid, and wins in the “winning” state. *ETE04* simply tests that a sniper can bid on two items. The functionality of *ETE03* and *ETE04* is so similar that we will treat them as one test subsequently. In *ETE05* the sniper is already in “losing” state before the auction closes, because of a stop price. *ETE06* tests failure reporting. The test sends an invalid message to the auction and causes the application to throw and handle an error, but leaves the auction unclosed.

6.2 RQ2: Suitability of Measurements for Understanding Test Relations

After measuring the similarity of the tests, we investigate each unit test via code inspection, and assess the ranking and the mapping, which results in the final conceptual mapping illustrated in Table 4. Based on this detailed investigation we finally assess the rankings of the similarity measurements. Below we outline correct identifications, surprises and deviations of the measurements with our initial understanding by sketching groups of unit tests. We will see that the automatic mapping reflects the final mapping derived after in-depth investigation very accurately, and is thus useful for supporting an outsider in understanding the test suite and its relations. The rankings and assessments for the best 5 results are illustrated in Table 2. For test case *ETE06* we present the top 10 results to illustrate the effect of the relevancy support (see Table 3). A ranking is indicated as *optimal* only in case it is highly related and ranked within the top first results. Otherwise, results highly related, or results related are indicated as okay (i.e., *ok*) in case they are within the first 5 results. On the other hand, in case of a related result, which is not highly related, but is ranked before the highly related ones, it is marked as not okay (i.e., *nok*).

Listing 1.2. Test case: *isWonWhenAuctionClosesWhileWinning*

```
@Test public void
 isWonWhenAuctionClosesWhileWinning() {
  assertEquals(SniperState.LOST, SniperState.JOINING.whenAuctionClosed());
  assertEquals(SniperState.LOST, SniperState.BIDDING.whenAuctionClosed());
  assertEquals(SniperState.WON, SniperState.WINNING.whenAuctionClosed()); }
```

Correct Identifications. *Report Close of Auction.* Unit test cases 02, 08, and 09 revolve around reporting the closing of an auction, and are thus indeed related to all ETE tests except to *ETE06*. Nevertheless, each of them provokes a different state of the sniper when the auction closed event takes place. Therefore, the mapping should be the strongest for *ETE01* with test 02, *ETE02* with test 08, *ETE3/04* with test 11, and *ETE05* with test 09. The measurements for these relations accurately reflect those subtle differences.

Notifies Bid Details. Tests 33 and 34 are related to all of the ETE tests, except for *ETE01*, which does not make a bid. As *ETE02* exclusively focuses on bidding, the relation is correctly identified as the strongest for this test. For other tests they appear on ranks 6 and 7.

Mapping per Focus. Test case 03 which only bids correctly achieves the highest rank for test *ETE02*. Test case 10, related to winning an auction, maps to *ETE03/04*. Tests 05, 06 and 09, which address losing before the auction is closed are also correctly identified as highest-ranking results for *ETE05*. Test cases 35-37, and 12-15 are testing the reporting of a failing auction. They are correctly ranked as highly related to *ETE06*. *ETE06* is a good example to demonstrate the impact of the *relevancy support based on occurrence*, described in Section 3.1. Test cases 33 and 34 share more steps with *ETE06* than for example test cases 35 and 37. Both achieve just a similarity ranking of 0.2. Nevertheless, tests 35 and 37 reflect much stronger the focus of *ETE06*. Because 35 and 37 are never indicated as related to any other ETE test, the *relevancy support* pushes them to the top results. The new ranking of, for example test 35, is calculated as its similarity divided by the number of times it has been ranked as a result among all tests (i.e., 0.2 divided by 1/6).

Surprises. *Winning and State Transitions.* A surprise was the ranking of test case 20 "isWonWhenAuctionClosesWhileWinning" within the results of *ETE01*, as the name suggests it is rather related to winning (i.e., *ETE03/04*). Inspecting the code, illustrated in Listing 1.2, reveals that the name is misleading as it tests different auction outcomes. Two times the auction is lost, contrary to the name, and it also triggers the rarely addressed state of *ETE01* (i.e., "joining" when the auction is closed). Test case 18 also triggers the transition between each stage and therefore should have a low relation to each of the test cases.

Not bidding, bidding and losing. Test cases 05 and 06, contrary to their name suggestions, do place bids and lose and are therefore also related to other test cases than *ETE06*. Actually only test case 32 does not make a bid, which is correctly mapped to *ETE01* and gets low ratings for the other tests. Since test case 06 also reaches the winning state before losing, the indicated relation to *ETE03/04* in understandable.

ETE 01				ETE 02				ETE 03 & 04				ETE 05			
test	*sim*	avg	*match*	test	*sim*	avg	*match*	test	*sim*	avg	*match*	test	*sim*	avg	*match*
02	1.20	1.73	optimal	03	0.55	1.95	optimal	11	0.64	3.15	optimal	05	0.55	1.95	optimal
09	0.67	2.50	ok	08	0.55	2.88	optimal	06	0.55	2.13	ok	06	0.55	2.13	optimal
08	0.67	2.88	ok	11	0.45	3.15	nok	08	0.45	2.88	ok	09	0.55	2.50	optimal
11	0.67	3.15	ok	33	0.44	1.85	ok	10	0.44	1.57	ok	08	0.45	2.88	ok
20	0.50	0.61	ok	34	0.44	1.85	ok	15	0.44	1.77	ok	11	0.45	3.15	ok

Table 2. Top 5 similarity rankings for ETE01 to ETE05

Defects and a Failing Auction. We expected test cases 21, 22 to be related to *ETE06*. But, tests 21 and 22 create a different failure as they put the system in a faulty state and assert a specific exception to be thrown. Such a behavior is not triggered in the end-to-end test, and consequently the non-appearance of those test cases for any ETE is correct.

Deviations. *Reporting winning.* Test case 11, which reports that an auction has been won after closing is ranked as the third result for *ETE02* even though this end-to-end test addresses losing. The common events, such as starting an auction, bidding for an item and closing an auction dominate the ranking.

Additional Lessons Learned. *Common functionality.* Some functionality is common to all tests. For example, tests of the class "*SniperTablesModelTest*" check the rendering of the user interface. Tests 01, 16, and 17 trigger common functionality such as adding a sniper and listeners. Such trace events are reduced and can yield to empty test cases. Traces reduced to empty traces are marked as common functionality in the ranking.

6.3 RQ3: Handling Mocking

The test suite of Auction Sniper makes heavy use of the mocking library JMock. Without explicitly addressing mocked types during the analysis test cases involving mocked classes are ranked very low or as unrelated even though they are highly related. For example, without the mockery aspect test case 35 is not linked to test *ETE06* as the runtime types differ. By addressing mockery classes as described in Section 2.2 we can correctly identify test relations.

7 Discussion

Lessons learned and limitations. *Separation of Set-up and Tear-down.* Consistent usage of set-up and tear-down methods improves the similarity results, as it helps in revealing the core functionality and focus of test cases. Test suites which a priori do not use set-up and tear-down methods to structure their test might yield less accurate results.

ETE 06							
test	*sim*	avg	match	test	sim_s	avg	match
12	0.50	1.59	optimal	13	1.20	0.20	optimal
15	0.50	1.77	optimal	35	1.20	0.20	optimal
14	0.40	1.22	optimal	36	1.20	0.20	optimal
33	0.40	1.85	ok	37	0.60	0.10	optimal
34	0.40	1.85	ok	12	0.60	1.59	optimal
03	0.40	1.95	ok	15	0.60	1.77	optimal
05	0.40	1.95	ok	14	0.48	1.22	optimal
06	0.40	2.13	ok	33	0.48	1.85	ok
10	0.30	1.57	nok	34	0.48	1.85	ok
08	0.30	2.88	nok	03	0.48	1.95	ok

Table 3. Similarity rankings for ETE06 with and without support

Test Case Name	Test Case	Relation
sniperJoinsAuctionUntilAuctionCloses – ETE01		
notifiesAuctionClosedWhenCloseMessageReceived	≡ 32	highly related
reportsLostWhenAuctionClosesImmediately	≡ 02	highly related
isWonWhenAuctionClosesWhileWinning	≡ 20	related
reportAuctionClosesX	≡ 08, 09, 11	related
sniperMakesAHigherBidButLoses – ETE02		
reportsLostIfAuctionClosesWhenBidding	≡ 08	highly related
bidsHigherAndReportsBiddingWhenNewPriceArrives	≡ 03	highly related
doesNotBidAndReportsLosingIfSubsequentPriceIsAboveStopPrice	≡ 05	related
doesNotBidAndReportsLosingIfPriceAfterWinningIsAboveStopPrice	≡ 06	related
reportAuctionClosesX	≈ 09, 11	related
sniperWinsAnAuctionByBiddingHigher – ETE03 and sniperBidsForMultipleItems – ETE04		
reportsWonIfAuctionClosesWhenWinning	≡ 11	highly related
reportsIsWinningWhenCurrentPriceComesFromSniper	≡ 10	highly related
doesNotBidAndReportsLosingIfPriceAfterWinningIsAboveStopPrice	≡ 06	related
reportAuctionClosesX	≡ 08, 09	related
sniperLosesAnAuctionWhenThePriceIsTooHigh – ETE05		
reportsLostIfAuctionClosesWhenLosing	≡ 09	highly related
doesNotBidAndReportsLosingIfSubsequentPriceIsAboveStopPrice	≡ 05	highly related
doesNotBidAndReportsLosingIfPriceAfterWinningIsAboveStopPrice	≡ 06	highly related
doesNotBidAndReportsLosingIfFirstPriceIsAboveStopPrice	≡ 04	highly related
continuesToBeLosingOnceStopPriceHasBeenReached	≡ 07	highly related
(reportAuctionClosesX)	≡ 08, 11	related
sniperReportsInvalidAuctionMessageAndStopsRespondingToEvents – ETE06		
notifiesAuctionFailedWhenBadMessageReceived	≡ 35	highly related
notifiesAuctionFailedWhenEventTypeMissing	≡ 36	highly related
writesMessageTranslationFailureToLog	≡ 37	highly related
reportsFailedIfAuctionFailsWhenBidding	≡ 12	highly related
reportsFailedIfAuctionFailsImmediately	≡ 13	highly related
reportsFailedIfAuctionFailsWhenLosing	≡ 14	highly related
reportsFailedIfAuctionFailsWhenWinning	≡ 15	highly related
ETE 01 – 06		
transitionsBetweenStates	≡ 18	related
ETE 02 – 06		
bidsHigherAndReportsBiddingWhenNewPriceArrives	≡ 03	related
ETE 02 - 05		
notifiesBidDetailsWhenCurrentPriceMessageReceivedFromOtherBidder	≡ 33	related
notifiesBidDetailsWhenCurrentPriceMessageReceivedFromSniper	≡ 34	related
Common functionality and UI		
UI related tests (e.g. test of class SniperTablesModelTest)	≡ 23 − 31	related
Listeners and common states	≡ 01, 16, 17	related
Functionality not addressed by any ETE		
defectIfAuctionClosesWhenWon	≡ 21	unrelated
defectIfAuctionClosesWhenLost	≡ 22	unrelated

Table 4. Final conceptual mapping of end-to-end tests to unit tests

Performance. The performance of the approach is an important criterion especially if the size and complexity of the system under study increases. During our two case studies, we experienced no performance issues with the systems under study. For larger systems further trace reduction techniques might become necessary [3]. On the other hand, performance depends more on the size of the traces (i.e., amount of functionality covered by a test), than on the number of tests. Test case size is independent of the complexity and size of the systems.

Future work. *Assertions.* At this stage, our technique does not address the meaning of assertions. As future work, we would like to investigate how the meaning of assertions can influence the ranking of a test case.

Test suite quality inspection. The discovered relations do not only help to see similarity of test cases, they also help to assess the quality of the test suite and discover areas for improvement, e.g., identifying unit test cases that do too much, or identifying behavior which is not addressed by any end-to-end test.

User study. We aim to further investigate the usefulness of our tool through a user study that allows actual developers and testers to work with it.

Threats to validity. Concerning *external validity*, our case studies address relatively small Java systems. Scalability to larger case studies is a key concern that we aim to address in our future work, making use of case studies from the Eclipse plug-in domain we used in earlier studies (Mylyn, EGit) [8].

With respect to *internal validity*, the main threat consists of the manually obtained conceptual mapping. Creating such a mapping is inherently subjective, as illustrated by the process we applied to the Auction Sniper case study.

In order to reduce *threats to reliability* and to improve repeatability, both our tool and the systems under study are available to other researchers.

8 Related Work

An initial catalogue of *test smells* negatively affecting understanding was presented by Van Deursen et al., together with a set of corresponding refactorings [5]. Later, a thorough treatment of the topic of refactoring test code was provided by Meszaros [10]. Van Rompaey et al. continued this line of work by studying automated analysis of these smells [13].

Tools for assisting in the understanding of test suites have been proposed by Cornelissen et al., who present a visualization of test suites as sequence diagrams [2]. Greiler et al. propose higher level visualizations, aimed at assisting developers in seeing plug-in interactions addressed by their test suites [8].

Galli et al. have developed a tool to order broken unit tests [7]. It is their aim to create a hierarchical relation between broken unit tests, so that the most specific unit test that fails can be inspected first. In essence, their technique allows to steer and optimize the debugging process.

Rothermel and Harrold discuss safe regression testing techniques in [11]; regression test selection techniques try to find those tests that are directly responsible for testing the changed parts of a program and subsequently only run these tests. Hurdugaci and Zaidman operationalize this in the IDE for unit tests [9].

Yoo et al. cluster test cases based on their similarity to support experts in test case prioritisation, which outperforms coverage-based prioritisation [15].

9 Conclusion

In this paper we showed how a combination of *dynamic analysis* and the shared word count metric can be used to establish relations between end-to-end and unit tests in order to assist developers in their (test suite) maintenance activities.

We evaluated our *test connection mining* techniques in two case studies, by elaborating the usefulness of the approach to improve understanding. We saw that after using the proposed trace reduction techniques our approach produces accurate test mappings, which can help to 1) identify relevant tests, 2) understand the functionality of a test by describing high-level test cases with related unit test cases and 3) reveal blank spots in the investigated unit test suites.

Contributions. The contributions of this paper are 1) tracing and trace reduction techniques tailored for handling test code, including test specific events such as set-up, tear-down and mocking 2) an assessment of the usefulness of the rankings based on two case studies, 3) the development of a *Test Similarity Correlator*, a framework for mining test connections.

References

1. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional (Oct 1999)
2. Cornelissen, B., van Deursen, A., Moonen, L., Zaidman, A.: Visualizing testsuites to aid in software understanding. In: Proc. of the European Conference on Software Maintenance and Reengineering (CSMR). pp. 213–222. IEEE CS (2007)
3. Cornelissen, B., Moonen, L., Zaidman, A.: An assessment methodology for trace reduction techniques. In: Proc. Int'l Conf. Software Maintenance (ICSM). pp. 107–116. IEEE CS (2008)
4. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R.: A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering 35(5), 684–702 (2009)
5. van Deursen, A., Moonen, L., van Den Bergh, A., Kok, G.: Refactoring test code. In: Extreme Programming Perspectives. pp. 141–152. Addison Wesley (2002)
6. Freeman, S., Pryce, N.: Growing Object-Oriented Software, Guided by Tests. Addison-Wesley Professional, 1st edn. (2009)
7. Galli, M., Lanza, M., Nierstrasz, O., Wuyts, R.: Ordering broken unit tests for focused debugging. In: Int'l Conf. Softw. Maintenance (ICSM). pp. 114–123. IEEE (2004)
8. Greiler, M., Groß, H.G., van Deursen, A.: Understanding plug-in test suites from an extensibility perspective. In: Proceedings Working Conference on Reverse Engineering (WCRE). pp. 67–76. IEEE CS (2010)
9. Hurdugaci, V., Zaidman, A.: Aiding developers to maintain developer tests. In: Conf. Softw. Maintenance and Reengineering (CSMR). pp. 11–20. IEEE CS (2012)
10. Meszaros, G.: xUnit Test Patterns: Refactoring Test Code. Addison-Wesley (2007)
11. Rothermel, G., Harrold, M.: Empirical studies of a safe regression test selection technique. IEEE Transactions on Software Engineering 24(6), 401–419 (1998)
12. Stephen, G.A.: String searching algorithms. World Scientific Publishing Co. (1994)
13. 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. IEEE Transactions on Software Engineering 33(12), 800–817 (2007)
14. Weiss, S., Indurkhya, N., Zhang, T., Damerau, F.: Text Mining: Predictive Methods for Analyzing Unstructured Information. SpringerVerlag (2004)
15. Yoo, S., Harman, M., Tonella, P., Susi, A.: Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: Proceedings of the eighteenth international symposium on Software testing and analysis. pp. 201–212. ISSTA '09, ACM, New York, NY, USA (2009)
16. Zaidman, A., Van Rompaey, B., van Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empir. Softw. Eng. 16(3), 325–364 (2011)

TUD-SERG-2012-008
ISSN 1872-5392