Acta Cybernetica **23** (2018) 903–919.

Analysis of Static and Dynamic Test-to-code Traceability Information[*]

Tamás Gergely[a], Gergő Balogh[ab], Ferenc Horváth[a], Béla Vancsics[a], Árpád Beszédes[ac], and Tibor Gyimóthy[ab]

Abstract

Unit test development has some widely accepted guidelines. Two of them concern the test and code relationship, namely isolation (unit tests should examine only a single unit) and separation (they should be placed next to this unit). These guidelines are not always kept by the developers. They can however be checked by investigating the relationship between tests and the source code, which is described by test-to-code traceability links. Still, these links perhaps cannot be inferred unambiguously from the test and production code.

We developed a method that is based on the computation of traceability links for different aspects and report Structural Unit Test Smells where the traceability links for the different aspects do not match. The two aspects are the static structure of the code that reflects the intentions of the developers and testers and the dynamic coverage which reveals the actual behavior of the code during test execution.

In this study, we investigated this method on real programs. We manually checked the reported Structural Unit Test Smells to find out whether they are real violations of the unit testing rules. Furthermore, the smells were analyzed to determine their root causes and possible ways of correction.

Keywords: test-to-code traceability, unit testing, code coverage, test smells, refactoring

[*]This study was supported by the project "Integrated program for training new generation of scientists in the fields of computer science", no EFOP-3.6.3-VEKOP-16-2017-0002. The project was supported by the European Union and co-funded by the European Social Fund.

[a]Department of Software Engineering, University of Szeged, Hungary, E-mail: `{gertom,geryxyz,hferenc,vancsics,beszedes,gyimothy}@inf.u-szeged.hu`

[b]MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, Hungary

[c]This author was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

DOI: 10.14232/actacyb.23.3.2018.11

1 Introduction

Unit testing is an important element of software quality assurance, and it plays an important role in software maintenance and evolution. For example, during continuous integration, unit tests are constantly re-executed and further evolved (by developers) in parallel with the system under test [13]. This is why the quality of unit tests (including maintability) is important for software quality.

There are several guidelines, design patterns, and frameworks that help the developers to write good unit test cases [17]. Among these guidelines, there are two that deal with the structural consistency between test and production code [17]. The first one is *isolation*, which means that unit tests should exercise only the unit they were designed for, while the second one is *separation*, meaning that the tests should be placed in the same logical or structural group (like packages or namespaces) as the units they are testing. These guidelines, if kept, assist both the traceability between the test and production code, and maintainability. However, some practical aspects may prevent unit test designers and developers from creating tests that completely conform to these definitions (*e.g.* calls to utility functions or general parts of the system [5, 21]). Also, refactorings and code reorganizations might detrimentally affect the fulfillment of the isolation and separation guidelines for test and production code. Places in the code where these rules are not kept can be treated as test smells ([28, 3]): they are not bugs nor do they harm maintainability by definition, but such locations should be investigated anyway.

Relations between test and production code elements may be treated as traceability links, and several approaches have been proposed for their recovery (*e.g.* [23, 22, 16, 19, 9, 7]). However, as different approaches use different information to recover traceability, these might produce different results [23].

In a previous study, we proposed a method for investigating unit test and code relationship [2]. Here, we use this method to identify so-called *Structural Unit Test Smells*, a concept which we introduce to describe structural issues in the tests with respect to the system as a whole, and not just issues in isolated pieces of code. The method uses the idea that test-to-code traceability recovered from different sources captures different type of relations. Namely, we compare traceability links recovered from static sources that reflect the intention of the test designers and from dynamic sources that reflect the actual behavior of the code during test case execution. Differences in the two types of recovered traceability might suggest test and code elements that violate isolation and separation guidelines.

In the first phase of our approach we compute the traceability links based on two fundamentally different but very basic aspects, these being (1) the static relationships of the tests and the tested code in the physical code structure, and (2) the dynamic behavior of the tests based on code coverage. In particular, we compute *clusterings* of tests and code for both static and dynamic relationships, which represent coherent sets of tests and tested code. These clusters represent sets whose elements are mutually traceable to each other, and may be beneficial over individual traceability between units and tests, which is often hard to express precisely. To compute the static clusters we use the packaging structure of the code, while for the dynamic clustering we employ community detection [6] on the method level code coverage information.

In the next phase, these two kinds of clusterings are compared with each other. We do this using a *Cluster Similarity Graph* (CSG) that represents the computed clusters as graph nodes and connects the static and dynamic clusters that have common elements. If both approaches produce the same clusters, then they are said to agree, connecting only pairs of (one static and one dynamic) nodes in the CSG, in which case we conclude that the (test and code) elements contained in the clusters conform to the given unit testing guidelines. However, in many cases there will be discrepancies in the results obtained represented as several interconnected nodes in the CSG, which we report as Structural Unit Test Smells. There may be various reasons for these SUTSs, but they are usually some combination that violates the isolation and/or separation principles mentioned above.

To assess the practical usability of the method, in this study, we applied it on non-trivial open source Java systems and their JUnit test suites. We manually investigated the reported Structural Unit Test Smells by recovering and analysing their context and finding the root cause of the detected discrepancy between the static and dynamic traceability. We also made decisions on each SUTS as to whether it is a 'false positive' (*i.e.* the test and code conforms to unit testing rules) or whether it points to test and production code that should be reorganized in some way.

The rest of the paper is organized as follows. In the next section we provide an overview of some background information and related work, then in Section 3 we describe our traceability recovery method, with the analysis of the detected discrepancies in Section 4. In Section 5 we discuss threats to validity of the study. Lastly, in Section 6 we draw some conclusions and make some suggestions for future work.

2 Background and Related Work

There are different levels of testing, one of which (the lowest level) is called unit (or component) testing. Unit tests are closely related to the source code and they seek to test separate code fragments. This kind of test helps one to find implementation errors early in the coding phase, and helps to reduce the overall cost of the quality assurance activity.

Several guidelines exist that provide hints on how to write good unit tests (*e.g.* [17, 27, 20]), but there are two basic principles that are mentioned by most of them. The first is that unit tests should be isolated (*i.e.* test only the elements of the target component) and separated (*i.e.* physically or logically grouped, aligned with the unit being tested). In practice, this means that unit tests should not (even indirectly) execute production code outside the tested unit, and they should follow a clear naming and packaging convention, which reflects both the purpose of the test and structure of the given system. Several studies have examined various characteristics of the source code with which the above mentioned two aspects can be measured and can be verified to some extent (see, for example [23]).

These two properties are necessary for the approach described in this paper. Namely, if both are strictly followed, the two automatic traceability analysis algorithms we used (one package-based and the other coverage-based), will produce the same results. However, this is not the case for realistic systems, so our approach relies on analyzing the differences between the two sets in order to infer things about the final traceability links.

Several methods have been proposed to recover traceability links between software artifacts of different types, including requirements, design documentation, code, test artifacts, and so on [24, 10]. The approaches include static and dynamic code analysis, heuristic methods, information retrieval, machine-learning, data-mining based methods.

In this study, we are concerned with a specific type of traceability, namely *test-to-code* links. The purpose of recovering such links is to assign test cases to code elements of the system under test based on the relationship that shows which code parts are tested by which tests. This information may be vital in different activities including development, testing and maintenance, as mentioned earlier.

We shall concentrate on unit tests, in which case the traceability information is mostly encoded in the source code implementing the production system and the test cases, and usually no external documentation is available for this purpose. Traceability recovery for unit test may seem straightforward at first sight, given that the basic purpose of a unit test is to test a single unit of code [4, 11]. However, in reality it is not so [16, 19].

Several studies have been conducted on this subject, which examined the problem of traceability and made suggestions about it [9, 7, 23, 22]. Most of these related studies emphasize that reliable test-to-code traceability links are difficult to obtain from a single source of information, and a combination of (or semi-automatic) methods are required. Here, we will utilize this finding to determine the test smells

Our study mainly focuses on test (and code) smell identification. The discrepancies found in the two automatic traceability analysis results can be viewed as some sort of smell, and this suggests potential problems in the structural organization of the tests and code. Code smells (first introduced by Fowler [15]) are an established concept for classifying shortcomings in the software. Similar concepts for checking software tests and test code for quality issues have also been applied. For tests that are implemented as executable code, Van Deursen *et al.* introduced the concept of *test smells*, which suggest poorly designed test code [12], and listed 11 test code smells with recommended refactorings. We can relate our study best to their concept of *Indirect Testing Smell.* Meszaros expanded the scope of the concept by describing test smells that act at a behavior or a project level, next to code-level smells [20]. Results that came after this study use these ideas in practice. For example, Breugelmans and Van Rompaey [8] present TestQ, which allows developers to visually explore test suites and quantify test smelliness. They visualized the relationship between test code and production code, and with it, engineers were able to better understand the structure and quality of the test suite of large systems [27].

Our study significantly differs from these approaches as we are not concerned with code-oriented issues in the tests, but with their dynamic behavior and relationship to their physical location in the system as a whole. We may identify *Structural Unit Test Smells* from an analysis of the discrepancies found in the automatic traceability analyses.

3 Method

We define *Structural Unit Test Smell* (SUTS) as those suspicious parts of either the test or production code which seem to violate best practices used during unit test creation, execution and maintenance. In this respect, they are essentially inconsistencies in the physical organization and the logical behavior of unit test code and the tested code.

3.1 Overview

Figure 1 provides an overview of the above process, which has several sequential phases. First, the physical organization of the *production* and *test code* into Java packages is inferred, and the required *test coverage* data is produced by executing the tests. In our setting, code coverage refers to the individual recording of all methods executed by each test case. Physical code structure and coverage will be used in the next phase as inputs to create two *clusterings* over the tests and code elements.

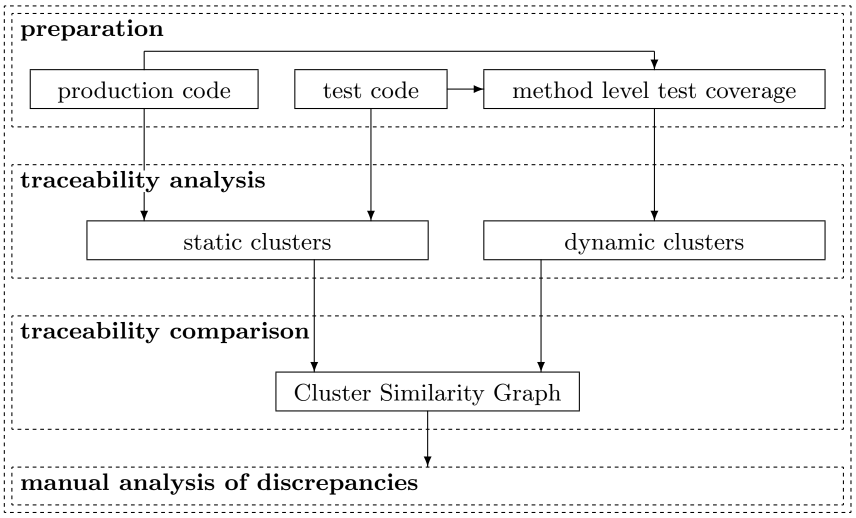


Figure 1: Overview of the method

These will represent the two types of relationships between the test and code, these being the two sets of automatically produced traceability links from two viewpoints, namely static and dynamic. Both clusterings produce sets of clusters that are made up of a combination of tests (unit test cases) and code elements (units under test). In our case, a unit test case is a Java test method (*e.g.* using the `@Test` annotation in the JUnit framework [26]), while a unit under test is a Java production code method.

In our approach, the elements of a cluster are mutually traceable to each other, and no individual traceability is considered between individual test methods and production methods. The advantage of this is that in many cases it is impossible to uniquely assign a test case to a unit; instead *groups* of test cases and units may represent a cohesive functional unit [18]. Also, minor inconsistencies, such as helper methods that are not directly tested, are concealed in this procedure. Details about the clustering based traceability algorithms are provided in the next section.

The automatically produced traceability links of the two analyses will be compared using a helper structure called the *Cluster Similarity Graph* (CSG) [2]. This is a directed bipartite graph whose nodes (disjointly) represent the clusters of the two clusterings. Each edge of the graph connects two nodes representing one static and one dynamic cluster, and weights on them denote the level of similarity between the two corresponding cluster nodes (based on the elements contained in the two corresponding clusters). Weights can be calculated using a pairwise similarity measure. In particular, we can use the *Inclusion measure.* Let K_1 and K_2 be two clusters of different types (one static and one dynamic). The Inclusion measure $I(K_1, K_2)$ expresses to what degree the elements of K_1 are included in K_2. A value of 0 means no inclusion (fully disjoint clusters), while a value of 1 means that K_1 is a subset of K_2. Edges with a 0 inclusion value are omitted from the CSG.

Figure 2 shows an example CSG taken from one of our subject systems, `oryx`. The static clusters are represented as purple rectangles, while the dynamic clusters are represented as green boxes. The edge weights are not shown in this example. Both types of clusters contain test and code elements. Edges in the figure mean that the two connected clusters have some common items (test or production methods). For example, the elements of dynamic cluster 9 are completely contained in the static cluster `com/cloudera/oryx/common` (as dynamic cluster 9 has no common elements with other clusters), and dynamic cluster 11 shares its elements with two static ones. Note that in the example the numbers of the dynamic clusters have no special significance, while the static clusters are named after the package that contains their elements.

After we had created the CSG, we looked at it closely. It is obvious that in the ideal case static and dynamic clusterings produce the same clusters, hence the CSG contains only connected pairs of nodes. However, in practice the CSGs are not like this, and each pattern in the graph that consists of more than two connected nodes can be treated as a Structural Unit Test Smell. Therefore, we manually examined the different patterns in the CSG and tried to discover what properties of the code and tests caused them. The results for this are presented in Section 4.

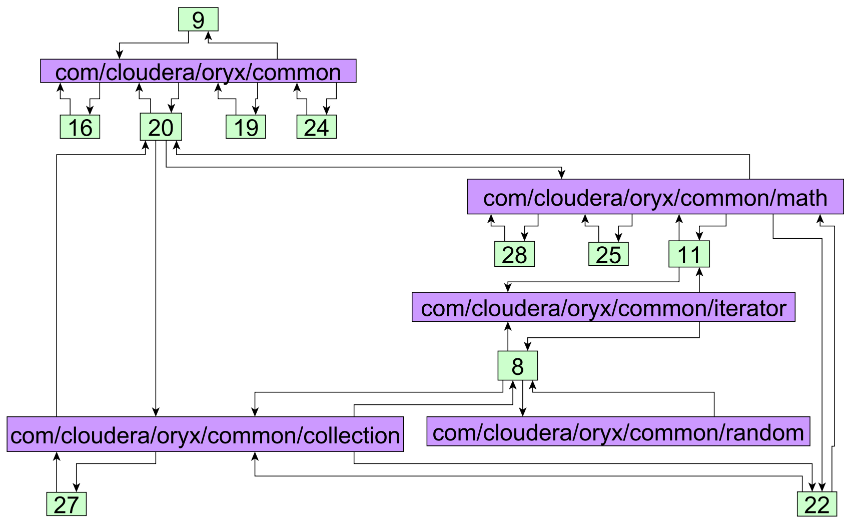

9
com/cloudera/oryx/common
16
20
19
24
com/cloudera/oryx/common/math
28
25
11
com/cloudera/oryx/common/iterator
8
com/cloudera/oryx/common/collection
com/cloudera/oryx/common/random
27
22

Figure 2: A part of the CSG of the `oryx` program

3.2 Clustering based traceability analysis

Our approach for unit test traceability recovery includes a step in which traceability links are identified automatically by analyzing the test and production code from two perspectives (static and dynamic). In both cases, clusters of code and tests are produced which jointly constitute a set of mutually traceable elements. Here, we deal with Java systems and rely on unit tests implemented in the JUnit test automation framework. In this context, elementary features are usually implemented in the production code as methods of classes, while the unit test cases are embodied as test methods. A system is then composed of methods grouped into classes and classes into packages. All of our algorithms have the method-level granularity, *i.e.* clusters are composed of production and test methods. Here, we do not explicitly take into account class information during the clusterings.

3.2.1 Static clustering

Through static clustering, our intention is to detect groups of tests and code that are connected together by the intention of the developer or tester. The placement of the unit tests and code elements within the package hierarchy of the system is a natural classification according to their intended role. When tests are placed within the package the tested code is located in, it helps other developers and testers to understand the connection between tests and their subjects. Hence, it is essential that the physical organization of the code and tests be reliable and reflect the developer's intentions.

Our package-based clustering simply means that we assign the fully qualified package name of the method to each production and test method, and treat methods (of both types) belonging to the same package as members of the same cluster. Class information and higher level package hierarchy are not directly taken into account. For example, package `com.cloudera.oryx.common` and its subpackages `com.cloudera.oryx.common.math` and `com.cloudera.oryx.common.random` are treated as unique clusters containing all the methods of all the classes directly contained within them. Furthermore, we do not directly consider the physical directory and file structure of the source code elements (although in Java, these usually tell us something about the package structure).

3.2.2 Dynamic clustering

In order to determine the clusters of tests and code based on the actual dynamic behavior of the test suite, we apply *community detection* [6, 14] on the code coverage relations.

Code coverage in this case means that, for each test case, we record what methods were invoked during the test case execution. This forms a binary matrix called a *coverage matrix*, with test cases assigned to its rows and methods assigned to the columns. A value of 1 in a matrix cell indicates that the particular method is invoked at least once during the execution of the corresponding test case (regardless of the actual statements and paths taken within the method body), and 0 indicates that it has not been covered by the test case.

Community detection algorithms were originally defined on (possibly directed and weighted) graphs. Thus, in order to use the selected algorithm, we construct a graph (referred to as the *coverage graph* in the following) from the coverage matrix. The nodes are the methods and tests of the system being analyzed, and there is an edge between a method and a test node if and only if the corresponding cell in the coverage matrix is 1. There is no edge between any two nodes of the same type.

The actual algorithm we used for community detection is the Louvain Modularity method [6]. It is a greedy optimization method based on the modularity metric, which penaltizes edges between clusters and rewards edges inside clusters. The algorithm works iteratively, and each pass is composed of two phases. In the first phase it starts processing single-node clusters and continues to unify clusters until no more unification leads to an increase of modularity. In the second phase, a new graph is created by assigning a single node to each clusters of the previous graph. New edges are also computed and weighted based on the edges between the cluster elements in the previous graph. The algorithm iterates these two steps until it reaches a graph in which no nodes can be unified in terms of modularity.

4 Analysis of traceability discrepancies

We manually analyzed all discrepancy instances found in the results produced by the static and dynamic traceability detection approaches. In this procedure, we considered the CSGs, the associated edge weights, and examined the corresponding parts of the production and test code. For the first step, each subject system was assigned to one of the authors of the paper for initial comprehension and analysis of the resulting patterns. The analysis required an understanding of the code structure and to some extent the intended goal of the test cases. API documentation, feature lists, and other public information were also examined during this phase. Next, the researchers made suggestions on the possible recovered traceability links and eventual code refactorings. Then, all the participants were involved in a discussion about the final decisions. The edge weights in the CSGs obtained during the analysis helped us to assess the importance of a specific cluster. For example, small inclusions were often ignored because these were in many cases due to some kind of outlier relationships that did not affect the overall structure of the clusters.

The results of the analysis were possible explanations for the reported SUTS with concrete suggestions, as well as the corresponding general guidelines for possible refactorings.

4.1 Subject programs

Our subject systems (see Table 1) were medium-to-large-size open-source Java programs, with unit tests implemented using the JUnit test automation framework. We chose these systems because they had a reasonable number of test cases compared to the system size.

Table 1: Subject programs

Program	Version	LOC	Methods	Tests
checkstyle	*6.11.1*	114K	2 655	1 487
netty	*4.0.29*	140K	8 230	3 982
orientdb	*2.0.10*	229K	13 118	925
oryx	*1.1.0*	31K	1 562	208

We modified the build processes of the systems to produce method level coverage information using the Clover coverage measurement tool [1]. For storing and manipulating the data, we used the SoDA framework [25], *e.g.* to process the coverage matrix. Then, we implemented a set of Python scripts to perform clusterings, including a native implementation of the community detection algorithm.

4.2 Identified *Structural Unit Test Smells*

Now, we present 8 SUTSs that we found and analyzed manually. These were the simplest smell patterns in the CSGs, where we suspected the nature of the smell and could give clear refactoring options (even if we sometimes provided more possible, contradictory options to a single smell instance). We encountered more complex patterns during our experiment (containing tens of clusters and hundreds of traceability links), but a deep analysis of these lay outside the scope of this present study.

4.2.1 com/puppycrawl/tools/checkstyle/doclets

This package belongs to our subject `checkstyle` and it is composed of the class `TokenTypesDoclet` with all 5 of its methods and of the test class `TokenTypesDocletTest` with its 6 test cases. The package is used to create a configuration/property file with short descriptions of `TokenTypes` constants. There are 4 dynamic clusters connected to this static cluster. They are one for option validation (1 method, 1 test case), a cluster for file name handling and file creation (2 methods, 3 test cases), one for counting options (1 method, 1 test case) and one for static initialization (1 method, 1 test case).

Possible explanation and refactoring: The dynamic clusters describe the subfunctionalities correctly. This SUTS is the result of the clustering and granularity we are working with, where our units are at the package level, while we work at the method level and this enables our method to identify smaller units. Namely, the sub-features are tested separately, but the implemented (and tested) classes are not separated into different packages, which is a quite reasonable decision in this case. Hence, we treat this SUTS as a false positive, and suggest that no refactoring should be performed.

4.2.2 io/netty/handler/codec/haproxy

`HAProxy` is a submodule of `netty` which is responsible for handling load balancing-related protocols. It has 7 classes, 42 methods and 30 test cases arranged in a single test class of over 1000 lines (`HAProxyMessageDecoderTest`). There are two dynamic clusters connected to it, these being one for messages and protocols (39 methods, 29 test cases) and one with two helper classes and their test case (3 methods, 1 test case).

Possible explanation and refactoring: A straightforward solution would be to split the package according to the dynamic behavior. However, we think that one of the resulting subpackages would be too small as a single package. Instead, the package should be divided into packages of messages, protocols and others. It would require other refactorings as well (*e.g.* involve splitting of the big test class) to produce a structure that conforms with the unit testing guidelines. However, this refactoring cannot be directly derived from the identified clusters alone. It requires a deeper analysis and knowledge of the code.

4.2.3 com/cloudera/oryx/als/common

This is the core package of the `als-common` submodule of subject `oryx`. It contains comparators, custom exception classes and small utility classes, and consists of 9 classes, one interface with altogether 18 methods and 4 test classes with 17 test cases. Four dynamic clusters are connected, these being a string-to-long map utility (1 method, 6 test cases), the `DataUtils` class and related test cases (1 method, 2 test cases), another string-to-long map utility (5 methods, 2 test cases) and a set of comparators and utility methods (11 methods, 7 test cases).

Possible explanation and refactoring: The dynamic clusters capture the sub-functionalities correctly, and there is room for refactoring. That is, exceptions, comparators, and utilities could be separated into three different packages, one for each. From the identified dynamic clusters, we seem to have a good basis for reorganizing the code, but additional decisions and corrections are needed. This Structural Unit Test Smell turned out to be true positive.

4.2.4 com/cloudera/oryx/common/io

The next *Structural Unit Test Smell* in `oryx` is the `io` package with `DelimitedDataUtils`, which is adapted from `SuperCSV` as a fast/lightweight alternative to its full API and `IOUtils`, a collection of simple utility methods related to I/O operations. It is composed of these two classes with 10 methods and the related 11 test cases. The four connected dynamic cluster are: `DelimitedDataUtils.encode` (3 methods, 5 test cases), `IOUtils.delete` (3 methods, 1 test case), `DelimitedDataUtils.decode` (2 methods, 4 test cases) and `IOUtils.copy` (2 methods, 1 test case).

Possible explanation and refactoring: This Structural Unit Test Smell is the result of how we define the unit in the static case. The dynamic clusters describe the sub-functionalities correctly at the method level, but the static cluster is too small to suggest some reorganization of the tests and the code. Actually, it turns out to be a false positive SUTS.

4.2.5 com/cloudera/oryx/common/stats

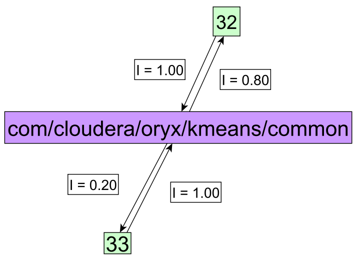
This static package is a common submodule of `oryx` which provides different statistics, and it has 5 classes with 24 methods, and 4 test classes with 13 test cases altogether. This pattern has 4 dynamic clusters: a weighted mean implementation for floating-point weights (6 methods, 5 test cases), a similar module with integer weights (7 methods, 6 test cases), a class encapsulating a set of statistics like mean, min and max (4 methods, 1 test case) and a bean class encapsulating some characteristics of the JVM runtime environment (7 methods, 1 test case).

Possible explanation and refactoring: The dynamic clustering split the static package into smaller units. Following the dynamic clusters would result in four units, but here we suggest that just the bean class `JVMEnvironment` should be separated, and the others have a similar functionality. Although the SUTS is valid, the clusters can only be partially used to determine the refactoring and further knowledge is required to do it correctly.

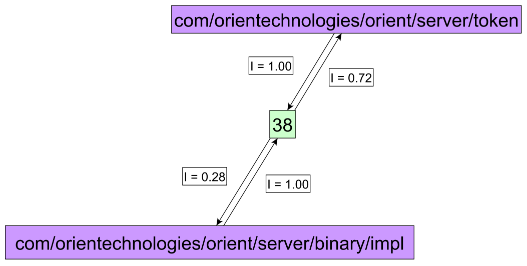
4.2.6 com/cloudera/oryx/kmeans/common

This package is the core of the `kmeans-common` submodule of `oryx` which is responsible for providing the k-means algorithm and several evaluation strategies. It includes 12 classes and 4 interfaces with 51 methods altogether, and 5 test classes about 60-70 lines long, providing 20 test cases. This case includes two coverage-based clusters (see Figure 3(a)), one for the evaluation strategies, weights, cluster centers, validity and statistics (40 methods, 17 test cases) and one that includes other evaluation strategies (11 methods, 3 test cases).

Possible explanation and refactoring: This package could be safely splits into two packages according to the dynamic clusters, one being responsible for the *statistical validation* and the other being responsible for the *strategies.* In Figure 3(a), cluster 32 should correspond to the *validation* and 33 should correspond to the *strategies.* This is a clear example of the situation where the Structural Unit Test Smell is not only valid, but the clusters involved in it also clearly show how the refactoring should be carried out.



(a)


com/orientechnologies/orient/server/token
I = 1.00
I = 0.72
38
I = 0.28
I = 1.00
com/orientechnologies/orient/server/binary/impl

(b)

Figure 3: Examples of a clustering comparison

4.2.7 com/cloudera/oryx/kmeans/computation/covariance

Another SUTS in `oryx` is the package responsible for handling covariance computations in k-means. It includes two classes called `CoMoment` and `Index` with 9 methods altogether, and the corresponding test classes with 3 test cases in total. There are 2 dynamic clusters involved, one for the first class (6 methods, 2 test cases) and one for the other class (3 methods, 1 test case).

Possible explanation and refactoring: In this SUTS, the static cluster was divided into two dynamic clusters. These two clusters are too small to be reasonably separated into distinct packages. Hence, the SUTS is a false positive. However, we note that in our investigation we found that this functionality might not be tested properly, and it might have some indirect effect associated with the smell that was reported.

4.2.8 An example of dynamic cluster

This SUTS belongs to the subject program called `orientdb`. The corresponding part of the CSG can be seen in Figure 3(b). This pattern consists of a coverage-based cluster (with serial number 38) and two related static clusters. By investigating the content of the dynamic cluster, we found that it is mainly responsible for token handling/serialization in the OrientDB Server. It includes all 6 classes (with 61 methods) from the `server/token` package and the `OBinaryToken` class (with 27 methods) from package `server/binary/impl`. In addition, it contains 2 test classes with 9 test cases. The former package is responsible for handling and serialization of Web authentication tokens, while the latter is a bean-like class which stores information about the user, database, protocol, driver and server. This class has no direct tests, but it is covered by those test cases which examine the token serializer and token handler classes.

Possible explanation and refactoring: Both static or dynamic clustering results could potentially be considered for refactoring, depending on what unit test writing principles the project follows. One solution might be to use mocking to eliminate the dynamic relation between the elements of the two packages. But merging the packages (*i.e.* moving `OBinaryToken` to the `token` package) to provide a single unit is also a reasonable option in this case. Thus, the reported Structural Unit Test Smell is valid, and suggests elements that should be refactored. Furthermore, the refactoring possibilities can be directly obtained from an analysis of clusters, although choosing one requires additional information.

5 Threats to validity

This study contains some threats to validity. First, we selected the subjects after assuming that the integrated tests using the JUnit framework are indeed unit tests, and not other kinds of automated tests. However, during manual investigation some tests turned out to be higher level tests, and in these cases the traceability links had a slightly different meaning from that for unit tests. Also, in practice the granularity and size of a unit might differ from what is expected (a Java method). Generally speaking, it is hard to ascertain automatically whether a test is intended or not intended to be a unit test, so we verified each identified patterns manually for these properties as well. However, in actual scenarios this information will probably be known beforehand.

Another aspect to consider about the manual analysis is that this study was performed by the authors of this paper, who are experienced researchers and programmers as well. However, none of them was a developer of the given systems, hence the decisions made about the Structural Unit Test Smells and refactorings would probably have been different if they had been made by a developer of the system.

6 Conclusions

In this study, we carried out an analysis of test-to-code traceability information. Unit test development has some widely accepted rules that support things like the maintenance of these tests suites. Some of them concern the structural attributes of these tests. These attributes can be described by traceability relations between the test and code. Previous studies demonstrated that fully automatic test-to-code traceability recovery is difficult, if not impossible in the general case [23, 16, 19]. There are several fundamental approaches proposed that have been proposed for this task, based on, among other things, static code analysis, call-graphs, dynamic dependency analysis, name analysis, change history and even questionnaire based approaches (see Section 2 above). However, there seems to be general agreement between researchers that no single method can provide accurate information about test and code relations.

Following this line of thinking, we developed a method that is able to detect Structural Unit Test Smells, *i.e.* locations in the code where unit test development rules are violated. In particular, we compute test-to-code traceability using two relatively straightforward automatic approaches, one based on the static physical code structure and the other on the dynamic behavior of test cases in terms of code coverage. Both can be viewed as objective descriptions of the relationship of the unit tests and code units, but from different perspectives; hence, each location where they disagree about traceability can be treated as a SUTS. Our approach is to use clustering and hence form mutually traceable groups of elements (instead of atomic traceability information), and this makes the method more robust because minor inconsistencies will probably not influence the overall results.

Here, we investigated the results of this method applied on four subject programs. Our goal was to manually check the reported Structural Unit Test Smells to see whether at least a part of these are real problems that needs to be examined. Experience indicates that most of the reported SUTSs point to parts of the test and code that could be reorganized to better follow unit test guidelines. However, in some situations it might not be worth modifying the tests and the code (*e.g.* for technical reasons). Overall, we found several typical reasons that could form the basis for future study and this might lead to an automatic classification of the Structural Unit Test Smells.

These findings have several implications. First, the method has a potential to find Structural Unit Test Smells, but the results will probably contain a large number of false positives. To filter out them, we need to carry out an investigation of the given situation. Fortunately, it seems that there are similar situations that can provide a basis for the automatic classification of the identified smells, and it may assist the developers in their refactoring activities. However, it is also clear from our manual analysis that automatic classification requires additional knowledge (*i.e.* simply relying on the currently used static and dynamic data is not enough). Furthermore, we found several intricate SUTS patterns in the CSGs, for which we could not make informed refactoring suggestions because of their complexity and size.

Lastly, there are future possible directions for further research. One is that we could identify and automatically recognize patterns, and then propose an appropriate refactoring solution for them. Another might be the investigation of some methods that simplifies the recognition of the graph patterns, even the complex ones, where possible. The class level hierarchy and traceability relations might also be worth investigating to see whether they can provide relevant information that would help us to identify Structural Unit Test Smells.

Acknowledgement

The project was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

[1] Atlassian. Clover Java and groovy code coverage tool homepage. `https://www.atlassian.com/software/clover/overview`. Last visited: 2017-04-06.

[2] Balogh, Gergő, Gergely, Tamás, Beszédes, Árpád, and Gyimóthy, Tibor. Are my unit tests in the right package? In *Proceedings of 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'16)*, pages 137–146. IEEE, October 2016.

[3] Bavota, Gabriele, Qusef, Abdallah, Oliveto, Rocco, De Lucia, Andrea, and Binkley, David. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012.

[4] Beck, Kent, editor. *Test Driven Development: By Example.* Addison-Wesley Professional, 2002.

[5] Bertolino, Antonia. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[6] Blondel, Vincent D, Guillaume, Jean-Loup, Lambiotte, Renaud, and Lefebvre, Etienne. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P1000, 2008.

[7] Bouillon, Philipp, Krinke, Jens, Meyer, Nils, and Steimann, Friedrich. Ezunit: A framework for associating failed unit tests with potential programming

errors. *Agile Processes in Software Engineering and Extreme Programming*, pages 101–104, 2007.

[8] Breugelmans, Manuel and Van Rompaey, Bart. Testq: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[9] Bruntink, Magiel and Van Deursen, Arie. Predicting class testability using object-oriented metrics. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 136–145. IEEE, 2004.

[10] De Lucia, Andrea, Fasano, Fausto, and Oliveto, Rocco. Traceability management for impact analysis. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 21–30. IEEE, 2008.

[11] Demeyer, Serge, Ducasse, Stéphane, and Nierstrasz, Oscar. *Object-oriented reengineering patterns.* Elsevier, 2002.

[12] Deursen, A. van, Moonen, L., Bergh, A. van den, and Kok, G. Refactoring test code. In Succi, G., Marchesi, M., Wells, D., and Williams, L., editors, *Extreme Programming Perspectives*, pages 141–152. Addison-Wesley, 2002.

[13] Feathers, Michael. *Working effectively with legacy code.* Prentice Hall Professional, 2004.

[14] Fortunato, Santo. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010.

[15] Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[16] Gaelli, Markus, Lanza, Michele, and Nierstrasz, Oscar. Towards a taxonomy of SUnit tests. In *Proceedings of 13th International Smalltalk Conference (ISC'05)*, 2005.

[17] Hamill, Paul. *Unit Test Frameworks: Tools for High-Quality Software Development.* O'Reilly Media, Inc., 2004.

[18] Horváth, Ferenc, Vancsics, Béla, Vidács, László, Beszédes, Árpád, Tengeri, Dávid, Gergely, Tamás, and Gyimóthy, Tibor. Test suite evaluation using code coverage based metrics. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15)*, pages 46–60, October 2015. Also appeared in CEUR Workshop Proceedings, Vol. 1525.

[19] Kanstrén, Teemu. Towards a deeper understanding of test coverage. *Journal of Software: Evolution and Process*, 20(1):59–76, 2008.

[20] Meszaros, Gerard. *xUnit test patterns: Refactoring test code.* Pearson Education, 2007.

[21] Myers, Glenford J, Sandler, Corey, and Badgett, Tom. *The art of software testing*. John Wiley & Sons, 2011.

[22] Qusef, Abdallah, Bavota, Gabriele, Oliveto, Rocco, De Lucia, Andrea, and Binkley, Dave. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168, 2014.

[23] Rompaey, B. V. and Demeyer, S. Establishing traceability links between unit test cases and units under test. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 209–218, March 2009.

[24] Spanoudakis, George and Zisman, Andrea. Software traceability: a roadmap. *Handbook of Software Engineering and Knowledge Engineering*, 3:395–428, 2005.

[25] Tengeri, Dávid, Beszédes, Árpád, Havas, Dávid, and Gyimóthy, Tibor. Toolset and program repository for code coverage-based test suite analysis and manipulation. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, pages 47–52. IEEE, September 2014.

[26] The JUnit Team. JUnit Java unit test framework homepage. `http://junit.org/`. Last visited: 2017-11-10.

[27] Van Rompaey, Bart and Demeyer, Serge. Exploring the composition of unit test suites. In *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 11–20. IEEE, 2008.

[28] Van Rompaey, Bart, Du Bois, Bart, and Demeyer, Serge. Characterizing the relative significance of a test smell. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 391–400. IEEE, 2006.

Received 30th May 2018