**Noname manuscript No.**
(will be inserted by the editor)

# Differences Between a Static and a Dynamic Test-to-Code Traceability Recovery Method

**Tamás Gergely · Gergő Balogh · Ferenc Horváth · Béla Vancsics · Árpád Beszédes · Tibor Gyimóthy**

Received: date / Accepted: date

**Abstract** Recovering test-to-code traceability links may be required in virtually every phase of development. This task might seem simple for unit tests thanks to two fundamental unit testing guidelines: isolation (unit tests should exercise only a single unit) and separation (they should be placed next to this unit). However, practice shows that recovery may be challenging because the guidelines typically cannot be fully followed. Furthermore, previous works have already demonstrated that fully automatic test-to-code traceability recovery for unit tests is virtually impossible in a general case.

In this work, we propose a semi-automatic method for this task, which is based on computing traceability links using static and dynamic approaches, comparing their results and presenting the discrepancies to the user, who will determine the final traceability links based on the differences and contextual information. We define a set of discrepancy patterns, which can help the user in this task. Additional outcomes of analyzing the discrepancies are structural unit testing issues and related refactoring suggestions. For the static test-to-code traceability, we rely on the physical code structure, while for the dynamic, we use code coverage information. In both cases, we compute combined test and code clusters which represent sets of mutually traceable elements. We also present an empirical study of the method involving 8 non-trivial open source Java systems.

This work was partially supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences and project number EFOP-3.6.3-VEKOP-16-2017-0002, which is co-funded by the European Social Fund.
The final publication is available at Springer via http://dx.doi.org/10.1007/s11219-018-9430-x.

Tamás Gergely · Gergő Balogh · Ferenc Horváth · Béla Vancsics · Árpád Beszédes · Tibor Gyimóthy
Department of Software Engineering, University of Szeged, Szeged, Hungary
E-mail: {gertom,geryxyz,hferenc,vancsics,beszedes,gyimothy}@inf.u-szeged.hu

Gergő Balogh · Tibor Gyimóthy
MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, Szeged, Hungary
E-mail: {geryxyz,gyimothy}@inf.u-szeged.hu

**Keywords** Test-to-code traceability, traceability link recovery, unit testing, code coverage, structural test smells, refactoring.

## 1 Introduction

Unit testing is an important element of software quality assurance (Black et al (2012)). In software maintenance and evolution, unit tests also play a crucial role: during regression testing unit tests are constantly re-executed and further evolved in parallel to the system under test (Feathers (2004)). Hence, their quality in general, and maintainability in particular are important. One aspect of maintainability is to establish reliable *traceability links* between unit test cases and units under test – which is the main theme of the present research.

There are a number of design patterns as well as testing frameworks available to aid programmers and testers to write good and easily traceable unit tests (Hamill (2004)). In particular, there are two unit testing guidelines that deal with the structural consistency between test and production code, which enable full traceability. The first one is *isolation*, which means that unit tests should exercise only the unit they were designed for, the second one being *separation*, meaning that the tests should be placed in the same logical or structural group (*e.g.* package) as the units they are testing. However, there are some practical aspects that act as barriers to design tests that completely conform to these text book definitions (*e.g.* calls to utility functions or other general parts of the system, see Bertolino (2007); Myers et al (2011)). The result is that unit test-to-code traceability information often cannot be directly derived from the code, and specific recovery effort needs to be made.

Recovering traceability links between unit test cases and units under test may be necessary in virtually every phase of development. For example, in the coding and testing phases, finding low level code related defects early is essential using appropriate unit tests, and in evolution, up to date unit tests are necessary for efficient regression testing. Several approaches have been proposed for this task (*e.g.* Qusef et al (2014); Gaelli et al (2005); Kanstrén (2008); Bruntink and Van Deursen (2004); Bouillon et al (2007)). However, practice shows that the fully automatic recovery of traceability links is difficult, and the different approaches might produce different results (Rompaey and Demeyer (2009); Qusef et al (2010)). Hence, combined or semi-automatic methods are often proposed for this task.

In this work, we present a semi-automatic method for unit test traceability recovery. In the first phase, we compute the traceability links based on two fundamentally different but very basic aspects: (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 harder to precisely express. For computing the static structural clusters we use the packaging structure of the code (referred to as *package based clusters*), while for the dynamic clustering we employ community detection (Blondel et al (2008)) on the code coverage information (called the *coverage based clusters*).

In the next phase, these two kinds of clusterings are compared to each other. If both approaches produce the same clusterings we conclude that the traceability links are reliable. However, in many cases there will be discrepancies in the produced results which we report as inconsistencies. There may be various reasons for these discrepancies but they are usually some combination of violating the isolation and/or separation principles mentioned above.

The final phase of the approach is then to analyze these discrepancies and, based on the context, produce the final recovered links. During this analysis, it may turn out that there are structural issues in the implemented tests and/or code, hence refactoring suggestions for the tests or code may be produced as well.

Because it usually involves other contextual information, the analysis phase needs to be done manually. To aid this process, we perform an automatic comparison and report the discrepancies in form of concrete patterns. In this paper, we introduce several such patterns, the details of their identification, and potential implications on the final traceability recovery.

We report on an empirical study, in which we manually investigated the traceability discrepancies of eight non-trivial open source Java systems to explain their context and provide suggestions for the final traceability recovery and eventual refactorings.

The practical usability of the results is manifold. Existing systems with integrated unit tests can be analyzed to recover traceability links more easily and reliably as the method is based on automatic approaches which compute the links from different viewpoints, and their comparison can be done semi-automatically. The method can also help in identifying problematic, hard to understand and maintain parts of the test code, and it might help in preventive maintenance as well by providing corresponding refactoring options.

This paper is an extension of our previous study, Balogh et al (2016), which introduced our concept on traceability recovery. This paper extends the previous study by a detailed manual analysis phase, additional discrepancy patterns and their enhanced detection method using Neighbor Degree Distribution vectors.

The rest of the paper is organized as follows. The next section overviews some background information and related work, while Section 3 details our traceability recovery method. The empirical study employing our method is presented in Section 4, with the analysis of the detected discrepancy patterns in Section 5. We provide conclusions about the presented work and sketch possible future directions in Section 6.

## 2 Background and related work

There are different levels of testing, one of which is unit testing. Unit tests are very closely related to the source code and they aim to test separate code fragments (Black et al (2012)). This kind of test helps to find implementation errors early in the coding phase, reducing the overall cost of the quality assurance activity.

### 2.1 Unit testing guidelines and frameworks

Several guidelines exist that give hints on how to write good unit tests (*e.g.* Hamill (2004); Breugelmans and Van Rompaey (2008); Van Rompaey and Demeyer (2008); Meszaros (2007)). The two basic principles telling that unit tests should be isolated (test only the elements of the target component) and separated (physically or logically grouped, aligned with the tested unit), are mentioned by most of them. 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 the structure of the tested system. Several studies have examined various characteristics of the source code with which the above mentioned two aspects can be measured and are verifiable to some extent, *e.g.* by Rompaey and Demeyer (2009).

These two properties are important for our approach too. Namely, if both are completely followed, the package and coverage based automatic traceability analysis algorithms will produce the same results. However, this is not the case in realistic systems, so our approach relies on analyzing the differences in the two sets to draw conclusions about the final traceability links.

A very important aspect to involve when unit testing research is concerned is the framework used to implement unit test suites. The most widely used is probably the xUnit family; in this work we consider Java systems and the JUnit toolset[1]. The features offered in the used framework highly determine the way unit testing is actually implemented in the project.

### 2.2 Traceability recovery in unit tests

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 (Spanoudakis and Zisman (2005); De Lucia et al (2008)). The approaches include static and dynamic code analysis, heuristic methods, information retrieval, machine-learning, and data mining based methods.

In this work, we are concerned with *test-to-code* traceability. The purpose of recovering this is to assign test cases to code elements based on the relationship that shows which code parts are tested by which tests. This information may be very important in development, testing or maintenance, as already discussed.

In this work, we concentrate on unit tests, in which case the traceability information is mostly encoded in the source code of the unit test cases, and usually no external documentation is available for this purpose. Traceability recovery for unit test may seem simple at first (Beck (2002); Demeyer et al (2002); Gaelli et al (2007)), however, in reality it is not (Gaelli et al (2005); Kanstrén (2008)).

Bruntink and Van Deursen (2004) illustrated the need and complexity of the test-to-code traceability. They investigated factors of the testability of Java systems. The authors concluded that the classes dependent upon other classes required more test code, and suggested the creation of composite test scenarios for the dependent classes. Their solution heavily relies on the test-to-code traceability relations.

[1] http://junit.org/ (last visited: 2018-06-26)

Bouillon et al (2007) presented an extension to the JUnit Eclipse plug-in. It used static call graphs to identify the classes under test for each unit test case and analyzed the comments to make the results more precise.

Rompaey and Demeyer (2009) evaluated the potential of six traceability resolution strategies (all are based on static information) for inferring relations between developer test cases and units under test. The authors concluded that no single strategy had high applicability, precision and recall. Strategies such as *Last Call Before Assert*, *Lexical Analysis* or *Co-Evolution* had a high applicability, but lower accuracy. However, combining these approaches with strategies relying on developer conventions (*e.g.* naming convention) and utilizing program specific knowledge (*e.g.* coding conventions) during the configuration of the methods provided better overall result.

Qusef et al (2014, 2010) proposed an approach (SCOTCH+ – Source code and COncept based Test to Code traceability Hunter) to support the developers during the identification of connections between unit tests and tested classes. It exploited dynamic slicing and textual analysis of class names, identifiers, and comments to retrieve traceability links.

In summary, most of the mentioned related works emphasize that reliable test-to-code traceability links are difficult to derive from a single source of information, and combined or semi-automatic methods are required. Our research follows this direction as well.

### 2.3 Test smells

The discrepancies found in the two automatic traceability analysis results can be seen as some sort of smells, which indicate potential problems in the structural organization of tests and code. For tests that are implemented as executable code, Deursen et al (2002) introduced the concept of *test smells*, which indicate poorly designed test code, and listed 11 test code smells with suggested refactorings. We can relate our work best to their *Indirect Testing* smell. Meszaros (2007) expanded the scope of the concept by describing test smells that act on a behavior or a project level, next to code-level smells. Results that came after this research use these ideas in practice. For example, Breugelmans and Van Rompaey (2008) 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 could understand the structure and quality of the test suite of large systems more easily (see Van Rompaey and Demeyer (2008)).

Our work significantly differs from these approaches as we are not concerned in code-oriented issues in the tests but in their dynamic behavior and relationship to their physical placement with respect to the system as a whole. We can identify *structural test smells* by analyzing the discrepancies found in the automatic traceability analyses.

### 2.4 Clustering in software

One of the basic components of the approach presented in this work is the identification of interrelated groups of tests and tested code. Tengeri et al (2015) proposed

an approach to group related test and code elements together, but this was based on manual classification. In the method, various metrics are computed and used as general indicators of test suite quality, and later it has been applied in a deep analysis of the WebKit system (Vidács et al (2016)).

There are various approaches and techniques for automatically grouping different items of software systems together based on different types of information, for example, code metrics, specific code behavior, or subsequent changes in the code (Pietrzak and Walter (2006)). Mitchell and Mancoridis (2006) examined the Bunch clustering system which used search techniques to perform clustering. The ARCH tool by Schwanke (1991) determined clusters using coupling and cohesion measurements. The Rigi system by Müller et al (1993) pioneered the concepts of isolating omnipresent modules, grouping modules with common clients and suppliers, and grouping modules that had similar names. The last idea was followed up by Anquetil and Lethbridge (1998), who used common patterns in file names as a clustering criterion.

Our coverage-based clustering method uses a community detection algorithm, which was successfully used in other areas (like biology, chemistry, economics and engineering) previously. Recently, efficient community detection algorithms have been developed which can cope with very large graphs (see Blondel et al (2008)). Application of these algorithms to software engineering problems is emerging. Hamilton and Danicic (2012) introduced the concept of *dependence communities* on program code and discussed their relationship to program slice graphs. They found that dependence communities reflect the semantic concerns in the programs. Šubelj and Bajec (2011) applied community detection on classes and their static dependencies to infer communities among software classes. We performed community detection on method level, using dynamic coverage information as relations between production code and test case methods, which we believe is a novel application of the technique.

### 2.5 Comparison of clusterings

Our method for recovering traceability information between code and tests is to compute two types of clusterings on test and code (these represent the traceability information) and then compare these clusterings to each other. In literature, there were several methods proposed to perform this task. A large part of the methods use various similarity measures, which express the differences using numerical values. Another approach to compare clusterings is to infer a sequence of transformation actions which may evolve one clustering into the other. In our approach, we utilize elements of both directions: we use a similarity measure to express the initial relationships between the clusters and then define discrepancy patterns that rely on these measures, which can finally lead to actual changes (refactorings) in the systems. In this section, we overview the most imporant related approaches in these areas.

Wagner and Wagner (2007) provide a list of possible measures for comparing clusters, noting that different measures may prefer or disregard various properties of the clusters (*e.g.* their absolute size). There are several groups of these measures based on whether they are symmetric (*e.g.* Jaccard index) or non-symmetric (*e.g.* Inclusion measure), count properties of the element relations (Chi Squared Coef-

ficient) or that of the clusters (F-Measure), etc. We decided to use the *Inclusion measure* in this work because, in our opinion, it is the most intuitive one which can help in the validation of the results, and can provide more information about the specific cases. We experimented with other measures as well but the empirical measurements showed similar results.

Palla et al (2007) analyzed community evolution of large databases, while Bóta et al (2011) investigated the dynamics of communities, which are essentially ways to describe the evolution of clusterings in general. In addition to the cluster comparison, the authors described what types of changes can occur in the clusters that will eventually lead to a sequence of transformation actions between the different clusterings. These actions are very similar to the refactorings related to tests and code we propose in this work, which can potentially resolve discrepancies found between the two examined traceability information types.

## 3 Method for semi-automatic traceability recovery

In this section, we describe our semi-automatic method for unit test traceability recovery and structural test smell identification.

### 3.1 Overview

Figure 1 shows an overview of the process, which has several sequential phases. First, the physical organization of the *production* and *test code* into Java packages is recorded, 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.

These will represent the two types of relationships between test and code: the two sets of automatically produced traceability links from two viewpoints, static and dynamic. Both clusterings produce sets of clusters that are composed 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), while a unit under test is a regular Java method.

In our approach, the elements of a cluster are mutually traceable to each other, and no individual traceability is established between individual test cases and units. The benefit of this is that in many cases it is impossible to uniquely assign a test case to a unit, rather *groups* of test cases and units can represent a cohesive functional unit (Horváth et al (2015)). Also, minor inconsistencies, such as helper methods that are not directly tested, are "smoothed away" with this procedure.

Details about the clustering based traceability algorithms are provided in Section 3.2.

The automatically produced traceability links will be compared using a helper structure, the *Cluster Similarity Graph* (CSG) introduced by Balogh et al (2016). This is a directed bipartite graph whose nodes represent the clusters of the two clusterings. Edges of the graph are used to denote the level of similarity between the two corresponding clusters. For this, a pairwise similarity measure is used as a label on each edge. In particular, we use the *Inclusion measure* for two clusters of

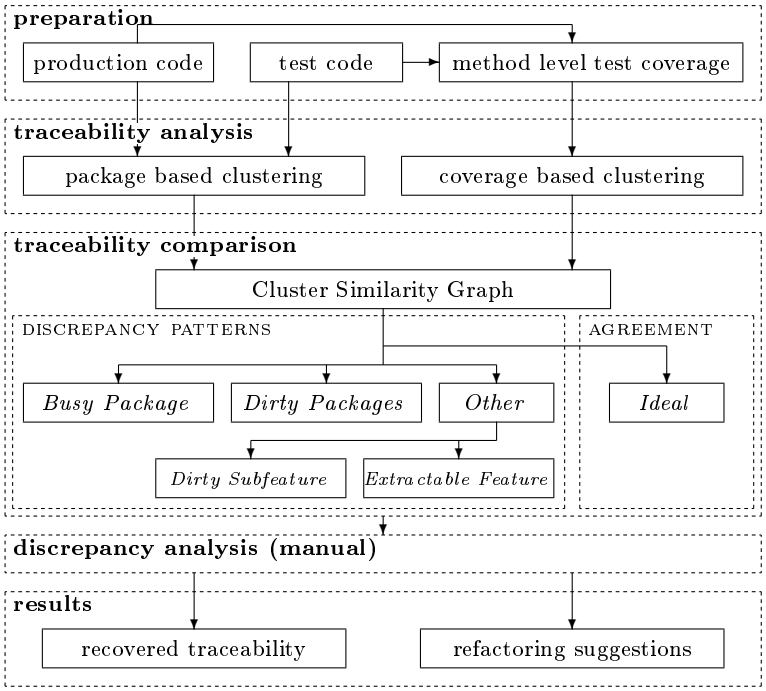


**Fig. 1** Overview of the method

different type, $K_1$ and $K_2$, to express to what degree $K_1$ is included in $K_2$. Value 0 means no inclusion, while 1 means that $K_2$ fully includes $K_1$. We initially omit edges with 0 inclusion value, which makes the CSG much smaller than a complete bipartite graph.

Figure 2 shows an example CSG, taken from one of our subject systems, `oryx`. In it, the package based clusters can be identified by dark grey rectangles with the name of the corresponding package, while the coverage based clusters are light grey boxes with a unique serial number. To simplify the example, we removed the labels denoting the weights of the edges. In a package based cluster, the code and test items from the same package are grouped together. For example, there is a package called *com/cloudera/oryx/common* with several subpackages, such as *.../math* and *.../collection*, which can be found in the CSG graph as well. Coverage based clusters are labeled by simple ordinal identifiers because no specific parts of the code can be assigned to them in general, which would provide a better naming scheme. The details on comparing clusters are given in Section 3.3.

The next step in our process is the analysis of the CSG to identify the distinct patterns, which describe various cases of the agreement and the discrepancies between the two automatic traceability results. Clearly, agreement between the two sets of results (an ideal correspondence between some parts of the two clusterings) can be captured as a pair of identical package and coverage based clusters that are not related to any other cluster by similarity (called the *Ideal* pattern). In essence, all other patterns in the CSG can be seen as some kind of a discrepancy needing further analysis.

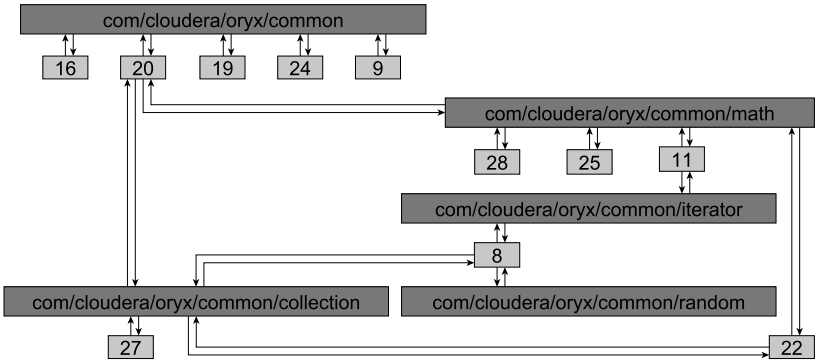


**Fig. 2** A part of the CSG of the `oryx` program

Each discrepancy type can be described using a pattern consisting of at least two interconnected cluster nodes of the CSG, for which some additional properties also hold. For example, the coverage based cluster 9 in Figure 2 is connected to the package based cluster *com/cloudera/oryx/common*, but this has connections to other coverage based clusters as well. This is an instance of the *Extractable Feature* pattern, which means that cluster 9 captures some kind of a subfeature, but this is not represented as a separate package. In this case, the dynamic traceability analysis provides more accurate traceability links than the static one. It may suggest a possible refactoring as well, namely to move the identified subfeatures to a separate package.

The corresponding block in Figure 1 shows a classification of the proposed discrepancy patterns. The detailed elaboration of the patterns will be given in Section 3.4.

The final phase of our method is the analysis of the traceability comparison with the goal to produce the final recovered traceability. Clearly, in the case of agreement we obtain the results instantly, but in many cases the detected discrepancies need to be analyzed to make informed decisions. The goal of analyzing the discrepancies is to provide explanations to them, select the final recovered traceability links and possibly give refactoring suggestions.

The decision "which of the traceability links are more reliable in the case of a particular discrepancy" is typically context dependent. This is because a discrepancy may reflect that either the isolation or the separation principles are violated, or both. In some cases these violations can be justified, and then one of the traceability types better reflects the intentions of the developers. In other cases they can actually reflect structural issues in the code that need to be refactored. In any case, the analysis needs to be manual because it requires the understanding of the underlying intentions of the programmers to some extent. It is left for future work to investigate if this process can be further automated in some way.

The results of our empirical study with the analysis of our benchmark systems is presented in Section 5.

## 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. In this work, we are dealing 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 aggregated into classes and classes into packages. All of our algorithms have the granularity level of methods, *i.e.* clusters are composed of production and test methods. We do not explicitly take into account the organization of methods into classes.

### *3.2.1 Package based clustering*

Through package based clustering, our aim 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 a hierarchical package structure 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 important that the physical organization of the code and tests is reliable and reflects the developer's intentions.

Our package based clustering simply means that we assign the fully qualified name of the containing package to each production and test method, and treat methods (of both types) belonging to the same package members of the same cluster. Class information and higher level package hierarchy are not directly taken into account or, in other words, the package hierarchy is flattened. For example, the package called `a.b.c` and its subpackages `a.b.c.d` and `a.b.c.e` are treated as unique clusters containing all methods of all classes directly contained within them, respectively. Furthermore, we do not consider the physical directory and file structure of the source code elements as they appear in the file system (although in Java, these usually reflect package structuring).

### *3.2.2 Coverage based clustering*

In order to determine the clustering of tests and code based on the actual dynamic behavior of the test suite, we apply *community detection* (Blondel et al (2008); Fortunato (2010)) on the code coverage information.

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 *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 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 that test case.

*Community detection* algorithms were originally defined on (possibly directed and weighted) graphs that represent complex networks (social, biological, technological, etc.), and recently have also been suggested for software engineering problems (*e.g.* by Hamilton and Danicic (2012)). Community structures are detected based on statistical information about the number of edges between sets of graph nodes. Thus, in order to use the chosen algorithm, we construct a graph from the coverage matrix, whose nodes are the methods and tests of the analyzed system (referred to as the *coverage graph* in the following). This way, we define a bipartite graph over the method and test sets because no edge will be present between two methods or two tests. Note, that for the working of the algorithm, this property will not be exploited, *i.e.* each node is treated uniformly in the graph for the purpose of community detection.

The actual algorithm we used for community detection is the Louvain Modularity method (Blondel et al (2008)). It is a greedy optimization method based on internal graph structure statistics to maximize modularity. The modularity of a clustering is a scalar value between $-1$ and 1 that measures the density of links inside the clusters as compared to links among the clusters. The algorithm works iterative, and each pass is composed of two phases. In the first phase it starts with each node isolated in its own cluster. Then it iterates through the nodes $i$ and its neighbors $j$, checking whether moving node $i$ to the cluster of $j$ would increase modularity. If the move of node $i$ that results in the maximum gain is positive, then the change is made. The first phase ends when no more moves result in positive gain. In the second phase, a new graph is created hierarchically by transforming the clusters into single nodes and creating and weighting the edges between them according to the sum of the corresponding edge weights of the original graph. Then the algorithm restarts with this new graph, and repeats the two phases until no change would result in positive gain.

### 3.3 Comparing clusterings

At this point, we have two sets of traceability links in form of clusters on the tests and code items: one based on the physical structure of the code and tests (package based clusters, denoted by $P$ clusters in the following), and another one based on the coverage data of the tests showing their behavior (coverage based clusters, denoted by $C$ clusters). To identify deviations in the two sets, the corresponding clusterings need to be compared. As mentioned earlier, there are two major ways to compare clusterings: by expressing the similarity of the clusters or by capturing actions needed to transform one clustering to the other. We follow the similarity measure based approach in this phase.

We use a non-symmetric similarity measure, the *Inclusion measure*. We empirically verified various other similarity measures, like *Jaccard*, and found that they were similar to each other in expressing the differences. Finally we choose the Inclusion measure, because, in our opinion, it is more intuitive, helping the validation of the results, and it provides more information that could be potentially utilized in the future.

*Inclusion measure* expresses to what degree a cluster is included in another one. A value $v \in [0, 1]$ means that the $v$-th part of a cluster is present in the other cluster, 0 meaning no inclusion (empty intersection) and 1 meaning full

inclusion (the cluster is a subset of the other one). The measure is computed for two arbitrary clusters of different type, $K_1$ and $K_2$, as:

$$I(K_1, K_2) = \frac{|K_1 \cap K_2|}{|K_1|}.$$

Using the measure, we can form a weighted complete bipartite directed graph (the *cluster similarity graph*, CSG). We decided to use a graph representation because it is, being visual, more readable by humans. This was an important aspect because our approach includes a manual investigation of the automatically identified traceability information. Nodes of CSG are the $C$ and $P$ clusters, and an edge from $P_i$ to $C_j$ is weighted by $I(P_i, C_j)$, while the reverse edge from $C_j$ to $P_i$ has $I(C_j, P_i)$ as its weight. Edges with a weight of 0 are omitted from the graph.

### 3.4 Discrepancy patterns

Now, we can use the CSG to check the similarity or search for discrepancies between the two clusterings. We have defined six different patterns: one pattern showing the ideal cluster correspondence (hence, traceability agreement, called the *Ideal* pattern) and five types of discrepancies, *Busy Package*, *Dirty Packages*, *Other*, and the two special cases of *Other*, *Extractable Feature* and *Dirty Subfeature*. Examples for these are shown in Figure 3.

Each pattern describes a setting of related $C$ and $P$ clusters with a specific set of inclusion measures as follows:

*Ideal* Here, the pair of $C$ and $P$ clusters contain the same elements, and there are no other clusters that include any of these elements (the inclusion measures are 1 in both directions). This is the ideal situation, which shows that there is an agreement between the two traceability analysis methods.

*Busy Package* This discrepancy describes a situation in which a $P$ cluster splits up into several $C$ clusters (the sum of inclusion measures $I(P, C_i)$ is 1), and each $C$ cluster is included completely in the $P$ cluster (their inclusions are 1). This is a clear situation in which the coverage based clustering captures isolated groups of code elements with distinct functionality, while the physical code structure is more coarse-grained. In practice, this means that package $P$ could be split into subpackages that reflect the distinct functionalities, and that probably the coverage based traceability links are more appropriate.

*Dirty Packages* This pattern is the opposite of the previous one: one $C$ cluster corresponds to a collection of $P$ clusters, and there are no other clusters involved (sum of $I(C, P_i)$ is 1 and each $I(P_i, C)$ is 1). In practice, this means that the dynamic test-to-code relationship of a set of packages is not clearly separable, they all seem to represent a bigger functional unit. This might be due to improper isolation of the units or an overly fine-grained package hierarchy, and could be remedied either by introducing mocking or package merging. Another explanation for this situation could be that the tests are higher level tests that involve not

only one unit and are, for example, integration tests. Depending on the situation, either package based or coverage based traceability links could be more reliable in this case.

*Other* The last category is practically the general case, when neither of the above more specific patterns could be identified. This typically means a mixed situation, and requires further analysis to determine the possible cause and implications. The inclusion measures can help, however, to identify cases which are not pure instances of *Busy Package* or *Dirty Packages* but are their partial examples. The final two patterns we define are two specific cases of this situation:

*Extractable Feature* This pattern refers to a case when there are $C$ clusters that are parts of a pattern which resembles *Busy Package*, but the related $P$ package has some other connections as well, not qualifying the pattern for *Busy Package*. Since these $C$ clusters capture some kind of a subfeature of the connected packages, but they are not represented in a distinct package, we call them *Extractable Feature*. Our example from Figure 2 includes a few instances of this pattern, namely $C$ clusters 9, 16, 19, 24, 25, 27 and 28.

*Dirty Subfeature* Similarly, *Other* patterns can have a subset, which can be treated as special cases of $P$ clusters in the *Dirty Packages* pattern. They are connected to a $C$ cluster which forms an imperfect *Dirty Packages* pattern. The name *Dirty Subfeature* suggests that this $P$ cluster implements a subfeature but its tests and code are dynamically connected to external elements as well. The example in Figure 2 includes one instance of this pattern: *com/cloudera/oryx/common/random*.

### *3.4.1 Pattern search*

Searching for the patterns themselves is done in two steps. First, a new helper data structure is computed for each cluster, the *Neighbor Degree Distribution* (NDD) vector. This vector describes the relationships of the examined CSG node to its direct neighbors. For an arbitrary cluster $K$ of any type, the $i^{th}$ element of NDD($K$) shows how many other clusters with degree $i$ are connected to $K$:

$$\begin{aligned} \mathrm{NDD}(K) &= (d_1, d_2, \ldots, d_n), \text{ where} \\ n &= \max\left(|\{\text{P clusters}\}|, |\{\text{C clusters}\}|\right) \\ d_i &= \left|\left\{K' : I(K, K') > 0 \wedge \deg(K') = i\right\}\right| \end{aligned}$$

Here, the *degree* of a cluster, $\deg(K)$, is the number of incoming non-zero weighted edges of the cluster, *i.e.* the number of other clusters that share elements with the examined one, and $n$, the length of the vector is the maximal possible degree in the graph. The example cluster nodes from Figure 3 include the corresponding NDD vectors.

In the second step of pattern search, specific NDD vectors are located as follows. Cluster pairs with $(1, 0, 0, \ldots)$ NDD vectors are part of an *Ideal* pattern, so they are reported first. Each discrepancy pattern is then composed of several clusters of both types, which need to have NDD vectors in the following combination:

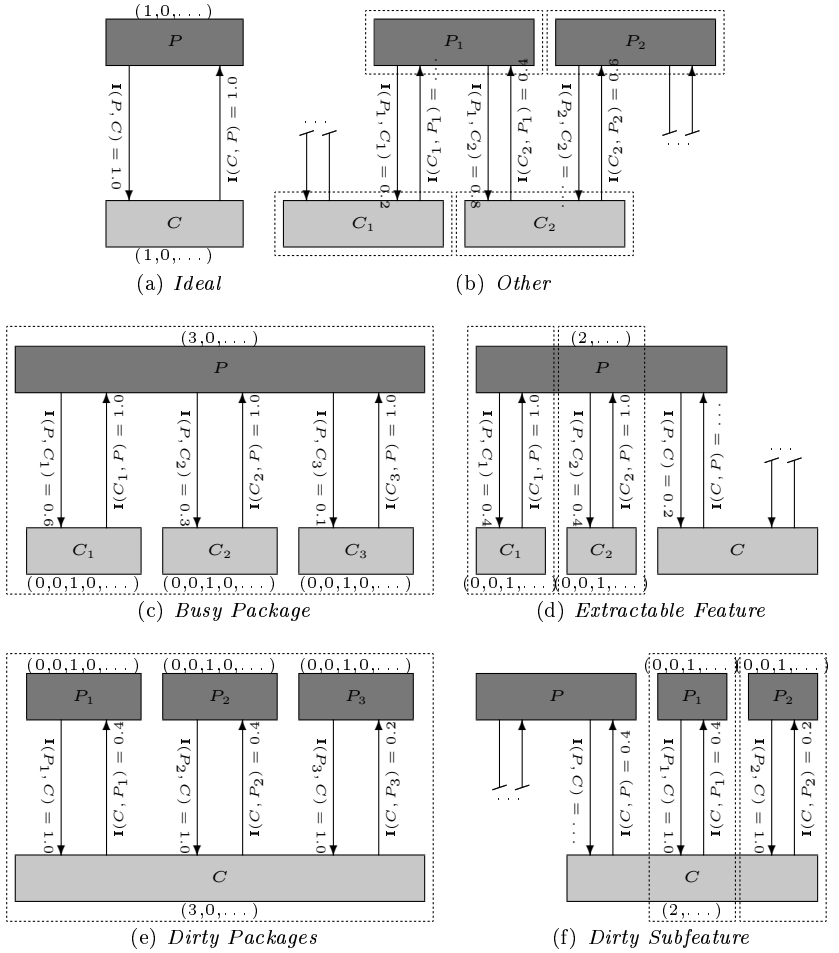


**Fig. 3** Patterns recognized in the Cluster Similarity Graph. The dashed areas represent the pattern instances.

- *Busy Package* patterns are composed of a $P$ cluster with $(x, 0, 0, \dots)$ vector, where $x > 1$, and exactly $x$ number of related $C$ clusters with NDD vectors of the form $(0, 0, \dots, d_x, 0, 0, \dots)$, $d_x = 1$.
- *Dirty Packages* can be identified in the same way as *Busy Package*, but with the roles of $P$ and $C$ exchanged.
- In the case of *Extractable Feature*, we are looking for $C$ clusters with NDD vectors of the form $(0, 0, \dots, 1, 0, 0, \dots)$, which are not part of regular *Busy Package* patterns.
- *Dirty Subfeature* patterns can be detected by locating $P$ clusters with NDD vectors $(0, 0, \dots, 1, 0, 0, \dots)$, which are not part of any *Dirty Packages*.
- Finally, all other clusters not participating in any of the above are treated as parts of a general *Other* discrepancy pattern.

In the case of *Other* patterns (including the two subtypes) reporting gets more complicated as we need to decide what parts of the CSG should be included as one specific pattern instance. To ease the analysis, we report an instance for each participating cluster individually (see the dashed parts in the example in Figure 3).

## 4 Empirical study

The usefulness of the method described above has been verified in an empirical study presented in this section, with the details of manual analysis of discrepancies found in the next one. The study involved three research questions:

- **RQ1:** In a set of open source projects, which use JUnit-based unit tests, how different are the traceability links identified by the two automatic cluster based approaches? In particular, how many discrepancy pattern instances in the traceability results can we detect using the method described in the previous section?
- **RQ2:** What are the properties and possible reasons for the found discrepancy pattern instances? This includes the identification of any structural test bad smells and refactoring suggestions as well. We address this question by manual analysis.
- **RQ3:** What general guidelines can we derive about producing the final recovered traceability links and possible refactoring suggestions?

To answer these questions, we conducted an experiment in which we relied on 8 non-trivial open source Java programs. On these programs we executed all phases of the method, including a detailed manual investigation of the traceability links and discrepancy patterns generated by the automatic methods.

### 4.1 Subject programs and detection framework

Our subject systems were medium to large size open source Java programs which have their unit tests implemented using the JUnit test automation framework. Columns 1–5 of Table 1 show some of their basic properties. We chose these systems because they had a reasonable number of test cases compared to the system size. We modified the build processes of the systems to produce method level coverage information using the Clover coverage measurement tool[2].This tool is based on source-code instrumentation and gives more precise information about source code entities than tools based on bytecode instrumentation (Tengeri et al (2016)).

[2] https://www.atlassian.com/software/clover (last visited: 2018-06-26)
[3] https://github.com/checkstyle/checkstyle (last visited: 2018-06-26)
[4] https://github.com/apache/commons-lang (last visited: 2018-06-26)
[5] https://github.com/apache/commons-math (last visited: 2018-06-26)
[6] https://github.com/JodaOrg/joda-time (last visited: 2018-06-26)
[7] https://github.com/jankotek/mapdb (last visited: 2018-06-26)
[8] https://github.com/netty/netty (last visited: 2018-06-26)
[9] https://github.com/Orientechnologies/orientdb (last visited: 2018-06-26)
[10] https://github.com/cloudera/oryx (considered obsolete, made private)

**Table 1** Subject programs and their basic properties

| Program | tag / hash | LOC | Methods | Tests | $P$ | $C$ |
|---|---|---|---|---|---|---|
| **checkstyle**[3] | *checkstyle-6.11.1* | 114K | 2 655 | 1 487 | 24 | 47 |
| **commons-lang**[4] | *#00fafe77* | 69K | 2 796 | 3 326 | 13 | 276 |
| **commons-math**[5] | *#2aa4681c* | 177K | 7 167 | 5 081 | 71 | 39 |
| **joda-time**[6] | *v2.9* | 85K | 3 898 | 4 174 | 9 | 22 |
| **mapdb**[7] | *mapdb-1.0.8* | 53K | 1 608 | 1 774 | 4 | 7 |
| **netty**[8] | *netty-4.0.29.Final* | 140K | 8 230 | 3 982 | 45 | 35 |
| **orientdb**[9] | *2.0.10* | 229K | 13 118 | 925 | 130 | 39 |
| **oryx**[10] | *oryx-1.1.0* | 31K | 1 562 | 208 | 27 | 40 |

For storing and manipulating the data, *e.g.* to process the coverage matrix, we used the SoDA framework by Tengeri et al (2014). Then, we implemented a set of Python scripts to perform clusterings, including a native implementation of the community detection algorithm, and the implementation of pattern search.

### 4.2 Cluster statistics

Once we produced the coverage data for the subjects, we processed the lists of methods and test cases and the detailed coverage data in order to extract package based clustering information and to determine coverage based clusters. The last two columns of Table 1 show the number of clusters found in the different subject programs. As can be seen, the proportion of $P$ and $C$ clusters is not balanced in any of the programs, and differences can be found in both directions.

However, by simply considering the number of clusters is not sufficient to draw any conclusions about their structure, let alone identify reliable traceability links and discrepancies. Consider, for instance, `orientdb`: here, we could not decide whether $P$ clusters are too small or the developers are using mocking and stubbing techniques inappropriately, hence the relatively low number of $C$ clusters. Or, if in the case of `commons-lang` where there are 21-times more $C$ that $P$ clusters, should the packages be split into smaller components? Are the package or the coverage based traceability links more reliable in these cases? To answer these kinds of questions, a more detailed analysis of the patterns found in the CSG is required.

### 4.3 Pattern counts

After determining the clusters, we constructed the CSGs and computed the NDD vectors for each program and cluster. Then, we performed the pattern search with the help of the vectors to locate the *Ideal* pattern and the four specific discrepancy patterns, *Busy Package*, *Dirty Packages*, *Dirty Subfeature* and *Extractable Feature* (for the *Other* pattern, we consider all other clusters not present in any of the previous patterns).

The second column of Table 2 shows the number of *Ideal* patterns the algorithm found for each subject (every instance involves one cluster of each type). As expected, generally there were very few of these patterns found. But purely based on this result, we might consider the last three programs better in following unit

**Table 2** Pattern counts – *Ideal*, *Busy Package* and *Dirty Packages*; columns 'count' indicate the number of corresponding patterns, columns '$C$ count' and '$P$ count' indicate the number of $C$ and $P$ clusters involved in each identified pattern

| Program | *Ideal* pattern count | *Busy Package* count | *Busy Package* $C$ count | *Dirty Packages* count | *Dirty Packages* $P$ count |
|---|---|---|---|---|---|
| **checkstyle** | 0 | 1 | {4} | 0 | |
| **commons-lang** | 0 | 0 | | 0 | |
| **commons-math** | 0 | 0 | | 0 | |
| **joda-time** | 0 | 0 | | 0 | |
| **mapdb** | 0 | 0 | | 0 | |
| **netty** | 4 | 1 | {2} | 0 | |
| **orientdb** | 2 | 0 | | 1 | {2} |
| **oryx** | 9 | 5 | {4,4,4,2,2} | 0 | |

testing guidelines than the other five programs. For instance, 1/3 of the packages in `oryx` include purely isolated and separated unit tests according to their code coverage. These instances can be treated as reliable elements of the final traceability recovery output.

Table 2 also shows the number of *Busy Package* and *Dirty Packages* patterns found in the subjects. Columns 3 and 5 count the actual instances of the corresponding patterns, *i.e.* the whole pattern is counted as one regardless of the number of participating clusters in it. The numbers in columns 4 and 6 correspond to the number of connected clusters in the respective instances. That is, for *Busy Package* it shows the number of $C$ clusters connected to the $P$ cluster, and in the case of *Dirty Packages* it is the number of connected $P$ clusters. We list all such connected cluster numbers in the case of `oryx`, which has more than one instance of this type.

It can be seen that there are relatively few discrepancy pattern instances in these two categories, and that the connected cluster numbers are relatively small as well. This suggests that the definitions of *Busy Package* and *Dirty Packages* might be too strict, because they require that there is a complete inclusion of the connected $C$ clusters and $P$ clusters, respectively. Cases when the corresponding pattern is present but there are some outliers will currently not be detected. This might be improved in the future by allowing a certain level of tolerance in the inclusion values on the CSG edges. For instance, by introducing a small threshold value below which the edge would be dropped, we would enable the detection of more patterns in these categories.

The biggest hit was the set of five *Busy Package* instances for `oryx`, and this, together with the 9 *Ideal* patterns for this program, leaves only 13 and 15 clusters to be present in the corresponding *Other* categories.

Table 3 shows the number of different forms of *Other* discrepancy patterns found, but in this case each participating cluster is counted individually (in other words, each cluster is individually treated as one pattern instance). Clusters participating in the *Other* pattern instances are divided into two groups, *P Other* and *C Other*, consisting of the package and coverage cluster elements, respectively. *Dirty Subfeature* and *Extractable Feature* are the two specific subtypes of *Other*, and as explained, the former are subsets of *P Other* clusters, and the latter of *C Other* clusters.

**Table 3** Pattern counts – *Other*; columns *P Other* and *C Other* indicate the number of clusters involved in these specific patterns, columns 'all' indicate the number of all involved clusters (including the specific ones)

| Program | *P Other* count | | *C Other* count | |
|---|---|---|---|---|
| | all | *Dirty Subfeature* | all | *Extractable Feature* |
| **checkstyle** | 23 | 3 | 43 | 29 |
| **commons-lang** | 13 | 1 | 276 | 260 |
| **commons-math** | 71 | 22 | 39 | 26 |
| **joda-time** | 9 | 1 | 22 | 14 |
| **mapdb** | 4 | 0 | 7 | 3 |
| **netty** | 40 | 30 | 29 | 17 |
| **orientdb** | 126 | 48 | 36 | 25 |
| **oryx** | 13 | 6 | 15 | 7 |

Due to the low number of *Busy Package* and *Dirty Packages* instances, the number of clusters participating in the *Other* category is quite high. Fortunately, a large portion of *Other* can be categorized as either *Dirty Subfeature* or *Extractable Feature*, as can be seen in Table 3.

This answers our **RQ1**, namely the quantitative analysis of the detected patterns.

## 5 Analysis of traceability discrepancies

We manually analyzed all discrepancy pattern instances found in the results produced by the static and dynamic traceability detection approaches. In this process, we considered the corresponding patterns in the CSGs, the associated edge weights, and examined the corresponding parts of the production and test code. In the first step, each subject system was assigned to one of the authors of the paper for initial comprehension and analysis of the resulted patterns. The analysis required the understanding of the code structure and the intended goal of the test cases to a certain degree. API documentation, feature lists, and other public information were also considered during this phase. Then, the researchers made suggestions on the possible recovered traceability links and eventual code refactorings. Finally, all the participants were involved in a discussion where the final decisions were made. The edge weights in the CSGs helped during the analysis 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 not affecting the overall structure of the clusters.

The results of the analysis were possible explanations for the reported discrepancies with concrete suggestions (answering **RQ2**), as well as the corresponding general guidelines for traceability recovery, structural test smells and refactoring possibilities (responding to **RQ3**). In this section, we first present the analysis for each pattern category, and then we summarize the recovery options in general terms.

### 5.1 Busy Package

The detection algorithm found 7 *Busy Package* patterns, 5 of which belong to `oryx` (see Table 2). We examined these patterns in detail and made suggestions on their traceability links and whether their code and test structure should be refactored.

In all the cases we found that the $C$ clusters produced more appropriate traceability relations. However, there was just a single package, *com/cloudera/oryx/-kmeans/common*, where we could suggest the refactoring of the package structure exactly as the $C$ clusters showed it. In three cases, *com/puppycrawl/tools/-checkstyle/doclets*, *com/cloudera/oryx/common/io*, and *com/cloudera/oryx/-kmeans/computation/covariance*, the split would result in very small packages; thus, although the $C$ clusters correctly showed the different functionalities, we suggested no change in the package structure but using the $C$ clusters for traceability recovery purposes.

In the case of the *io/netty/handler/codec/haproxy* and *com/cloudera/oryx/-common/stats* even the $C$ clusters could not entirely identify the separate functionalities of the packages. After the examination of these situations, we suggested in both case the refactoring of the package, namely, the split of it into more packages and also the refactoring of the tests to eliminate unnecessary calls that violate the isolation guideline. The last *Busy Package* pattern we found was *com/-cloudera/oryx/als/common*. Although the $C$ clusters correctly capture the traceability among the elements, forming packages from all of them would result in some very small packages. Thus, we suggested to split the package into three subpackages, two according to two $C$ clusters, and one for the remaining $C$ clusters.

### 5.2 Dirty Packages

The only *Dirty Packages* pattern the algorithm detected belongs to the subject `orientdb`. This pattern consists of a $C$ cluster and two connected $P$ clusters. One of the packages does not contain tests, but is indirectly tested by the other one.

Both package or coverage based traceability links could potentially be considered in this case. However, since the not tested package contains a bean-like class, it should probably not be mocked. Thus, merging the two packages would be a possible refactoring in this case, which, again, corresponds to the coverage based traceability links.

### 5.3 Other

All clusters not belonging to the *Ideal* pattern or any of the two previous discrepancy patterns are treated as *Other* patterns, and as we can observe from Table 3, there are many of them. We investigated these as well, and apart from the two specific subtypes, *Extractable Feature* and *Dirty Subfeature*, we identified two additional explanations in this category. However, these are difficult to precisely define and quantify, so we will present them only in general terms below.

*Extractable Feature* The last column of Table 3 shows the number of $C$ clusters that qualify for *Extractable Feature*, which is a quite big portion of the $C$ *Other*

cases. These patterns, as mentioned earlier, are usually simple to refactor by creating a new package for the corresponding extractable subfunctionality, similarly to *Busy Package*. Also, coverage based traceability should be considered in this case.

*Dirty Subfeature* Similarly, a subset of *P Other* clusters will form *Dirty Subfeature* patterns (see column 3 of Table 3). In this case, there can be different explanations and possible refactorings (they can be merged into other packages or the involved functionalities be mocked, as is the case with *Dirty Packages*). Consequently, either package based or coverage based traceability links should be used, depending on the situation.

*Utility Functions* The root cause for this issue is that the units are directly using objects and methods of other units, either called from the production code or from the tests, while they are not in sub-package relation with each other. In many cases, this is due to implementing some kind of a utility function that contains simple, general and independent methods used by large parts of the system. Mock implementation of these would usually be not much simpler than the real implementation, so these pieces of code should not be refactored, and the original package based traceability should be used.

*Lazy Subpackages* This is the case when a $C$ cluster is connected to more $P$ clusters, but these are related to each other by subpackaging in which the subpackages often do not contain test cases. In other words, the functionality is separated into subpackages but the tests are implemented higher in the package hierarchy. This pattern can be seen as a special case of the *Dirty Subfeature* pattern in which the coverage based traceability links should be used. Reorganization and merging of packages could be a viable refactoring in this case to better reflect the intended structure of the tests and code.

### 5.4 Summary of traceability recovery and refactoring options

Results from the manual analysis presented in the previous section suggested that the final traceability recovery options depend on the actual situation and are heavily context dependent. Therefore, it is difficult to set up general rules or devise precise algorithms for deciding which automatic traceability analysis approach is suitable in case of a discrepancy. Yet, there were typical cases which we identified and may serve as general guidelines for producing the final recovered links. Also, in most of the cases, we can suggest typical refactoring options for the particular discrepancy patterns. However, the question if a specific pattern should be treated as an actual refactorable smell or not, is difficult to generalize. In Table 4, we summarize our findings.

Generally speaking, handling *Busy Package* patterns is the simplest as it typically means that the coverage based traceability is clear, which is not reflected in the package organization of the tests and code. In this case, refactoring could also be suggested: to split the package corresponding to the $P$ cluster in the pattern into multiple packages according to the $C$ clusters. However, in many cases other properties of the system do not justify the actual modifications. These new

**Table 4** Traceability recovery and refactoring options

| Discrepancy pattern | Suggested traceability type | Possible refactoring |
|---|---|---|
| *Busy Package* | Coverage | Split package |
| *Dirty Packages* | Coverage | Merge packages |
| *Dirty Packages* | Package | Create mock objects |
| *Dirty Packages* | Mixed | Mixed |
| *Extractable Feature* | Coverage | Create new package |
| *Dirty Subfeature* | Coverage | Merge into package |
| *Dirty Subfeature* | Package | Create mock objects |
| *Utility Functions* | Package | None |
| *Lazy Subpackages* | Coverage | Merge packages |

packages could also be subpackages of the original one if it is not eliminated completely. The contents of the new packages are automatically provided based on the $C$ clusters in the discrepancy pattern, hence refactoring is straightforward and mostly automatic.

We identified two basic options for handling *Dirty Packages* if refactoring is required: merging the $P$ packages of the pattern (this is the case we found in our subject system) or eliminating the external calls between the $P$ packages by introducing mock objects. These two cases reflect also the two suggested traceability types to use: coverage based in the case of package merge and package based for mock objects. Merging means moving the contents of the packages into a bigger package (perhaps the container package), hence following the dynamic relationships indicated by the $C$ cluster. This option may be implemented in a mostly automatic way. However, eliminating the calls between the $P$ packages is not so trivial, as it requires the introduction of a mock object for each unit test violating the isolation principle. In many cases, the solution might be a mixed one in which also package reorganization and mocking is done, depending on other conditions. In this case, the final traceability recovery will be mixed as well. Also, in some cases, the *Dirty Packages* pattern should not involve refactoring at all, which is typical when the tested units are some kind of more general entities such as utility functions.

The traceability recovery and refactoring options for the two specific *Other* patterns are similar to that of *Busy Package* and *Dirty Packages*. In the case of *Extractable Feature*, a new package is created from the corresponding $C$ cluster, and this functionality is removed from the participating $P$ cluster, which means that coverage based traceability is to be used. Similarly to *Dirty Packages*, *Dirty Subfeature* also has two options: merging packages into larger ones or introducing mock objects with the corresponding traceability options. Finally, the two additional cases listed for the *Other* category have similar solutions to the previous ones.

## 6 Conclusions

### 6.1 Discussion

Previous works have already demonstrated that fully automatic test-to-code traceability recovery is challenging, if not impossible in a general case (Rompaey and Demeyer (2009); Gaelli et al (2005); Kanstrén (2008)). There are several fundamental approaches proposed for this task: based on, among others, static code analysis, call-graphs, dynamic dependency analysis, name analysis, change history and even questionnaire based approaches (see Section 2 for details). But there seems to be an agreement among researchers that only some kind of a combined or semi-automatic method can provide accurate enough information.

Following this direction, we developed our semi-automatic recovery approach, whose initial description was given by Balogh et al (2016), and which was extended in this work. Our goal was to limit the manual part to the decisions which are context dependent and require some level of knowledge about the system design and test writing practices, hence are hard to automate. In particular, we start with two relatively straightforward automatic approaches, one based on static physical code structure and the other on dynamic behavior of test cases in terms of code coverage. Both can be seen as objective descriptions of the relationship of the unit tests and code units, but from different viewpoints. Our approach to use clustering and thus form mutually traceable groups of elements (instead of atomic traceability information) makes the method more robust because minor inconsistencies will not influence the overall results. Also, the manual phase will be more feasible because a smaller number of elements need to be investigated.

After completing the automatic analysis phase of the recovery process, we aid the user of the method by automatically computing the differences between these results and organizing the differences according to a well-defined set of patterns. These patterns are extensions of those presented by Balogh et al (2016), and here we introduced the NDD to help the detection of these patterns in the CSG. Experience shows from our own empirical study that this information can greatly help in deciding on the final traceability links. As we detail in Section 5.4, we were able to provide some general guidelines as well on how to interpret the results of the automatic analysis.

Our empirical study also demonstrated that the proposed method is able to detect actual issues in the examined systems in relation to the two investigated unit testing principles, isolation and separation. This means that, as a side effect of the traceability recovery effort, such findings can be treated as structural bad smells and may be potential refactoring tasks for these projects.

Although we implemented the approach to handle Java systems employing the JUnit automation framework for unit testing, the method does not depend on a particular language, framework or, in fact, unit testing granularity level. In different settings, other code entities might be considered as a unit (a module, a service, etc.), but this does not influence the applicability of the general concepts of our approach.

Similarly, the two automatic traceability recovery algorithms, the package based and coverage based clustering, could easily be replaced with other analysis algorithms. The comparison and the pattern detection framework could still be used to aid a possibly different kind of manual phase and final decision making.

Finally, the analysis framework, including the two clustering methods, the CSG and pattern detection using the NDD vectors, proved to be scalable on these non-trivial systems. Hence, we believe it would be appropriate for the analysis of bigger systems as well.

### 6.2 Threats to validity

This work involves some threats to validity. First, we selected the subjects 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 than for unit tests. Also, in practice the granularity and size of a unit might differ from what is expected (a Java method). Generally, it is very hard to ascertain automatically if a test is not intended to be a unit test or if the intended granularity is different, so we verified each identified pattern instance manually for these properties as well. However, in actual usage scenarios, this information will probably be known in advance.

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

Finally, the generalizations we made as part of RQ3 might not be directly applicable to other systems, domains or technologies as they were based on the investigated subjects and our judgments only, and no external validation or replication has been performed.

### 6.3 Future work

This work has lots of potential for extension and refinement. Our most concrete discrepancy patterns, *Busy Package* and *Dirty Packages* located relatively few instances, but in the case of the *Other* variants there were more hits. We will further refine this category, possibly by the relaxation of the CSG edge weights, and defining other pattern variants, to increase the number of automatically detectable patterns.

Our framework including the two clustering methods, the CSG and the NDD vectors, can be extended to detect traceability links for other types of testing. In particular, we started to develop a pattern detection decision model for integration tests. In this case, it is expected that tests span multiple modules and connect them via test execution (contrary to a unit test).

The other planned improvements are more of a technical nature, for instance, introduction of a threshold for CSG inclusion weights (as discussed in Section 4). We also started to work on methods for more automatic decision making about the suitable traceability and refactorings options, for instance, based on the CSG edge weights or other features extracted from the production and test code.

## References

Anquetil N, Lethbridge T (1998) Extracting concepts from file names: a new file clustering criterion. In: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, pp 84–93

Balogh G, Gergely T, Beszédes Á, Gyimóthy T (2016) Are my unit tests in the right package? In: Proceedings of 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'16), pp 137–146

Beck K (ed) (2002) Test Driven Development: By Example. Addison-Wesley Professional

Bertolino A (2007) Software testing research: Achievements, challenges, dreams. In: 2007 Future of Software Engineering, IEEE Computer Society, pp 85–103

Black R, van Veenendaal E, Graham D (2012) Foundations of Software Testing: ISTQB Certification. Cengage Learning

Blondel VD, Guillaume JL, Lambiotte R, Lefebvre E (2008) Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment 2008(10):P1000

Bóta A, Krész M, Pluhár A (2011) Dynamic communities and their detection. Acta Cybernetica 20(1):35–52

Bouillon P, Krinke J, Meyer N, Steimann F (2007) Ezunit: A framework for associating failed unit tests with potential programming errors. Agile Processes in Software Engineering and Extreme Programming pp 101–104

Breugelmans M, Van Rompaey B (2008) Testq: Exploring structural and maintenance characteristics of unit test suites. In: WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques

Bruntink M, Van Deursen A (2004) Predicting class testability using object-oriented metrics. In: Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on, IEEE, pp 136–145

De Lucia A, Fasano F, Oliveto R (2008) Traceability management for impact analysis. In: Frontiers of Software Maintenance, 2008. FoSM 2008., IEEE, pp 21–30

Demeyer S, Ducasse S, Nierstrasz O (2002) Object-oriented reengineering patterns. Elsevier

Deursen Av, Moonen L, Bergh Avd, Kok G (2002) Refactoring test code. In: Succi G, Marchesi M, Wells D, Williams L (eds) Extreme Programming Perspectives, Addison-Wesley, pp 141–152

Feathers M (2004) Working effectively with legacy code. Prentice Hall Professional

Fortunato S (2010) Community detection in graphs. Physics reports 486(3):75–174

Gaelli M, Lanza M, Nierstrasz O (2005) Towards a taxonomy of SUnit tests. In: Proceedings of 13th International Smalltalk Conference (ISC'05)

Gaelli M, Wampfler R, Nierstrasz O (2007) Composing tests from examples. Journal of Object Technology 6(9):71–86

Hamill P (2004) Unit Test Frameworks: Tools for High-Quality Software Development. O'Reilly Media, Inc.

Hamilton J, Danicic S (2012) Dependence communities in source code. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, pp 579–582

Horváth F, Vancsics B, Vidács L, Beszédes Á, Tengeri D, Gergely T, Gyimóthy T (2015) Test suite evaluation using code coverage based metrics. In: Proceed-

ings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15), pp 46–60, also appears in CEUR Workshop Proceedings, Vol-1525, urn:nbn:de:0074-1525-1

Kanstrén T (2008) Towards a deeper understanding of test coverage. Journal of Software: Evolution and Process 20(1):59–76

Meszaros G (2007) xUnit test patterns: Refactoring test code. Pearson Education

Mitchell BS, Mancoridis S (2006) On the automatic modularization of software systems using the bunch tool. IEEE Transactions on Software Engineering 32(3):193–208

Müller HA, Orgun MA, Tilley SR, Uhl JS (1993) A reverse-engineering approach to subsystem structure identification. Journal of Software Maintenance: Research and Practice 5(4):181–204

Myers GJ, Sandler C, Badgett T (2011) The art of software testing. John Wiley & Sons

Palla G, Barabási AL, Vicsek T (2007) Quantifying social group evolution. Nature 446(7136):664

Pietrzak B, Walter B (2006) Leveraging code smell detection with inter-smell relations. In: International Conference on Extreme Programming and Agile Processes in Software Engineering, Springer, pp 75–84

Qusef A, Oliveto R, De Lucia A (2010) Recovering traceability links between unit tests and classes under test: An improved method. In: 2010 IEEE International Conference on Software Maintenance, IEEE, pp 1–10

Qusef A, Bavota G, Oliveto R, De Lucia A, Binkley D (2014) Recovering test-to-code traceability using slicing and textual analysis. Journal of Systems and Software 88:147–168

Rompaey BV, Demeyer S (2009) Establishing traceability links between unit test cases and units under test. In: Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on, pp 209–218

Schwanke RW (1991) An intelligent tool for re-engineering software modularity. In: Software Engineering, 1991. Proceedings., 13th International Conference on, IEEE, pp 83–92

Spanoudakis G, Zisman A (2005) Software traceability: a roadmap. Handbook of Software Engineering and Knowledge Engineering 3:395–428

Šubelj L, Bajec M (2011) Community structure of complex software systems: Analysis and applications. Physica A: Statistical Mechanics and its Applications 390(16):2968–2975

Tengeri D, Beszédes Á, Havas D, Gyimóthy T (2014) 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), pp 47–52

Tengeri D, Beszédes Á, Gergely T, Vidács L, Havas D, Gyimóthy T (2015) Beyond code coverage - an approach for test suite assessment and improvement. In: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15); 10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART'15), pp 1–7

Tengeri D, Horváth F, Beszédes Á, Gergely T, Gyimóthy T (2016) Negative effects of bytecode instrumentation on Java source code coverage. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and

Reengineering (SANER 2016), pp 225–235
Van Rompaey B, Demeyer S (2008) Exploring the composition of unit test suites. In: Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on, IEEE, pp 11–20
Vidács L, Horváth F, Tengeri D, Beszédes Á (2016) Assessing the test suite of a large system based on code coverage, efficiency and uniqueness. In: Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, the First International Workshop on Validating Software Tests (VST'16), pp 13–16
Wagner S, Wagner D (2007) Comparing clusterings: an overview. Universität Karlsruhe, Fakultät für Informatik Karlsruhe