Are My Unit Tests in the Right Package?

Gergő Balogh
University of Szeged, Hungary
Software Engineering Department
geryxyz@inf.u-szeged.hu

Tamás Gergely
University of Szeged, Hungary
Software Engineering Department
gertom@inf.u-szeged.hu

Árpád Beszédes
University of Szeged, Hungary
Software Engineering Department
beszedes@inf.u-szeged.hu

Tibor Gyimóthy
University of Szeged, Hungary
Software Engineering Department
gyimothy@inf.u-szeged.hu

***Abstract*—The software development industry has adopted written and de facto standards for creating effective and maintainable unit tests. Unfortunately, like any other source code artifact, they are often written without conforming to these guidelines, or they may evolve into such a state. In this work, we address a specific type of issues related to unit tests. We seek to automatically uncover violations of two fundamental rules: 1) unit tests should exercise only the unit they were designed for, and 2) they should follow a clear packaging convention. Our approach is to use code coverage to investigate the dynamic behaviour of the tests with respect to the code elements of the program, and use this information to identify highly correlated groups of tests and code elements (using community detection algorithm). This grouping is then compared to the trivial grouping determined by package structure, and any discrepancies found are treated as "bad smells." We report on our related measurements on a set of large open source systems with notable unit test suites, and provide guidelines through examples for refactoring the problematic tests.**

***Index Terms*—Code coverage, unit testing, clusterization, package hierarchy, community detection, test smells and refactoring**

I. Introduction

Source level testing is an integral part of most software quality assurance approaches, and there are various types of it including static testing, as well as white-box dynamic testing techniques. In the present work, we deal with the latter; more precisely, with unit level testing. The goal of unit testing is to test a single, standalone unit of the software on itself, *i.e.* isolated from the influence of other components. It does not check either for problems related to the intercommunication with other components, or more higher levels of functionalities. The importance of unit testing is increasing with the popularity of agile development methodologies [1].

Unit tests are often implemented as integral parts of the source of the system under test, written in the language of the system, and usually with the help of specialized frameworks like JUnit [2] for Java. Consequently, these tests might be the subject of source code analysis, just as the system code itself. Source code analysis may then be used for various purposes including test code quality assessment, test comprehension, refactoring, re-documentation, and others.

To develop and evolve high quality unit tests in a long term, is not an easy task. Some researchers have addressed this problem; to join this quest, in this work we approach unit test quality from a different angle. There are two fundamental properties of unit level tests that make them different from higher level test, and that are emphasized in most testing textbooks and unit test writing guidelines (see, *e.g.* [3]). The first one is that a unit test should not test anything outside the unit it is testing. The second expectation is that the test code should be properly organized into the physical source code structure, that is, following the structure of the source code it tests. The goal of our research is to algorithmize the verification of these properties in existing systems and associated test cases. With it, we can then locate problematic parts (*bad smells*) in the test code and eventually propose suitable refactorings to increase this aspect of the tests' quality.

But how can these properties be measured? How can we decide whether a unit test is in the right package, or whether it tests only that part of code it is intended to test? Our approach is to use *detailed code coverage information*, that is, to investigate the dynamic behaviour of the tests with respect to the code elements of the program. It means that we collect runtime information for each test case and record individually what code elements (statements or methods, for instance) each test case executed. This is then stored in a *coverage matrix*, a binary matrix with test cases in its rows, code elements in the columns and 1s in the cells if the given element is reached when executing the given test case. Such detailed coverage information may reveal whether a unit test that physically belongs to a component covers only the code of that component or does it exercise other units too.

Effectively, we want to compare two *clusterings* of the test cases and code elements: one "as implemented" and the other "as behaving." The *as implemented* classification is simply the physical structure of the program and test code, organized into language packages. The other classification is derived from the coverage matrix by applying an algorithm to detect such tightly connected groups of tests and code based on their dynamic relationship of code coverage. Typically, there does not exist a real classification of tests and code without overlap, so for this purpose we selected to apply a heuristic approach proven in other domains to find highly coupled elements, that is *community detection* [4], [5]. It will output sets of nodes (mixed tests and code) which are tightly bound together.

In this work, we present our approach to automatically compare these two clusterings by using suitable measures, and pinpoint to the discrepancies between them. These discrepancies can be thought of as "bad smells", so we also elaborate on the possible refactorings to bring the *as intended* and *as behaving* structures closer together. As it turned out, it is not simple to determine specific parts of the tests that should be refactored, and how they should be modified.

The concepts above have been empirically investigated on a set of real size open source Java programs with significant test suites. To summarize, we provide the following contributions:

1) We applied community detection algorithm on a code coverage matrix to detect sets of closely related unit tests and code elements.
2) We performed measurements on real open source systems, and compared the identified clusterings with the trivial clustering based on physical code structure.
3) We categorized the discrepancies between the two into typical cases, quantified their amount in the subject programs, and provided guidelines how these can be used as actual bad smells and associated refactorings to improve the structures of existing tests.

The rest of the paper is organized as follows. In Section II, we elaborate on the background of this research and provide a motivating example. Section III explains the clustering methods, while Section IV presents our related measurements. Section V discusses the patterns which we found in the differences of the clusterings, with proposed actual bad smells and refactorings. Finally, Section VI deals with related work, before we conclude in Section VII.

II. Background

A. Best practices of unit test writing

Unit testing is a low level testing activity having a close relation to the source code of the system under test. During this test, we search for defects in and verify correct functioning of software components (modules, programs, objects, classes, etc.) which are separately testable [6]. There are many guidelines how to write and organize unit tests (*e.g.* [3], [7], [8], [9]), but most of them start by emphasizing two basic test design concepts: "unit tests should be isolated" and "test only one code unit at a time" [3]. In addition, unit testing frameworks – such as JUnit, which is part of the *de facto* unit test family of frameworks – have naming and packaging conventions on how unit tests should be organized with respect to their intended goal (unit to test) [2]. Also, since different build systems and development environments suggested similar conventions, these eventually became best practices adopted by practitioners.

One of these conventions is about how tests are placed into logical or physical modules (such as packages in the case of Java and folders on the file system). Most environments logically group test and program code together, while physically separating them from each other. This can mean, for instance, that program code and the associated tests are put in the same logical package or namespace, while they are in separate folders on the file system. In addition, naming conventions are often proposed. For instance, the name of a test class is usually formed from the tested class name by adding a "Test" suffix (or prefix) to it, and similarly, the name of a test method usually start with the "test" prefix and if it is designed to test a single method instead of a more general behaviour then it continues with the name of the tested method.

For the purposes of the present research, we thus assume that a well-designed unit test has two important properties:

1) A unit test should exercise the unit and only the unit it was designed for. Execution of code in other units on which the tested one is dependent should be eliminated using *stubs* and *mocks*.
2) Unit tests should follow a clear naming and packaging convention, which reflects both the purpose of the test and the structure of the tested system, providing clear traceability between the test cases and the tested units.

If the guidelines and conventions are not fully followed, the interpretation of test results, comprehension of the testing artifacts and their relation to the code, and the maintenance of the tests in general will be compromised. In many systems, the size and complexity of the unit tests is comparable to that of the software itself [10], so this must be taken seriously. As it is the case with software in general, violations of these conventions are often not the result of poor test writing skills of the programmers or testers but of the natural evolution of the system [11], [12]. For instance, it is sometimes the case that existing units are extended, modified or their physical location is changed, while the associated tests are not immediately updated to reflect the changes [10].

B. Motivating example

In order to fully understand what kinds of issues we seek to find with our approach, consider the example in Figure 1. It shows a client-server software with its tests, having two components: a client consisting of the *HttpClient* and *FtpClient* units and a server consisting of *HttpServer* and *FtpServer*. The components are represented by the *client* and *server* packages (also denoted by purple frames in the figure).

Following property #2 from above (naming and packaging), the four test classes in the example are named *TestHttpClient*, *TestFtpClient*, *TestHttpServer* and *TestFtpServer*, and each of them is placed in the package where the class (unit) it is intended to test is located. Now, consider property #1 (test one unit only) and the client component, for instance. According to this rule, tests of this component should only exercise methods in the *client* package. The test case implemented in *testSendRequest()* will invoke the method *sendRequest()* of class *HttpClient*, which activates method *processRequest()* of class *HttpServer*. However, since the outcome of the client unit test depends on the behaviour of the server unit, this violates the rule. Similar behaviour can be observed for *testSendFile()*.

If similar situations accumulate in a system's test suite, the relationships of the tests and code as visible statically from the packaging and naming might be different to what the actual behaviour of the tests would show. To eliminate the discrepancies between the two kinds of groupings, the developers and testers have two basic options. They might eliminate the existing relations that connect two units, or reorganize the units to follow the existing relations.

In our example, testers might use mocks or stubs (included in the *client* package as part of the tests) instead of the real server objects when the clients are tested. In this case the

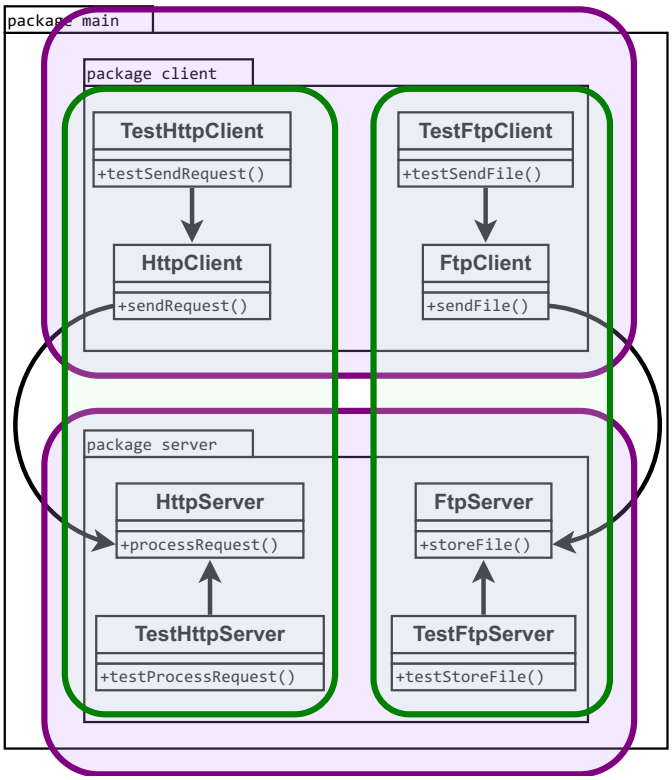

Fig. 1. A running example

relation between the *send** and *process** methods are eliminated, and property #1 is not violated anymore. The second possibility to correct the discrepancies is the reorganization of the packages. The testers and developers might decide to use *http* and *ftp* packages instead of *client* and *server* packages. This way the previously violating relations will become in-unit relations, which no more violates property #1. Note, that the decision depends on several circumstances, and requires a deep investigation of the affected elements and their relations.

The issue with the above mentioned and similar situations is that their detection is difficult, especially if it is to be done manually given the complexity of systems. Static analysis might help but we would face difficulties to properly detect connections between the units. Hence, we rely on lightweight dynamic analysis, namely the coverage information between the tests and the code elements, to find out the connections between them. By comparing the obtained connections to the relationship of the test and code in the packaging structure, we will be able to detect suspicious discrepancies, and eventually offer refactoring solutions to them.

III. Clustering Tests and Code

In the previous section, two important properties of well-designed unit tests have been put forth: their executions are restricted to the tested unit and they are properly named and structured. To check whether this holds for a particular test suite, the manual option would be to verify what pieces of code each test case exercises (directly of indirectly) and compare this to the physical location of the test. This is, however, unfeasible for real-word test suites.

To automate this task, we employ two clustering algorithms that can group together test and code items. The first one (Section III-A) is based on code coverage, and captures dynamic relations between the test suite and the system under test. This is then compared to the other, trivial clustering (Section III-B), that works from static information and captures the structural properties of the tests and program code.

A. Community based clustering

In order to determine the clustering of the tests and code based on the dynamic behaviour of the test suite, we will apply *community detection* [4], [5] on the detailed code coverage information. Detailed code coverage in this case is that, for each test case, we record individually what code elements (methods, in our case) it executed. This forms a binary matrix (called *coverage matrix*), with test cases in its rows and methods in the columns. A value of 1 in a matrix cell indicates that the method is executed 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. An example coverage matrix is shown in Figure 2, which corresponds to our running example from Section II. The first row, for example, shows that *HttpClient.sendRequest()* and *HttpServer.processRequest()* were called during the execution of *TestHttpClient.testSendRequest()*.

	HttpClient.sendRequest()	HttpServer.processRequest()	FtpClient.sendFile()	FtpServer.storeFile()
TestHttpClient.testSendRequest()	1	1	0	0
TestHttpServer.testProcessRequest()	0	1	0	0
TestFtpClient.testSendFile()	0	0	1	1
TestFtpServer.testStoreFile()	0	0	0	1

Fig. 2. A sample coverage matrix

The concept of clustering based on dynamic behaviour used in this work can be illustrated by investigating different regions in the coverage matrix. Groups of tests and methods that form “dense regions” in the matrix may be grouped together indicating that there is a tight correspondence between them from dynamic point of view. These regions contain more 1 values in the cells, while outside of them in their rows and columns the 0 values are more common. In our example, such regions are marked with rectangles. The property of the members of such groups (or clusters) is that their test cases more probably cover their methods than other methods, and that their methods are more probably covered by their test cases than by other test cases.

Clustering coverage matrices to find such groups is not simple, and generally there is not a single “perfect” clustering

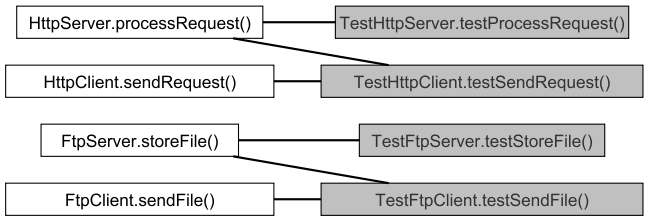

Fig. 3. Sample coverage graph

possible. It would mean that members of a cluster show complete coverage while there is no coverage reaching out of any cluster. This is unrealistic for real software, so in practice the clusters vary in size, number and coverage density. There might be different approaches to detect these clusters, but they are based on some kind of a heuristic that tries to maximize the coverage within a cluster and minimize it outside.

Our choice for cluster identification was to use *community detection* [5]. This set of algorithms is 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 [13], [14]. Community structures are detected based on statistical information about the number of edges between sets of graph nodes. So, in the next step 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). A method and a test node in this graph are connected with a single unweighted and undirected edge if and only if the method was covered during the execution of that particular test, *i.e.* there is a 1 in the corresponding matrix cell. 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. Figure 3 shows the coverage graph of our example.

The actual algorithm we used for community detection is the Louvain Modularity method [4] (also used by Hamilton and Danicic [14]), 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 iteratively, 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 of modularity. 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 (including loop edges created from intra-cluster edges). Then the algorithm restarts with this new graph, and repeats the two phases until no change would result in positive gain.

When this algorithm is applied on the coverage graph of our example program, it starts with 8 clusters, each containing a single method. Then, at the end of the first phase of the first iteration two clusters will be formed: the algorithm puts the *TestHttpServer.testSendRequest()*, *HttpServer.sendRequest()*, *HttpClient.processRequest()*, and *TestHttpClient.testProcessRequest()* into one cluster, and *TestFtpServer.testSendFile()*, *FtpServer.sendFile()*, *FtpClient.storeFile()*, and *TestFtpClient.testStoreFile()* into another cluster. These are the clusters we identified and marked with green frames in Figure 1. In the second phase of the first iteration, the algorithm constructs a new graph consisting of two nodes (corresponding to the two clusters) and only two loop edges for these nodes. In the next iteration of the algorithm no change is made (*i.e.* both nodes remain isolated), so the algorithm ends up with these two clusters.

B. 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. However, it might happen that the developers did not follow unit testing guidelines or the system evolved such that due to package reorganization the package structure does not reflect the actual role of the tests. Hence, it is interesting to ask how package based clustering will relate to community based clustering.

Our package based clustering simply means that we assign the fully qualified name of the innermost containing package to each test and method, and treat tests and methods belonging to the same package members of the same cluster. There are two important things to note here: first, 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). Second, names of the test and code elements are not considered either; the classification if a particular piece of code is a unit test or regular code is determined by the rules of JUnit (such as the special annotations), our unit testing framework.[1]

In the case of our running example, the method *sendRequest()* and the test *testSendRequest()* are both located in the *client* package, which is in the *main* package, hence both of these items are put into the *main/client* cluster. The respective methods of the other two classes of the same

[1]As mentioned, this is our assumption of how developers intend to organize their unit tests, which might not hold for some projects.

package will belong to this cluster as well, and a similar structuring will be obtained also in the *main/server* package. (These are indicated by purple rectangles in Figure 1.)

This clustering eventually reflects the intention of programmers and testers of this system because it clearly reflects what should be tested with what tests, and as such they are easy to locate within the code structure. However, as we elaborated at the end of Section II, this contradicts to what the behaviour based grouping shows (see the green rectangles in Figure 1), because the tests *testSendRequest()* and *testSendFile()* of *main.client* will cover respective methods from *main.server*. In other words, there is a discrepancy between the two clusterings, which we will investigate in more detail in the following sections.

IV. Clustering Measurements

Our first set of measurements, presented in this section, deals with the overall level of agreement between the package based and the community based clusterings of the tests and code elements. The classification of the concrete differences will be performed in Section V.

A. Experiment setup

In our experiments, we used a set of open source Java programs that are non-trivial both in their size and in terms of their test suites. We searched for small and medium sized projects on GitHub [15] whose test suites are based on the JUnit framework [2], and preferring those that had been used in some previous experiments. The selected eight subject programs and their properties are listed in Table I. The build processes of the systems were extended to include code coverage measurement, for which purpose we used the Clover coverage measurement tool [16]. This tool is based on source-code instrumentation, which gives more precise information about source code entities [17]. We decided to use method-level coverage analysis, *i.e.* the atomic code element of a coverage matrix in its columns is a Java method.

The lists of methods and test cases were then processed to extract package based clustering information as described in Section III-B. Next, we performed the community detection method described in Section III-A on the detailed coverage data.

TABLE I
Subject programs and their basic properties

Program	Version	LOC	Methods	Tests
checkstyle	*6.11.1*	114K	2 655	1 487
commons-lang	*#00fafe77*	69K	2 796	3 326
commons-math	*#2aa4681c*	177K	7 167	5 081
joda-time	*2.9*	85K	3 898	4 174
mapdb	*1.0.8*	53K	1 608	1 774
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

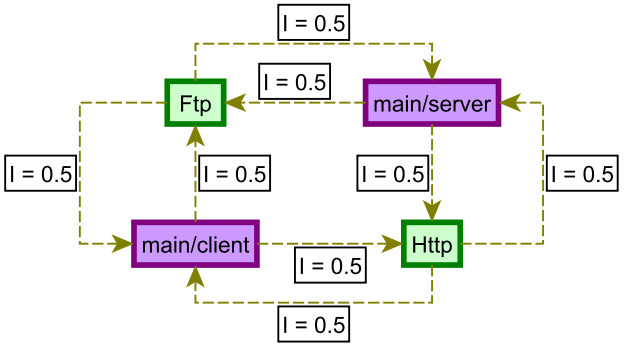

Fig. 4. A sample cluster similarity graph showing Inclusion measures

B. Similarity metrics

At this point, we have two sets of clusters on the tests and code items: one based on the physical structure of the code and tests (according to property #2), and another one based on the coverage data of the tests showing their behaviour (according to property #1). In an ideal case these are the same: there is a one-to-one assignment between the clusters of the two clusterings. However, we expected that there is a disagreement between the two, so we first looked for methods to quantify the disagreement on an overall level.

In the related literature, there were several methods proposed to compare two clusterings and express their similarity. They are usually based on various similarity measures; Wagner and Wagner [18] provide a list of these. Our choice was to use *Jaccard index*, so we computed this measure on the clusterings derived for our subject programs. *Jaccard index* ranges from zero to one, and it is based on counting pairs of objects that are "classified" in the same way in both clusterings.

The resulting similarity values are shown in the second column of Table II. While the obtained values might indicate the level of agreement on a very high level, the index computation algorithms may prefer or disregard different properties of the clusterings (*e.g.* the number of nodes put in a separate cluster by both clusterings), and some of them are more sensitive to small changes in certain input parameters than others [18]. Due to these reasons and the fact that it shows only an overall picture without pinpointing to actual issues, we abandoned these high level measurements from the rest of the experiments.

Therefore, we decided to compare the resulting clusters on a more detailed level: each cluster of one clustering was compared to each cluster of the other. This way, we got pairwise relations for all possible combinations of the cluster correspondence, which can be represented as a weighted complete bipartite directed graph (referred to as *cluster similarity graph – CSG* in the following). Nodes of this graph are the clusters of each of the clusterings in two disjoint sets, and the edges contain labels corresponding to a similarity measure defined for pairs of clusters from the two clusterings.

We will use a non-symmetric similarity measure, which will enable a more detailed kind of analysis. The *Inclusion measure* for two clusters C_1 and C_2 expresses in what degree C_1 is included in C_2. Value 0 means no inclusion, while 1 means that C_2 fully includes C_1. This measure is computed as:

$$I(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1|}.$$

Inclusion measure forms a CSG, where an edge from the package based cluster node P to a community based cluster node C is weighted by $I(P, C)$ and the reverse edge from C to P has $I(C, P)$ as its weight. The edges start from the included cluster and point to the container.

We illustrate the CSG of our example in Figure 4. In this, for clusters $P_{main/client}$, $P_{main/server}$, C_{HTTP}, and C_{FTP}, we can measure 8 inclusion measure values (*i.e.* $I(P_{main/client}, C_{\text{HTTP}})$, $I(P_{main/client}, C_{\text{FTP}})$, $I(P_{main/server}, C_{\text{HTTP}})$, $I(P_{main/server}, C_{\text{FTP}})$, and their reverses $I(C_{\text{HTTP}}, P_{main/client})$, $I(C_{\text{FTP}}, P_{main/client})$, $I(C_{\text{HTTP}}, P_{main/server})$, $I(C_{\text{FTP}}, P_{main/server})$). As each cluster contains 4 items of which exactly 2 can be found in one of the clusters of the other type, the inclusion measure for any allowed cluster combination is $2/4$, indicating that all edges of the graph have a weight of 0.5.

C. Cluster similarity graph statistics

Table II shows statistics about the cluster nodes of the CSG-s computed for our subject programs. The *clst* columns show the number of corresponding cluster nodes in the CSG. The following columns show statistics on the number of nodes these clusters contain from the underlying coverage graphs, *i.e.* methods and test cases. Note, that the total number of nodes (as the product of the cluster number and their average size) may differ from those presented in Table I. The reason for this is that the clustering algorithm cannot handle isolated nodes (*i.e.* the methods that are not covered by any test cases), so these items are excluded from the clustering.

Different programs show quite different properties. We can find programs where much more community based clusters were detected than package based clusters (*e.g. commons-lang*), but the opposite situation can also be found (*e.g. orientdb*). The minimum, maximum, median, and average values, both the absolute and the percentile ones, also vary in a wide range for different programs. By comparing these values of the package based clusters, we can observe the trend that there are more smaller clusters and fewer larger ones. This phenomenon is more emphasized among the community based clusters: except for *mapdb* (where the size of the community based clusters seems to be more balanced), there are a lot of really small clusters (with a couple of elements only) and there are only very few large ones.

In general, community based cluster nodes contain less elements than the package based ones (see the median percentiles, for instance). These together draws up that some really small parts of the software are tested in isolation as units, while a large part of the code together with its tests forms a monolythic block in which no single units can be detected. However, where there are more community based clusters than package based ones, our concept of a unit might be too coarse grained (*i.e.* we could check the program with class-based clustering).

Table III shows statistics on the relations between the clusters in the CSG. The *all rel.* column shows the total number of relations in the graphs, while the following one counts only those that have non-zero weights (*i.e.* where the Inclusion measure value between two clusters is not 0). In the rest of the statistics, only these non-zero relations are used.

The *density* column shows the ratio of non-zero relations and all relations. This value might indicate an overall picture on how the clusters are in general related to each other. However, since it does not reflect the distribution of values within the non-zero range it is not always good as a general indicator. A lot of zero-value and a few high value relations are typical of well-structured systems while a lot of non-zero but low values indicate worse structuredness. The medians and averages of inclusion values can give additional insight besides density values (shown in the last two columns of the table). Low average and median inclusion values together with high density are signs of worse structuredness (for example, *joda-time* and *mapdb*), while the same with low density (such as *orientdb*) probably indicate better structured programs.

Further, high average and median inclusion might also indicate good structure, especially combined with low density. This is the case with *netty* and *oryx*. An interesting addition is that systems *netty*, *orientdb* and *oryx* are the only ones having separately buildable sub-modules, meaning they are probably better structured by design, and these systems scored the best according to our metrics as well.

TABLE III
STATISTICS ON INCLUSION MEASURE OF OUR SUBJECT PROGRAMS

	all rel.	n. z.	density	min.	max.	med.	avg.
checkstyle	2160	278	12.9%	0.001	1.000	0.082	0.248
commons-lang	7150	616	8.6%	0.000	1.000	0.147	0.468
commons-math	5822	514	8.8%	0.001	1.000	0.038	0.218
joda-time	378	132	34.9%	0.000	1.000	0.082	0.227
mapdb	56	30	53.6%	0.001	1.000	0.224	0.367
netty	3060	150	4.9%	0.001	1.000	0.493	0.527
orientdb	10920	608	5.6%	0.001	1.000	0.077	0.283
oryx	2106	106	5.0%	0.016	1.000	0.751	0.623

These global statistical data could be analyzed further, however, similar to the global Jaccard index presented in Table II, they give only an overall picture of the structure of the unit tests. To see the details and infer information that can directly lead to the enhancement of the tests, we need a deeper analysis of the CSG-s.

V. SMELLS AND REFACTORINGS

In this section, we summarize our observations we made after carefully investigating the details of the two clusterings computed for our subject systems. Due to the large number of clusters and their relations that were determined by the automatic methods (see data in Tables II and III), we could not investigate all of them. Instead, we systematically examined

TABLE II
STATISTICS ON CLUSTER SIMILARITY GRAPHS OF OUR SUBJECT PROGRAMS. PERCENTAGES IN PARENTHESES ARE RELATIVE TO THE TOTAL NUMBER OF NODES IN THE COVERAGE GRAPHS.

	Jaccard index	package based clusters					community based clusters				
		clst.	min.	max.	median	average	clst.	min.	max.	median	average
checkstyle	0.12	24	5 (0.1)%	608 (15.5)%	129 (3.3%)	164.0 (4.2%)	45	2 (0.1%)	924 (23.5%)	2 (0.1%)	87.4 (2.2%)
commons-lang	0.21	13	4 (0.1)%	2102 (35.5)%	216 (3.7%)	454.8 (7.7%)	275	2 (0.0%)	1033 (17.5%)	3 (0.1%)	21.5 (0.4%)
commons-math	0.15	71	2 (0.0)%	1850 (16.6)%	78 (0.7%)	156.9 (1.4%)	41	2 (0.0%)	1621 (14.6%)	7 (0.1%)	271.7 (2.4%)
joda-time	0.16	9	1 (0.0)%	4531 (59.1)%	259 (3.4%)	852.4 (11.1%)	21	2 (0.0%)	1937 (25.2%)	2 (0.0%)	365.3 (4.8%)
mapdb	0.22	4	6 (0.2)%	2699 (95.1)%	66 (2.3%)	709.3 (25.0%)	7	3 (0.1%)	854 (30.1%)	427 (15.1%)	405.3 (14.3%)
netty	0.30	45	4 (0.1)%	4218 (54.9)%	56 (0.7%)	170.7 (2.2%)	34	2 (0.0%)	1806 (23.5%)	35 (0.4%)	225.9 (2.9%)
orientdb	0.08	130	1 (0.0)%	465 (8.8)%	17 (0.3%)	40.5 (0.8%)	42	2 (0.0%)	1645 (31.3%)	7 (0.1%)	125.2 (2.4%)
oryx	0.39	27	2 (0.3)%	117 (17.9)%	14 (2.1%)	24.2 (3.7%)	39	2 (0.3%)	66 (10.1%)	11 (1.7%)	16.8 (2.6%)

each community cluster and its surroundings using a script and based on the Inclusion values related to these clusters, looked for various patterns in the graphs. We verified the correctness of the algorithm and its implementation by randomly selecting and checking some of the reported cases.

In order to ease our manual investigations, we visualized the cluster similarity graphs with the following rules (see, for instance, Figure 6):

1) Each cluster is represented by a rectangle, whose color encodes its type: purple is package based, green is community based.
2) We reduce the number of connections between the nodes by displaying only those that have the first three highest values among the outgoing edges for each node.
3) The similarity measures are displayed as labels on the edges.
4) Instead of using some arbitrary identifiers for the clusters, we generated a string that could be used as a mnemonic for the content of the cluster. This string is composed using a simple string processing of the names of the methods and tests in the clusters with the following steps.
   a) We remove the largest common substring of all names of the coverage graph. It usually contains the name of the system and the root package, for example: *com.cloudera.oryx.* for *oryx*.
   b) We take the longest common substring of all names of methods and tests in that particular cluster and use it as a summary of the cluster, for example: `code "*kmeans/common/" ... tested by test "*kmeans/common/"`
   c) We collect the 10 most common words of the rest of the names of methods and tests and use these to show other relevant members of the cluster. For example: `also: Centers math3 apache commons Ljava Validity Statistics Cluster Real linear etc.`
   d) We include the number of contained nodes in the cluster in the last line.

We observed different patterns in the CSG-s, but they could be grouped roughly into two categories: "normal structures" (discussed in Section V-A) and "anomalies" (Section V-B). Normal scenarios either show one-to-one correspondence between the two clusterings or the discrepancies are such that can be justified by engineering decisions, and that actually show valid designs. Anomalies, on the other hand, show some kind of *bad smells* in the design, which could be eventually fixed by suitable refactorings.

In the present research we do not aim to provide a systematic way of automatically detecting these bad smells and suggest refactorings. Instead, we aim to categorize the different scenarios with our initial ideas for their refactoring, and quantify them in our subjects (discussed in Section V-C).

A. Normal scenarios

These scenarios reflect situations when the design of the test cases and their location seems normal and modification is not necessary. In the ideal case a single unit observable along the test suite structure is also observable from the actual behaviour of the tests. The methods and the tests are exactly the same in both clusters, hence they are alternative manifestations of the same entity. The inclusion measures between the two clusters are 1 in either direction in these cases. These patterns will be referred to as *Ideal* in the following. A possible relaxation of this scenario could be to use a threshold value less than 1 for the measures above which the clusters are treated identical.

Figure 5 shows an Ideal case from *oryx*: the two clusters are formed from the 4 tests and 18 methods of the *commons.servercomp* package.

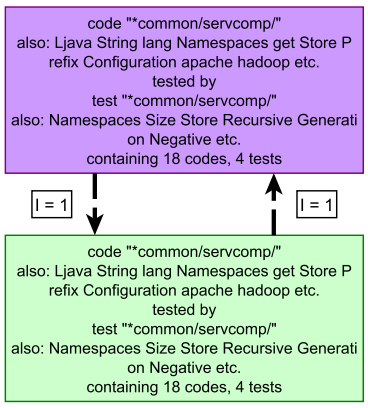

Fig. 5. Perfect correspondence between the two clusterings (*Ideal* pattern). Example is from *oryx*.

Figure 6 shows a scenario that we also treat as a normal one. Namely, we have one *package based* cluster that consists of more *community based* clusters. This reflects the situation when the definition of the unit the testers use is not atomic (*e.g.* working with classes as units, while methods could also be separately tested). In this case the tests of the single unit are partitioned, and different test cases are testing (covering) different parts of the unit, and the partitions together correspond to the cluster as a whole. This pattern will be called in the following the *Clear-cut* scenario. The property of Clear-cuts with a package based cluster P and its associated community clusters C_i is that P is the union of the C_i clusters. This means that each C_i cluster is fully included in the P cluster ($I(C_i, P) = 1$) and the union of C_i clusters fully includes P (*i.e.* $\sum_i I(P, C_i) = 1$). This could also be modified so that a threshold is used for the measures instead of 1.

The Clear-cut example in Figure 6 is from the *oryx* program. The package based cluster is defined for the *common/stats* package. However, community detection revealed that there are (at least) three distinct clusters corresponding to the behaviour of the tests: two for the concepts of double and integer calculations, respectively, and some common tests for the JVM environment. These three community clusters are fully included in the package based cluster (shown by the edge labels $I = 1.0$). Note, that the sum of the inclusion values in the opposite direction is not 1 as would be expected, because (as mentioned earlier) we show only related clusters with the three highest values.

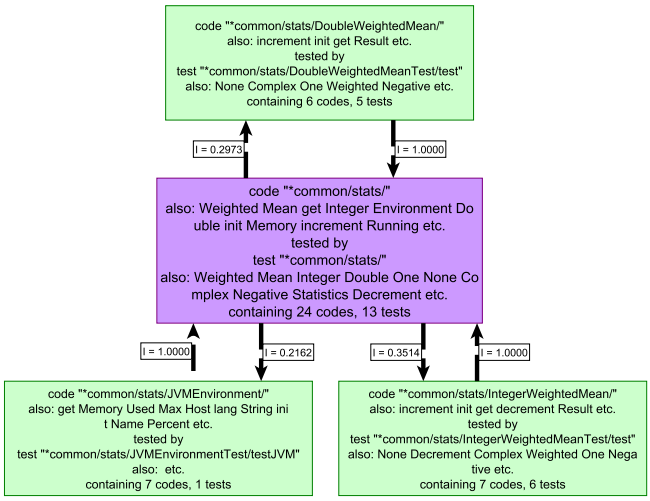

Fig. 6. Example for *Clear-cut* pattern from *oryx*.

B. Anomalies

We treat anomaly to be present in the clustering comparison when package based clusters do not clearly correspond to a set of associated community clusters as in the case of the Clear-cut pattern. In other words, anomaly is when a package based cluster P does not fully include a community based cluster C, that is $I(C, P) < 1$. In this case the remaining elements of C (those not included in P) will be included in other package based clusters P_i, resulting in inclusion values $I(C, P_i) < 1$. We will refer to this pattern as the *Anomaly* pattern. In this case, a single community contributes to more package based clusters, meaning that tests are bypassing unit borders, seemingly violating property #1. Similarly to the normal scenarios, this one could also be relaxed by lowering the threshold for inclusion below 1.

Note, that here we do not examine how packages are included in communities, *i.e.* the $I(P_i, C)$ weights of edges starting from the package clusters and ending in the community. Such an examination might help to distinguish sub-types of anomalies, which is not the topic of the present research.

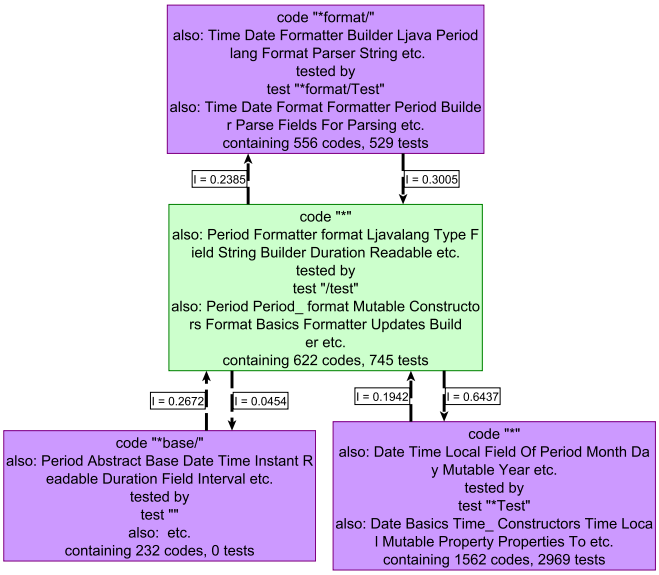

Fig. 7. The *Anomaly* pattern shows that parts of more than one package result in a community cluster. Example is from *joda-time*.

An anomaly pattern can be seen in Figure 7, which is from *joda-time*. At least three package based clusters include the same community cluster with inclusion values 0.64, 0.24, and 0.04. Although the 0.04 inclusion value and lower values (not represented in the figure) are small enough to be treated as insignificant, the two highest values are worth to be considered. The community cluster seems to represent a quite large amount of functionality (see the number of its elements). This, by itself, might be an indicator that the contained tests and code should be separated. We can observe, further, that there is one common concept, "*period*", which is in all of the related package clusters as well. However, the two most important package based clusters are seemingly implementing different concepts: "*date*", "*time*", etc. So, the question arises if the community should be separated by limiting the functionality of its tests, or the package based clusters should maybe integrated together to better reflect their behaviour?

In general, the first solution can be performed through rewriting the tests. The testers should investigate what relations in the code cause the involvement of other units, and then

they should try to eliminate these dependencies by mocking or stubbing the cross-unit entities. However, sometimes the implementation does not allow this kind of replacement (*e.g.* when the code directly creates some cross-unit instances). In such cases only the second solution may be potentially applicable: the developers and testers should reorganize the units, creating a single one that better reflects the relationships and the dynamic behaviour of the code and the tests. Furthermore, these two solutions can be applied together. For example, in a first step the community cluster could be broken up into several ones by mocking the tests, and then a reorganization of the units could form the packages that reflect the behaviour of the system.

C. Distribution of patterns in the subjects

We automatically searched for the presence of the patterns in our subject systems using the script mentioned at the beginning of this section. It examines each package or community cluster and their surroundings on the cluster similarity graph, and using the inclusion measure determines one of the three cases. The corresponding statistics are shown in Table IV. The table is divided into three regions, representing the three cases elaborated in the previous sections. For the *Ideal* and *Clear-cut* patterns we base our measurements on the number of package based clusters that are involved in such scenarios, while in case of the *Anomaly* pattern the number of affected community clusters will be used. The first column in each region shows the actual count of the given pattern occurrence found in the corresponding subject, while *possible occurrences* denotes the total number of clusters of the corresponding type. The last column in each region shows the ratio of these elements.

TABLE IV
COUNT OF CLUSTER COMPARISON PATTERNS IN THE SUBJECT SYSTEMS

smell	Ideal			Clear-cut			Anomaly		
	occurrences	possible occ.	occurrences %	occurrences	possible occ.	occurrences %	occurrences	possible occ.	occurrences %
checkstyle	0	24	0%	1	24	4%	13	45	29%
commons-lang	0	13	0%	0	13	0%	15	275	6%
commons-math	0	71	0%	0	71	0%	15	41	37%
joda-time	0	9	0%	0	9	0%	8	21	38%
mapdb	0	4	0%	0	4	0%	4	7	57%
netty	4	45	9%	1	45	2%	12	34	35%
orientdb	2	130	2%	0	130	0%	13	42	31%
oryx	8	27	30%	4	27	15%	8	39	21%

We can observe that, although there are ideal and clear cut scenarios for some programs, most of the clusters – furthermore, in four systems (*commons-lang*, *commons-math*, *joda-time*, and *netty*) all of them – are involved in anomalies. Our plan was not to investigate each pattern occurrence in detail in this phase of the research; instead, we manually checked the systems with most detected anomalies. This indicates the need of reorganization of units or re-implementation of the unit tests, especially for programs like *joda-time* and *mapdb*. For example this manual inspection revealed that *joda-time* uses a very small number of packages to group the tests and methods. The intention of the developers is encoded into pre- and postfixes of the class names, for example *TestDateMidnight_Basics, TestDateMidnight_Constructors* and *TestDateMidnight_Properties.* This information could be expressed by moving the relevant items into different packages. However, to ensure causal relationship between these properties and the high number of anomalies, further investigation is required.

VI. RELATED WORK

We organize this section along the lines of the following topics, which our work is based on: test smell analysis, software clustering detection, and community structures.

There is a large body of work in the area of code smells, and researchers only recently started to apply similar concepts to check software tests and test code for quality issues. For tests that are implemented as executable code, Van Deursen *et al.* introduced the concept of *test smells*, which indicate poorly designed test code [19], and listed 11 test code smells with suggested refactorings. Our work best relates to their *Indirect Testing* smell. Meszaros broadened the scope of the concept, by describing test smells that act on a behaviour or a project level, next to code-level smells [9]. There are also some follow-up researches that use these ideas in practice. For example, Breugelmans and Van Rompaey present TestQ which allows developers to visually explore test suites and quantify test smelliness. They also demonstrate its use on test suites for both C++ and Java systems [7]. Van Rompaey and Demeyer also proposed a visualization of unit tests that focused on the relations between test code and production code to help software engineers understand the structure and quality of the test suite of large systems [8]. Our work significantly differs from these approaches because we are not concerned in code-oriented issues in the tests but in their dynamic behaviour and relationship to their physical placement.

We proposed a related approach to group related test and code elements together, but this was based on manual classification done by the testers and developers [20]. 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 [21].

There are various approaches and techniques for automatically grouping different items of software systems together based on their structural or behavioural properties. Mitchell and Mancoridis [13] examine the Bunch clustering system which, unlike other software clustering tools, uses search techniques to perform clustering. Schwanke's ARCH tool [22] determined clusters using coupling and cohesion measurements. The Rigi system [23], by Müller *et al.* 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 [24], who used common patterns in file names as a clustering criterion.

The concept of community structure arises from the analysis of social networks in sociology. Community structures can be identified in many other real world graphs and have applications in biology, economics and engineering, among others. Recently, efficient community detection algorithms have been developed which can cope with very large graphs with millions of nodes and potentially billions of edges [4]. Application of these algorithms to software engineering problems is emerging. Hamilton and Danicic [14] 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 [25] 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.

VII. Conclusion

This work addressed the quality of unit test suites from a novel angle. Our approach is to compare the physical organization of tests and tested code in the package hierarchy to what can be observed from dynamic behaviour of the tests. The application of community detection algorithms for the latter is a viable approach, and we believe that this kind of analysis of unit tests may reveal knowledge about them not investigated earlier. Our results indicate that for realistic systems, there are a quite lot of discrepancies between the package based and community based structures. But it does not necessarily mean that each of these need to be fixed in the first place by some kind of refactoring of test code. Furthermore, it is not generally possible to decide if there is a problem with the placement of test cases in the package structure or with the way test cases invoke elements of the tested code.

The latter will be our main direction for future work. More precisely, we plan to investigate in what situations do violations of clustering indicate the need for refactoring, and when should we suggest moving test cases to different packages or the internal working of the test case should be modified instead. This way, we would obtain a real bad smell and refactoring catalog for this particular kind of test code quality issues. Our plans for the continuation also include more detailed analysis of the anomaly patterns, to define more specific cases. Finally, we would like to compare the approach to what manual classification can reveal about the system and the tests, and how these can be combined with other (possibly static) test code measurement aspects.

Acknowledgment

This work was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

[1] L. Crispin and J. Gregory, Eds., *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009.

[2] "JUnit Java unit test framework homepage," http://junit.org/, last visited: 2016-05-27.

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

[4] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[5] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3, pp. 75–174, 2010.

[6] R. Black, E. van Veenendaal, and D. Graham, *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2012.

[7] M. Breugelmans and B. Van Rompaey, "Testq: Exploring structural and maintenance characteristics of unit test suites," in *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[8] B. Van Rompaey and S. Demeyer, "Exploring the composition of unit test suites," in *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*. IEEE, 2008, pp. 11–20.

[9] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[10] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. v. Deursen, "Mining software repositories to study co-evolution of production & test code," in *1st International Conference on Software Testing, Verification, and Validation*, April 2008, pp. 220–229.

[11] T. Mens, "Introduction and roadmap: History and challenges of software evolution," in *Software evolution*. Springer, 2008, pp. 1–11.

[12] T. M. Pigoski, *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

[13] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[14] J. Hamilton and S. Danicic, "Dependence communities in source code," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 579–582.

[15] "GitHub homepage," https://github.com/, last visited: 2016-05-27.

[16] "Clover Java and groovy code coverage tool homepage," https://www.atlassian.com/software/clover/overview, last visited: 2016-05-27.

[17] D. Tengeri, F. Horváth, Á. Beszédes, T. Gergely, and T. Gyimóthy, "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)*, Mar. 2016, pp. 225–235.

[18] S. Wagner and D. Wagner, *Comparing clusterings: an overview*. Universität Karlsruhe, Fakultät für Informatik Karlsruhe, 2007.

[19] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. Kok, "Refactoring test code," in *Extreme Programming Perspectives*, G. Succi, M. Marchesi, D. Wells, and L. Williams, Eds. Addison-Wesley, 2002, pp. 141–152.

[20] D. Tengeri, Á. Beszédes, T. Gergely, L. Vidács, D. Havas, and T. Gyimóthy, "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)*, Apr. 2015, pp. 1–7.

[21] L. Vidács, F. Horváth, D. Tengeri, and Á. Beszédes, "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)*, Mar. 2016, pp. 13–16.

[22] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *Software Engineering, 1991. Proceedings., 13th International Conference on*. IEEE, 1991, pp. 83–92.

[23] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[24] N. Anquetil and T. Lethbridge, "Extracting concepts from file names: a new file clustering criterion," in *Proceedings of the 20th international conference on Software engineering*. IEEE Computer Society, 1998, pp. 84–93.

[25] L. Šubelj and M. Bajec, "Community structure of complex software systems: Analysis and applications," *Physica A: Statistical Mechanics and its Applications*, vol. 390, no. 16, pp. 2968–2975, 2011.