

Comparison of Test Groups Based on Behavior and Package Hierarchy

Béla Vancsics(✉)

Department of Software Engineering, University of Szeged,
Dugonics tér 13, Szeged 6720, Hungary
vancsics@inf.u-szeged.hu

Abstract. Data collected from the software and the development process is usually part of some kind of underlying structure, hence they could be organized into groups or clusters. This is true of the test cases. There are several concepts for creating groups, one approach is that the tests are grouped by their location, another approach is that they are categorized based on their behavior. For unit tests, ideally, these two approaches would give the same result. However this is not true in reality!

In our empirical study the difference between the two concepts (on Java systems) was examined, also the behaviour groups based on coverage and the location-groups based on package hierarchy were analyzed.

Groups containing tests were examined as well. In addition the difference between package-based groups and behavior-based groups based on coverage matrix was explored. To quantify the results, empirical experiments were performed and the differences were identified with the metrics used in the literature.

Keywords: Code coverage · Clustering · Software evolution

1 Introduction

Knowing about groups of strongly related elements has many benefits in software engineering. These groups reflect the underlying relations in the software such as structure, behavior or other aspects of the designer's intentions. These are often called *clusters* in software engineering research. Clusters can not only be interpreted for software components but in terms of teams, error reports, documents, informal messages, version controlling information, and many others as well. Cluster identification, or simply *clustering*, means finding these groups by automatic means. It has different applications in maintenance, program comprehension, redocumentation, traceability and so on [13,16,18].

Clustering software engineering artifacts has a large literature, and there are quite effective algorithms and tools available for various purposes [2,12,14].

Well-formed groups can help to understand the program, create appropriate unit tests [1], detect bad smell [21], feature detection [9] increase maintenance as well. It is difficult to measure a cluster algorithms' success, because there is no perfect clustering and "standards" making is very costly. There are many clustering

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 388–402, 2019.
https://doi.org/10.1007/978-3-030-24305-0_29

approaches and plenty of algorithms [3]. Out of these, choosing the most suitable for us is a non-trivial question. Clustering software engineering artifacts has large literature, and there are quite effective algorithms available [2,14].

Dynamic information was compared to the static information extracted from the location of the source code. The difference between the two approaches points to places where static and dynamic groups present discrepancies, what is to say the intention of the designer (shown by the location) and the behavior of the coverage shows the divergences. These differences make it difficult to understand the program, impair maintenance and thus increase development (and maintenance) costs.

In our research, we investigated how the groups created on the basis of static and dynamic test information are similar. If there is a high degree of similarity between the two clusters, it means that the location of the test in the structure represents the "function" of the test. In this study several clustering algorithms were used, differences between the results were examined. This way we got a more comprehensive picture of the differences between the two types of structure.

2 Overview and Goals

In this paper, we present results of an empirical study whose goal is to compare different clustering approaches on test cases based on behavior and package hierarchy. Several similarities and approaches were examined to determine the dynamic groups. To quantify the results the clustering metrics recognized by the literature were used.

This research is a step towards a practically usable method, which would give the place of anomalies between structural and behavioral groups and could help the refactoring.

2.1 Method

Figure 1 shows a general overview of the experiment. There are four main parts of the process:

1. Construction of Package-Based Clustering:
   The package clusters are treated as a good approach which indicates the function of the test cases and thus helps developers' and testers' understanding. The appropriate package structure improves maintainability and traceability.
2. Construction of Coverage-Based Clustering:
   On the basis of the coverage information, we examine how similar the test cases are to each other. Elements with similar coverage will have a high chance of belonging to the same group because they are likely to test the same function.
3. Comparison of the two clusterings:
   The similarity between the groups obtained in the previous two steps is quantified. Thus, the difference between them becomes measurable, so the efficiency of the algorithms (in this aspect) can be evaluated quantitatively.

4. Evaluation:
   We examine the results of each algorithm according to different aspects, and investigate whether there is a method that stands out from the others (i.e. the result is most similar to the Package-Based Clustering).

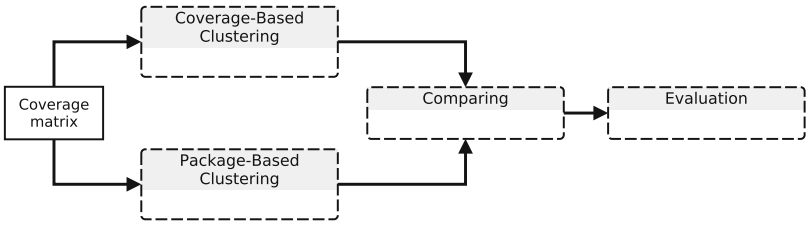

Fig. 1. Overview of the method: it contains four steps of Package-Based Clustering, construction of Coverage-Based Clustering, comparison of two algorithms and evaluation

2.2 Subjects

We conducted the experiment by applying the clustering methods on 7 small to medium sized open source Java systems. All of the subjects' test suites are written in JUnit, the unit testing framework often used for Java systems. Table 1 lists the subject systems along with some of their basic properties.

The granularity we used in our experiments was JUnit test cases, i. e., the basic entities used for clustering are the test cases. Clover tool[1] was used for coverage measurement [20]. It is an open source tool managed by Atlassian (until 2017) which uses source code instrumentation technique. It gives more precise information about source code entities than tools based on bytecode instrumentation (e.g Cobertura and JaCoCo).

Table 1. Subject programs and their basic properties

Program	Version	LOC	Methods	Tests
Checkstyle	6.11.1	114K	2655	1487
Commons-lang	#00fafe77	69K	2796	3326
Commons-math	#2aa4681c	177K	7167	5081
Joda-Time	2.9	85K	3898	4174
Netty	4.0.29	140K	8230	3982
Orientdb	2.0.10	229K	13118	925
Oryx	1.1.0	31K	1562	208

[1] https://atlassian.com/software/clover.

2.3 Research Questions

We may describe the goals of our empirical study more precisely as follows.

RQ1 How close are the "behavior-based" Coverage-Based Clusterings to the Package-Based Clusterings?
How similar are the resulting groups of different algorithms to the package structure?

RQ2 Are there Coverage-Based Clustering configurations that are definitely better (or worse) than others?
Which is the projection-algorithm "pair" that gives a much better (or worse) similarity?

RQ3 How great are the deviation of results within a program?
We look at the size difference between the results of the algorithms.

3 Package-Based Clustering

It is shown how the Package-Based Clustering is created that is based on the physical code structure (Fig. 2). Since we are dealing with Java systems in the present experiment and, in Java, the basic physical entity that represents the system's hierarchical architecture is a *package*, we are using this term in this part.

Our aim with test clustering is to detect groups of tests that are connected together by the "intention" of the developer or tester. This means that their placement within a hierarchical package structure of the system is a natural classification according to their intended role. It helps other developers and testers understand the connection between tests and their subjects. Packages often include subpackages as well.

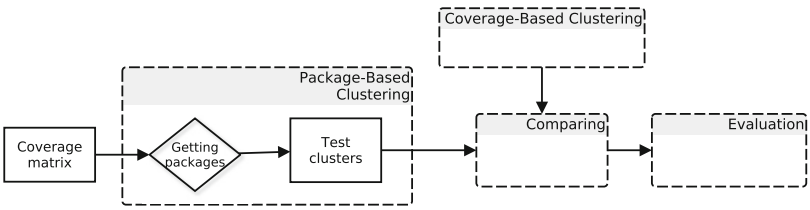

Fig. 2. Package-Based Clustering

Our Package-Based Clustering simply means that we assign the fully qualified name of the innermost containing package to each test, and treat them 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, tests' names 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.

Some properties of the resulting clusters is shown in Table 2.

Table 2. Properties of package-based test clusters

Program	Number of test cluster	Size of the smallest group	Size of the largest group
Checkstyle	23	5	178
Commons-lang	12	17	1156
Commons-math	61	1	924
Joda-Time	7	1	2969
Netty	35	1	3109
Orientdb	53	1	185
Oryx	23	1	45

4 Coverage-Based Clustering

Coverage-Based Clustering takes into account only the dynamic code coverage relationship between the test cases and the code elements. Using code coverage in software testing is based on simple concepts [8,19]. The coverage matrix is a bit-matrix representing which test cases cover which code elements.

$$A_{cov} = \begin{pmatrix} x_{1,1} & \dots & x_{1,l} \\ \vdots & \ddots & \vdots \\ x_{k,1} & \dots & x_{k,l} \end{pmatrix} \quad \begin{array}{l} T = \{t_1, t_2, ...t_k : test\ cases\} \\ M = \{m_1, m_2, ...m_l : methods\} \\ x_{a,b} \in \{0,1\} : \ a \in T, \ b \in M \\ x_{i,j} = 1 \text{ if method } j \text{ is executed by test } i \end{array}$$

The basic idea behind this approach is that we want to identify groups in the coverage data, which are more closely resembled to the in-group elements than the out-group elements. These regions may indicate that there is a tight correspondence between the associated test cases from a dynamic point of view. For finding similarity, we use "projection-functions" that indicate how similar the two elements are to each other. The basis for clustering will be the similarity matrix thus obtained (Fig. 3).

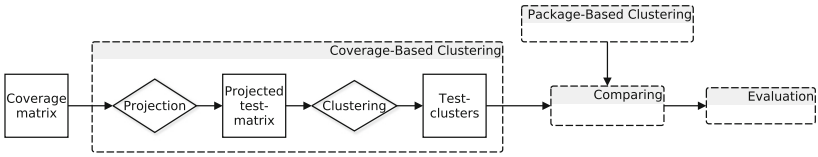

Fig. 3. Coverage-Based Clustering

4.1 Projection

From coverage matrix was produced a "projected" matrices ($A \rightarrow P$) by a (dis)similarity function. Projected matrix captures the similarities among pairs of tests. The larger this value, the more similar the coverage and the more likely they are responsible for the same feature.

The projected matrix specifies how two elements are similar to each other. The higher the value, the more similar the two elements are (based on coverage). These projected matrices are the basics of clustering.

For example: if $t_a \in T$ and $t_b \in T$ and $P_{t_a,t_b} = 0.98$, then tests will most likely cost the same cluster because their coverage is very similar.

Well-known similarity (or distance) functions have been selected to describe the relationship between two elements. These functions give the similarity of two (u and v) boolean (test coverage) vectors, where $c_{i,j}$ is the number of occurrences of $u[k] = i$ and $v[k] = j$ for $k < n$ (n = length of vector) and $i, j \in \{0, 1\}$.

Table 3. Similarity functions

$$\underline{\mathbf{D}}\text{ice(u, v)} = \frac{c_{1,0} + c_{0,1}}{2 * c_{1,1} + c_{0,1} + c_{1,0}} \qquad \underline{\mathbf{H}}\text{amming(u, v)} = \frac{c_{1,0} + c_{0,1}}{n}$$

$$\underline{\mathbf{J}}\text{accard(u, v)} = \frac{c_{1,0} + c_{0,1}}{c_{1,1} + c_{0,1} + c_{1,0}} \qquad \underline{\mathbf{K}}\text{ulsinski(u, v)} = \frac{c_{1,0} + c_{0,1} + c_{1,1} + n}{c_{0,1} + c_{1,0} + n}$$

$$\underline{\mathbf{R}}\text{ogers-}\underline{\mathbf{T}}\text{animoto(u, v)} = \frac{R}{c_{1,1} + c_{0,0} + R} \qquad \underline{\mathbf{R}}\text{ussell-}\underline{\mathbf{R}}\text{ao(u, v)} = \frac{n - c_{1,1}}{n}$$

$$\underline{\mathbf{S}}\text{okal-}\underline{\mathbf{M}}\text{ichener(u, v)} = \frac{R}{c_{1,0} + c_{0,1} + R} \qquad \underline{\mathbf{S}}\text{okal-}\underline{\mathbf{S}}\text{neath(u, v)} = \frac{R}{c_{1,1} + R}$$

$$\underline{\mathbf{Y}}\text{ule(u, v)} = \frac{R}{c_{1,1} * c_{0,0} + \frac{R}{2}}$$

$$\text{,where } R = 2 * (c_{1,0} + c_{0,1})$$

4.2 Clustering Algorithms

Many different clustering algorithms exist for different problem domains [7]. The aim of clustering is to group the items based on some relation (e. g. similarity or distance) between them.

Classification is similar to clustering nevertheless there is an important difference: classification is supervised while clustering is an unsupervised grouping. The resulting clusters can be overlapping (an item can belong to several groups) or disjoint (an element can only belong to one group). Clustering algorithms can have three important properties (scale invariance, consistency and richness), however, maximum of two properties can be realized for a certain clustering algorithm [10].

The algorithms can be divided into two groups based on whether the number of clusters is a parameter (e. g.: K-means) or is determined by the method (e. g.: Affinity Propagation). In the first case the predefined cluster number has a significant influence on the result, but it is also a drawback because, generally, it is impossible to determine the cluster number in advance. It should also be expected to be determined automatically.

In our research, we only used algorithms that do not need to give the number of clusters.

Affinity Propagation (AP). This algorithm is based on a "passing-message-between-datapoints" strategy. It requires as input a matrix and gives as the output a set of disjointed clusters. Real-valued messages are exchanged between data points until a high-quality set of exemplars and corresponding clusters gradually emerge. The method finds "exemplars" (or data point) that is representative enough of itself and some other data points [6].

The algorithm proceeds by two message passing steps. They will update two matrices.

- The first matrix has values $r(i, j)$ that quantify how well-suited x_j is to serve as the exemplar for x_i, relative to other candidate exemplars for x_i ($x_{i,*}, x_{j,*} \in A'$).
- The second matrix contains values $a(i, j)$ that represent how "proper" it would be for x_i to pick x_j as its exemplar, taking into account other points' preference for x_j as an exemplar.

The algorithm then performs the updates iteratively:

- $r(i,j) \leftarrow s(i,j) - max_{j' \neq j}\{a(i,j') + s(i,j')\}$
- $a(i,j) \leftarrow min\left(0, r(j,j) + \sum_{i' \notin \{i,j\}} max(0, r(i',j))\right), (i \neq k)$
  $a(j,j) \leftarrow \sum_{i' \neq j} max(0, r(i',j))$
- $s\,(i,j) = -\left\|x_i - x_j\right\|^2$

The iterations are repeated until there is change in the cluster.

Birch Clustering (BC). Balanced Iterative Reducing and Clustering using Hierarchies [11,23](BIRCH) is based on hierarchical clustering approach. It is an efficient and scalable clustering method, based on an in-memory data structure called CF-tree (clustering feature-tree), which serves as an in-memory summary of the data distribution.

The data is represented by the CF nodes. Each node represents a cluster in the cluster hierarchy, intermediate nodes are superclusters and the leaf nodes are the actual clusters. The algorithm needs two parameters: (i) threshold - it is a limit between sample and the existing subclusters (ii) branching factor - it is a limit for the number of subclusters.

In the first step, the method builds a clustering feature (CF) vector that summarizes information for each cluster.

$$CF(C) = (|C|, S_1, S_2), \text{ where } |C| = \text{number of instance in C cluster}$$

$$S_1 = \sum_{X_i \in N} X_i \ , \ S_2 = \sum_{X_i \in N} |X_i|^2$$

In the second step, the algorithm discovers all leaf in the CF-tree to rebuild a new, smaller tree. It removes the outliers and grouping congested subclusters. In the third step, an agglomerative hierarchical clustering is used to cluster all leafs.

DBSCAN Clustering (DB). Density-Based Spatial Clustering of Applications with Noise [5](DBSCAN) is one of the most common and most cited clustering algorithm.

It is a density-based clustering method which uses the concept of density reachability and density connectivity: given a group of points in some space, it groups together points that are closely connected together.

Let D be a set of data points:

- ϵ-neighborhood of a point: $N_\epsilon(p) = \{q \in D | \text{distance}(p, q) \leq \epsilon\}$
- *Core Point*: $N_\epsilon(p) \geq N_{min}$
- Directly Density-Reachable (DDR): if $p \in N_\epsilon(q)$ and $N_\epsilon(q) \geq N_{min}$ then p is DDR from q
- Density-Reachable (DR): if there is a sequence of points $p = p_1, p_2, ..., p_n, p_1 = q, p_n = q$ such as p_{i+1} is DDR from p_i
- Density-Connected (DC): if there is r such as both, p and q are density-reachable from r then p is DC to q

The clusters are the set of density-connected points that are maximal with respect to density-reachability. Let C be a cluster: $\forall p, q$: if $p \in C$ and q is density-reachable from p then $q \in C$.

DBSCAN has several variations and extensions [7].

Mean Shift Clustering (MS). Mean shift is the center-based clustering algorithm [4]. It is a simple iterative procedure that shifts each data point to the average of data points in its neighborhood [4].

The clustering aims to discover "blobs" in a smooth density of samples. It iteratively shifts each point in the data set until it is the top of its nearest kernel density estimation (KDE) surface peak. The algorithm copies the original data set and it freezes the original points, then copied points are shifted against the frozen data. The point moves closer to the nearest KDE surface peak by iterations.

Gaussian kernel function $K(x_i - x) = e^{-c\|x_i - x\|^2}$ determines the weight of nearby points for re-estimation of the mean.

The weighted mean of the density in the window determined by kernel function is

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x)x_i}{\sum_{x_i \in N(x)} K(x_i - x)},$$

where $N(x)$ is the neighborhood of initial estimate (x). The algorithm sets $x \leftarrow m(x)$ and the estimation is repeat until $m(x)$ converges.

5 Comparing and Evaluation

The quantification of the results is on three levels (Fig. 4): (1) clustering of the tests (there are only tests in the groups) were examined (2) methods' clustering (the groups contain only methods) and (3) combined (or heterogeneous) clusters. The focus was only on the test clusters.

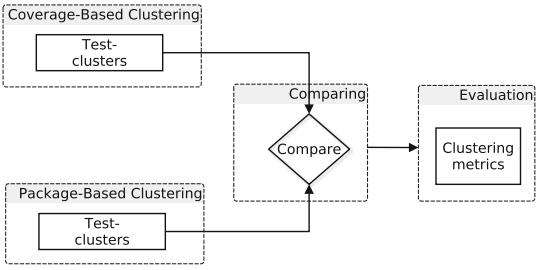

Fig. 4. Overview of comparing and evaluation

Comparing two clusterings C and C means calculating a similarity value $\delta(C, C')$ using some measure δ, which ranges $[0, 1]$, 0 being the least similar and 1 representing identical clustering. Except for the identical clustering, the similarity is difficult to express since two clusterings may differ in many ways: in the cluster numbers, in the content of the clusters, in the amount of relationships within and between clusters, etc. Hence, there might not exist a universal measure for the similarity; each different definition might be suitable from a certain perspective. In the literature, different comparison methods have been

advised [7], and we selected six of them in order to see if they show similar or different results. Each metric is interpreted in the test- , method- and heterogeneous clusters.

Evaluating the performance of a clustering algorithm is not trivial, but there are some approaches that are accepted by the scientific literature. In the research, five popular metrics were used to quantify the differences.

5.1 Adjusted and Normalized Mutual Information

The Mutual Information (MI) of two variables is a measure of the mutual dependence between the two variables. Adjusted Mutual Information [22] (AMI) is an adjustment of the MI score to account for chance. (This is necessary because MI is higher for two clusterings with a larger number of clusters, disregarding whether there is actually more information shared or not.) Normalized Mutual Information [17] (NMI) is a normalization of the MI score to scale the results between 0 and 1.

$$MI(C, C') = \sum_{i=1}^{|C|} \sum_{j=1}^{|C'|} \frac{|C_i \cap C'_j|}{N} \cdot log\left(\frac{N \cdot |C_i \cap C'_j|}{|C_j| \cdot |C'_j|}\right)$$

$$H(C) = -\sum_{i=1}^{|C|} \frac{|C_i|}{N} \cdot log\left(\frac{|C_i|}{N}\right) \ , \ H(C') = -\sum_{j=1}^{|C'|} \frac{|C'_j|}{N} \cdot log\left(\frac{|C'_j|}{N}\right)$$

$$\boxed{AMI(C, C') = \frac{MI - E[MI]}{max(H(C'), H(C')) - E[MI]}} \qquad \boxed{NMI(C, C') = \frac{MI(C, C')}{\sqrt{H(C) \cdot H(C')}}}$$

5.2 Completeness, Homogeneity and V-measure Scores

A clustering result appeases homogeneity if all of its clusters contain only data points which are members of a single class and it satisfies completeness if all the data points that are members of a given class are elements of the same cluster. [15]. V-measure (v) is an entropy-based measure, which is calculated by the harmonic mean of distinct homogeneity (h) and completeness(c) scores [15].

$$a_{ij} = ||a \in N : a \in C_i \text{ and } a \in C'_j|| \text{ ,where N is data points}$$

$$H(C|C') = -\sum_{k=1}^{|C'|} \sum_{c=1}^{|C|} \frac{a_{ck}}{N} log \frac{a_{ck}}{\sum_{c=1}^{|C|} a_{ck}} \ , \ H(C) = -\sum_{c=1}^{|C|} \frac{\sum_{k=1}^{|C'|} a_{ck}}{n} log \frac{\sum_{k=1}^{|C'|} a_{ck}}{n}$$

$$H(C'|C) = -\sum_{c=1}^{|C|} \sum_{k=1}^{|C'|} \frac{a_{ck}}{N} log \frac{a_{ck}}{\sum_{k=1}^{|C'|} a_{ck}} \ , \ H(C') = -\sum_{k=1}^{|C'|} \frac{\sum_{c=1}^{|C|} a_{ck}}{n} log \frac{\sum_{c=1}^{|C|} a_{ck}}{n}$$

$$h = \begin{cases} 1 & \text{if } H(C,C') = 0 \\ 1 - \frac{H(C|C')}{H(C)} & \text{else} \end{cases} \qquad c = \begin{cases} 1 & \text{if } H(C',C) = 0 \\ 1 - \frac{H(C'|C)}{H(C')} & \text{else} \end{cases}$$

$$v = \frac{2 \cdot h \cdot c}{h + c}$$

5.3 Results

Different metrics facilitate quantitative comparison of the projection algorithm pairs. In this study, we investigated which projections and algorithms are the best and the worst results in most cases.

Table 4 shows the average metric values and their standard deviation. In most cases, these values are between 0,2 and 0,5 (but interestingly, AMI value is usually lower). In addition, the *homogenity metrics* have high standard deviation, that is to say, the values obtained are in a broad spectrum. In general, *OrientDB* had the highest average values, so the groups of *OrientDB* tests are the most similar to each other. The smallest group matches was observed in the case of *Joda-Time*.

Table 4. Average values and standard deviations

		v	h	NMI	AMI	c
Checkstyle	AVG	0,3064	0,3000	0,3218	0,1886	0,3857
	STD.DEV	0,1454	0,2361	0,1449	0,0779	0,0628
Commons-math	AVG	0,3502	0,4000	0,3866	0,1866	0,4845
	STD.DEV	0,2031	0,3411	0,1917	0,1189	0,1153
Commons-lang	AVG	0,3611	0,4968	0,4066	0,2061	0,4334
	STD.DEV	0,1147	0,3333	0,1214	0,0868	0,1479
Joda-Time	AVG	0,1446	0,3610	0,1821	0,0767	0,1095
	STD.DEV	0,0728	0,3340	0,1132	0,0308	0,0274
Netty	AVG	0,2210	0,4222	0,2613	0,1159	0,2204
	STD.DEV	0,1100	0,3299	0,1323	0,0763	0,133
OrientDB	AVG	0,3984	0,3972	0,4381	0,2293	0,5949
	STD.DEV	0,2254	0,3215	0,1971	0,1543	0,0586
Oryx	AVG	0,2323	0,3017	0,2604	0,1075	0,3034
	STD.DEV	0,0815	0,1983	0,0759	0,0464	0,1768

Tables 5 and 6 show how many times the algorithms and projections produced the best and worst results. It can be seen that in the case of the three metrics (V-measure, homogeneity and NMI) Birch clustering gave the best results in

most cases, but in the other two cases this method was not good. Overall, in most cases the Birch gave the best results (17), followed by Mean Shift (8), which performed well in complexity metric. Most of the time, the DBSCAN had the worst results (34), followed by the Mean Shift (10). Interestingly, the Birch never gave the worst values.

In the case of projections, the *Yule* was the best, producing the best results 20 times. It gave the highest metric values almost always and only once appeared in the worst configuration. Surprisingly, there were four projections (*Hamming, Rogers-Tanimoto, Russell-Rao and Sokal-Michener*) that never appeared in the best-performing configuration. Of the projections, the *Hamming* gave the worst results for a total of 12 times (*Rogers-Tanimoto and Sokal-Michener*: 8 - 8).

Table 5. Average scores and standard deviation

	Number of best cases				Number of worst cases			
	AP	BC	DB	MS	AP	BC	DB	MS
v	2	5	0	0	0	0	8	1
h	0	6	0	0	0	0	9	1
NMI	0	6	0	1	0	0	7	2
AMI	3	0	2	2	0	0	9	1
c	1	0	1	5	2	0	1	5
SUM	**6**	**17**	**3**	**8**	**2**	**0**	**34**	**10**

Table 6. The best and the worst projections

	Number of best cases						Number of worst cases					
	v	h	NMI	AMI	c	SUM	v	h	NMI	AMI	c	SUM
D	1	0	0	1	0	**2**	0	0	1	0	1	**2**
H	0	0	0	0	0	**0**	3	3	2	3	1	**12**
J	1	0	3	1	1	**6**	1	1	1	1	1	**5**
K	0	0	0	1	0	**1**	1	1	1	1	1	**5**
RT	0	0	0	0	0	**0**	2	2	1	2	1	**8**
RR	0	0	0	0	0	**0**	0	0	0	0	1	**1**
SM	0	0	0	0	0	**0**	2	2	1	2	1	**8**
SS	0	4	0	0	2	**6**	1	1	1	1	0	**4**
Y	5	3	4	4	4	**20**	0	0	0	0	1	**1**

Table 7 shows the best and the worst metrics (per project), the number of clusters and the algorithm-projection pair. It can be observed that the number of clusters vary across a wide range, which may be very different from the number

of packages. The worst results were typically achieved by configurations with very low cluster number (but there is an exception that gave a high number).

The answer for the **RQ1** and **RQ3**: It was found that the programs can be relatively well structured on the basis of behavior, except in the case of *Joda-Time* that has produced poor results (Tables 4 and 7).

Table 7. The best and the worst configurations for projects

		best score			worst score		
		score	$\|C'\|$	algorithm-projection pair	score	$\|C'\|$	algorithm-projection pair
Checkstyle	v	0,6150	639	BC-Y	0,0050	2	DB-H
	h	0,9348	639	BC-Y	0,0025	2	DB-H
	NMI	0,6545	639	BC-Y	0,0311	2	DB-H
	AMI	0,3020	63	DB-J	0,0010	2	DB-H
	c	0,5032	53	DB-Y	0,1733	5	MS-Y
Commons-Math	v	0,6124	2454	BC-J	0,0214	4	DB-RT, DB-SM
	h	0,9919	2835	BC-Y	0,0101	4	DB-RT, DB-SM
	NMI	0,6619	2454	BC-J	0,0671	2	MS-D
	AMI	0,3977	375	AP-D	0,0081	4	DB-RT, DB-SM
	c	0,9176	3	MS-SS	0,1783	2	MS-D
Commons-Lang	v	0,5229	364	AP-Y	0,0154	2	DB-SM, DB-RT
	h	0,9974	1508	BC-SS	0,0079	2	DB-SM, DB-RT
	NMI	0,5718	941	BC-Y	0,0537	2	DB-SM, DB-RT
	AMI	0,3906	93	DB-Y	0,0070	2	DB-SM, DB-RT
	c	0,6975	2	MS-J	0,2492	906	AP-SM, AP-RT
Joda-Time	v	0,2459	368	AP-Y	0,0043	3	DB-H
	h	0,9919	2479	BC-Y	0,0022	3	DB-H
	NMI	0,3655	1753	BC-J	0,0146	3	DB-H
	AMI	0,1279	14	MS-Y	0,0012	3	DB-H
	c	0,1569	14	MS-Y	0,0603	2	MS-RR
Netty	v	0,5215	8	MS-Y	0,0024	2	DB-H
	h	0,9936	1461	BC-SS	0,0012	2	DB-H
	NMI	0,5336	8	MS-Y	0,0263	2	DB-H
	AMI	0,4221	8	MS-Y	0,0009	2	DB-H
	c	0,6616	8	MS-Y	0,0837	2	MS-J
OrientDB	v	0,7065	265	BC-D	0,0386	2	DB-K
	h	0,9592	350	BC-SS	0,0202	2	DB-K
	NMI	0,7297	298	BC-J	0,0940	2	DB-K
	AMI	0,4893	50	AP-K	0,0138	2	DB-K
	c	0,8043	2	MS-SS	0,4383	2	DB-K
Oryx	v	0,7801	79	BC-Y	0,1085	3	DB-SS, MS-J
	h	0,9889	103	BC-SS	0,0592	3	DB-SS, MS-J
	NMI	0,7973	79	BC-Y	0,1953	3	DB-SS, MS-J
	AMI	0,5952	28	AP-Y	0,0343	3	DB-SS, MS-J
	c	0,7149	28	AP-Y	0,4612	10	MS-H

The answer for the **RQ2**: By analyzing Table 7, we can find out how many times the best and the worst values were given by projection-algorithm pairs. Yule projection with Birch (8) and MeanShift (6) gave the best scores in most cases, these were "the best configurations". The Hamming-DBSCAN pair stands out from the worst configurations, and it has the worst metrics 12 times.

All in all, we can find that there are extremely good projection-algorithm pairs whose results are very similar to the structure, but in many cases clusters can vary significantly and this is also apparent in standard deviation. However, by choosing the appropriate methods, the differences can highlight points in the program where transparency and maintainability can be increased by refactoring.

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

References

1. Balogh, G., Gergely, T., Beszédes, A., Gyimóthy, T.: Are my unit tests in the right package? In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 137–146. IEEE (2016)
2. Barman, S., Gope, H.L., Islam, M.M., Hasan, M.M., Salma, U.: Clustering techniques for software engineering. Indonesian J. Electr. Eng. Comput. Sci. **4**(2), 465–472 (2016)
3. Berkhin, P.: A survey of clustering data mining techniques. In: Kogan, J., Nicholas, C., Teboulle, M. (eds.) Grouping Multidimensional Data. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-28349-8_2
4. Comaniciu, D., Meer, P.: Mean shift: a robust approach toward feature space analysis. IEEE Trans. Pattern Anal. Mach. Intell. **24**(5), 603–619 (2002)
5. Ester, M., Kriegel, H.P., Sander, J., Xu, X., et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD, vol. 96, 226–231 (1996)
6. Frey, B.J., Dueck, D.: Clustering by passing messages between data points. Science **315**(5814), 972–976 (2007)
7. Gan, G., Ma, C., Wu, J.: Data Clustering: Theory, Algorithms, and Applications. Society for Industrial and Applied Mathematics, Philadelphia (2007)
8. Horváth, F., et al.: Test suite evaluation using code coverage based metrics. In: Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST2015), pp. 46–60, October 2015
9. Kicsi, A., Vidács, L., Beszédes, A., Kocsis, F., Kovács, I.: Information retrieval based feature analysis for product line adoption in 4gl systems. In: Proceedings of the 17th International Conference on Computational Science and Its Applications - ICCSA 2017, pp. 1–6. IEEE (2017)
10. Kleinberg, J.: An impossibility theorem for clustering. In: NIPS, vol. 15, pp. 463–470 (2002)
11. Lorbeer, B., Kosareva, A., Deva, B., Softić, D., Ruppel, P., Küpper, A.: Variations on the clustering algorithm Birch. Big Data Res. **11**, 44–53 (2017)
12. Lung, C.H., Zaman, M., Nandi, A.: Applications of clustering techniques to software partitioning, recovery and restructuring. J. Syst. Softw. **73**(2), 227–244 (2004)

13. Maletic, J.I., Marcus, A.: Supporting program comprehension using semantic and structural information. In: Proceedings of the 23rd International Conference on Software Engineering, pp. 103–112. IEEE Computer Society (2001)
14. Mitchell, B.: Clustering software systems to identify subsystem structures. Department of Mathematics & Computer Science Drexel University, Philadelphia (2006)
15. Rosenberg, A., Hirschberg, J.: V-measure: A conditional entropy-based external cluster evaluation measure. In: Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL) (2007)
16. Shtern, M., Tzerpos, V.: Clustering methodologies for software engineering. Adv. Softw. Eng. **2012**, 1 (2012)
17. Strehl, A., Ghosh, J.: Cluster ensembles-a knowledge reuse framework for combining multiple partitions. J. Mach. Learn. Res. **3**, 583–617 (2002)
18. Tabacznyj, C.: Abstract clustering for program comprehension. In: Proceedings of the 4th Irish conference on Formal Methods, pp. 93–108. British Computer Society (2000)
19. Tengeri, D., Beszédes, A., Havas, D., Gyimóthy, T.: Toolset and program repository for code coverage-based test suite analysis and manipulation. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 47–52. IEEE (2014)
20. Tengeri, D., Horváth, F., Beszédes, Á., Gergely, T., Gyimóthy, T.: 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, March 2016
21. Van Rompaey, B., Demeyer, S.: Exploring the composition of unit test suites. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops, ASE Workshops, pp. 11–20. IEEE (2008)
22. Vinh, N.X., Epps, J., Bailey, J.: Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. J. Mach. Learn. Res. **11**, 2837–2854 (2010)
23. Zhang, T., Ramakrishnan, R., Livny, M.: Birch: an efficient data clustering method for very large databases. In: ACM Sigmod Record, vol. 25, pp. 103–114. ACM (1996)