МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2015
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2015
Proceedings of the Forty Fourth Spring Conference
of the Union of Bulgarian Mathematicians
SOK "Kamchia", April 2–6, 2015

DISCOVERING CLASSES SUITABLE FOR SPLITTING*

Todor Cholakov, Dimitar Birov

In this paper we propose an algorithm and a tool for discovering classes suitable for splitting into two or more, smaller and more encapsulated classes. We give an approximation of the complexity of the algorithm and we report examples from experiments performed on open source projects. We developed a tool in purpose to experiment with the algorithm. It helps the users to visualize both the general results of the algorithm and the specific code changes that are suggested. We found that it is very useful for industry grade applications.

1. Introduction. The classes that we are going to discuss in this paper are these that can be easily split into smaller and more encapsulated classes. The discussed classes may be either big or small and usually implement two or three (in rare cases more) concepts or logically different tasks. Each of these tasks is well defined in the class – it has its own set of methods and data and is relatively independent of the other tasks. Some authors [1] reference these classes as god-classes, but this is not true in all cases as we will see in the next paragraphs.

This kind of classes need to be refactored at the earliest possible time when they have been discovered, because they make the code maintenance very difficult and code comprehension harder. Also they have a trend towards becoming bigger and bigger. The things go even worse when the tasks within the class share common data or methods making both the recognition of the class and the refactoring process difficult.

The classes that we are discussing here in some cases may be considered as an initial step towards god-classes during the development cycle. As well as they might be a first step towards the development of a new module. There are god-classes that can easily be partitioned into smaller ones, but in most cases the methods in the god-classes are too internally connected and the refactoring is much more complicated.

As we mentioned above, it is not necessary for a class to be large in order to be suitable for splitting into smaller classes. However it is good to define a reasonable minimum under which the algorithm will discard the class. This is necessary because classes that are too small may be in the beginning of their development and the logic that would unite the data and the methods is still to be implemented. Additionally if the algorithm considers too small classes this causes trouble to the software engineer

*2010 Mathematics Subject Classification: 68N19.
Key words: reengineering, refactoring, god-classes, tools.
The authors gratefully acknowledges the financial support by the Bulgarian National Science Fund within project DO 02-102/23.04.2009.

deciding what to refactor, because the small classes are much more than the large ones and the benefit of their splitting is much smaller, because the quality of the code does not improve significantly.

There is a variety of algorithms and approaches regarding class decomposition in research community. Some authors like Bavota et al. [2] look for the semantic similarity between the methods that are supposed to be extracted to a new class. Simon et al. [3] use an algorithm based on cohesion and coupling between the methods to separate the class into clusters. They visualize the output to the user to further decide whether the class is suitable for refactoring or not. Fokaef et al. [1] also use metrics based refactoring, but propose an algorithm suitable for automatic discovery and application of the *extract class* [4] refactoring.

However none of their techniques takes into account either the incoming or the outgoing connections of the class. Using the incoming and outgoing connections of the potential class to be split, improves the accuracy of the algorithm reducing both false positives and false negatives.

For the purpose of describing the algorithm we need to define the concept of call graphs. Call graphs [5] are graphs representing method, function, procedure or constructor invocations within the analyzed program or piece of software. In the rest of the paper we will use the term "method" to designate the three of them. The vertices of such graphs are methods. And there is an edge (oriented arrow) from *method1* to *method2* if *method1* calls directly *method2*.

There are two kinds of call graphs depending on the labeling of the edges:

- **Statistical call graphs**. The edges of these graphs have weights, depending on the number of calls that each edge represents. Between each two nodes in the graph there may be at most two edges – at most one in each direction.
- **Canonical call graphs**. Each edge in these graphs corresponds to a method call. The edges are labeled with the locations where these calls occur.

Call graphs may also vary depending on the level of detail they represent:

- The simplest approach is only direct method calls to denote edges in the graph.
- Adding implicit and explicit constructors increases the level of detail of the graph.
- Additionally field references and assignments may be added as edges in the graph.
- The highest level of detail may be achieved when adding some of the language constructions as special method calls.

Call graphs are useful for analyzing program behavior and structure. They allow the software developer to easily follow the program execution path and eventually to find and fix bugs, which would otherwise be hard to reproduce.

In this paper we use only statistical call graphs. As the weights of the edges are not necessary for understanding the discussed conceptions, they are skipped from the examples in this paper.

In the next section we are going to discuss the characteristics of the classes suitable for splitting and possible criteria for their recognition in each case. In section 3 the proposed algorithm is described into details. In section 4 we examine the complexity of the algorithm. In section 5 we demonstrate some experiments using the described algorithm. In section 6 we discuss the possible approaches toward the visualization of the results of the algorithm.

2. Classes suitable for splitting. Before we describe the algorithm for discovering such classes it is necessary to analyze their structure and characteristics in order to be able to extract those features that would help us to recognize such classes.

The first kind of classes suitable for partitioning are those having high internal connectivity. It is possible to divide such classes into parts based on their own methods and data. Each subclass has its own set of data and its methods mainly use that set (they may use other fields or methods but this happens much more rarely). In the scope of a single subclass the methods have high degree of connectivity, while connections to methods or variables belonging to the other subclasses are an exception (see figure 1 below).

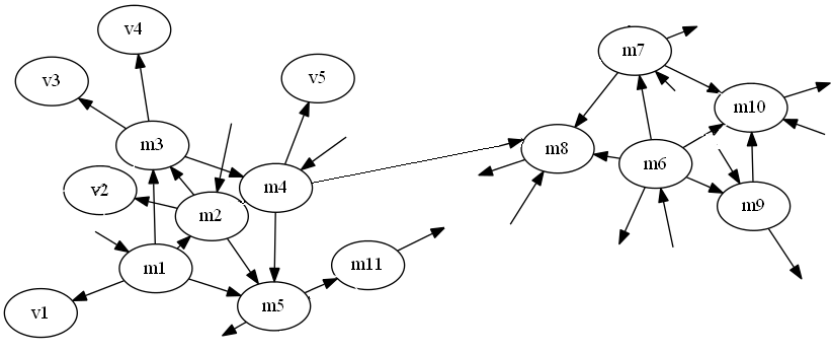

Fig. 1. Class suitable for splitting that has high degree of internal connectivity

Problems regarding recognition may be caused by variables which are shared between most of the methods and by constructors and initializer methods that change most of the fields in the class. This corresponds to libraries and dispatchers when talking about module recognition. In our case however problems with recognition may also be caused by the relations to other classes. Problems arise when all the methods of the analyzed class are used by the same group of external classes. In this case if we consider the incoming connections as a criterion for the clusterization, this complicates the recognition process. On the other hand outbound connections in most cases they may be useful, because the different functionalities in the class are supposed to use different external libraries and different parts of the product. However in some cases they might cause additional problems if the same external libraries and classes are used by all subclasses of the class.

The second kind of classes suitable for partitioning are those that have low internal connectivity. These classes often function as a proxies or wrappers. Their methods call directly external methods, or refer some of the fields of the class. In both cases we have low complexity of each method and low degree of connectivity between the different methods in the class.

The main difficulty regarding these classes comes from the fact that if we remove the external connections, the clustering algorithm will return too many clusters (possible subclasses) and each cluster will contain only two or three elements as shown on Fig. 2.

We have the following options for recognizing such classes:

• The first option is to remove all incoming connection and to partition using the

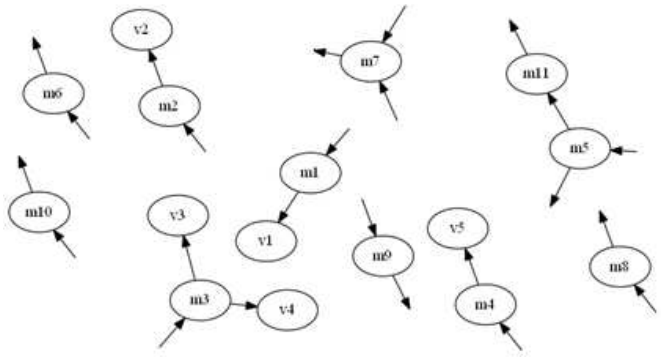

Fig. 2. Classes having low internal connectivity would be split in too many small modules

internal and the outbound connections. As a result the algorithm would recognize correctly proxies that refer to different classes. However classes that use internal data will not be recognized correctly. Classes that wrap a single object or proxy to a single interface should be generally acceptable, but may be wrongly recognized as suitable for partitioning.

- The second option is to remove both the incoming and the outgoing calls and to introduce a minimum size for the class. This would again recognize otherwise acceptable classes, but the size constraint will ensure that their count is relatively small and those classes are suitable for refactoring anyway.
- The third option is to remove and leave the incoming calls, but remove the outgoing ones. We partition the class depending on the incoming calls. Here we may group the external methods in classes or even in modules (or in case of Java [6] in packages). This approach may generate wrong results if otherwise different parts of the class are used in the same places.

All the approaches have their problem and advantages, so we chose the second one, because it generates the most consistent results and is most easy to implement.

3. An algorithm for recognizing suitable classes. The algorithm for recognizing classes suitable for partitioning, that we are presenting has the following parameters. The software engineer may tune up the algorithm application for particular case, according his own expertise or software project needs through setting appropriate values for the parameters below:

- ***minSize*** – classes having less than minSize elements are not considered at all by the algorithm. A suitable value for this parameter is 20. Classes having less than 20 elements usually don't need refactoring even if they can be partitioned.
- ***internalConnectivityMargin*** – this is the limit above which the classes will be processed as having good internal connectivity and otherwise they will be processed as having low internal connectivity. The internal connectivity is calculated as the ratio between the number of the internal connections in the class and its elements.
- ***minUnconnectedSize*** – classes having low internal connectivity must have at least minUnconnectedSize elements in order to be classified as suitable for refactoring. The value of this parameter must be significantly greater than ***minSize***.

- ***clusterSizeDeviation*** – this parameter defines the maximum acceptable deviation of the size of each cluster in a class compared to the average cluster size in this class, in order to report the class as suitable for refactoring. This allows us to avoid cases when a single class having 103 elements is partitioned into subclasses having 100 and 3 elements respectively.
- All other parameters of the used clustering algorithm are also considered as parameters of the current algorithm. Their values may vary significantly than the values suitable for partitioning the whole product into modules.

The algorithm itself goes as follows:

1. A statistical call graph for the product is created. The graph must contain information both about the direct calls between the methods and about the field references.
2. A list of the classes in the graph is created, remembering for each class the number of its elements and the number of its internal connections.
3. The classes having more than ***minSize*** elements are stored in separate list.
4. All classes in the list that don't have any outgoing connections are removed. In this way all classes that belong to external libraries are filtered. These classes are not parsed and don't have any internal or outbound connections. This also eliminates classes containing only data but no methods or other logic.
5. All classes from the list in step 4 that have internal connectivity less than ***internalConnectivityMargin*** are moved into a separate list.
6. The classes in the list from step 4 which have more than ***minUnconnectedSize*** are proposed for refactoring. If the value of ***internalConnectivityMargin*** is low enough we may be sure that these classes can be successfully partitioned because the number of internal connections is very small.
7. From the list created at step 4 a new list containing those classes having internal connectivity greater or equal to ***internalConnectivityMargin*** is created.
8. For each of the classes in step 7 the following sub steps are performed:
    a. A statistical call graph containing only the elements of the current class and their neighbors is created.
    b. All inbound connections to the elements of the class from outside the class are removed.
    c. All methods outside the current class that have neither inbound nor outbound connections after the previous step are removed.
    d. The resulting graph is partitioned using a suitable algorithm.
    e. All elements from the resulting graph, which don't belong to the current class are removed.
    f. All clusters from the resulting graph that have no elements after the previous step are deleted.
    g. The value ***mediumSize*** is calculated as the average of the sizes of all clusters in the class except the special clusters "Weak" and "Obsolete". If all the clusters except "Weak" and "Obsolete", have between $\frac{mediumSize}{clusterSizeDeviation}$ and $mediumSize.clusterSizeDeviation$ elements, the algorithm continues to the next step. Otherwise the class is considered as not suitable for refactoring and the algorithm continues to the next class. The purpose of this step is

to eliminate all classes that have too big difference between the sizes of the different clusters. Refactoring such classes will not improve the readability of the code.

h. If there are at least two clusters left after step 8g except "Weak" and "Obsolete", the current class is added to the list of classes suitable for refactoring.

4. Speed and complexity. We will examine the complexity of the algorithm based on the complexity of its steps. As a base for the calculation we will use the number of the nodes n in the original statistical call graph and the number of edges m in the same graph.

Step 1 has a complexity of $O(n)$, because in order to create the graph we pass through each of the methods in the product exactly once. At the same time it has complexity of $O(m)$ because each call in the analyzed software should be processed once.

Step 2 has a complexity of $O(n^2)$, because in order to define the classes and calculate their metrics we have to go through each method and for each of the methods to process all its connections. In the worst case when each node is connected to all other nodes, this would result in $O(n^2)$ iterations. In the common case the complexity is about $O(n)$. In the same time the complexity of this step is $O(m)$ because each edge in the graph is processed once.

Steps 3, 4, 5, 6 and 7 have complexity of $O(n)$, because for each of them it is necessary to process the nodes in the graph not more than once and in the worst case we need to process all the nodes. These steps do not depend on the number of edges in the graph, so they have complexity of $O(1)$ regarding m.

We will consider step 8 in more details because it consists of several sub steps, some of them having a significant complexity. In the worst case we will have $n/\text{minSize}$ classes to process, and for each of them we must execute steps 8a to 8f.

Step 8a has complexity of $O(n)$, because all the nodes of the original call graph must be processed – some of them will be deleted and some will remain in the new graph. The complexity regarding the number of edges is $O(m)$, because each edge should be processed in order to delete or copy it.

Step 8b is interesting – it is necessary to process all the elements of the class and to delete all incoming connections. In the worst case if each element of the graph is connected to all other nodes, we will have to perform $(n-k).k$ iterations, where k is the number of the elements in the current class. The maximum value of this expression is achieved when k equals $n/2$. But combined to the base loop of step 8, the complexity can't be greater than $O(n^2)$, because processing all the classes (the outer loop) and processing all the elements of each class (Step 8b) result in processing all the elements in the class at most once. Additionally each of the elements has at most $n-1$ neighbors. In fact this step combined with the main loop of step 8 processes all the edges in the graph, having complexity of $O(m)$.

Step 8c has a complexity of $O(n)$, because all the elements of the call graph that are not removed, must be processed. In the worst case this will result in processing all the nodes of the original graph. This step does not depend on the number of edges in the graph, so it has complexity of $O(1)$ regarding m.

Step 8d has complexity of $O(n^3)$ or $O(m.\log(m))$, which is the complexity of our algorithm for clusterization [7]. Using another algorithm would change the complexity of this step. At this step it is still possible that the number of the nodes in the current

graph is at the same order as the original call graph.

Steps 8e and 8f have a complexity of $O(n)$ each. Depending on the implementation it is even possible to combine these steps together. Again both steps don't depend on the number of edges in the graph so they have complexity of $O(1)$ regarding m.

Step 8g has a complexity of $O(n)$, because each cluster must be processed and in the worst scenario they will be as many as the nodes in the graph. This step does not depend on the number of edges of the graph so it has a complexity of $O(1)$ regarding m.

Step 8h has complexity of $O(n)$ because in the main loop of step 8 in the worst scenario all the nodes of the graph would be processed. Regarding m this step has complexity of $O(1)$ because it doesn't depend on the number of the edges in the graph.

From all the above we may state that the complexity of the algorithm is $O(n^4)$ or $O(n.m.\log(m))$. The number of about n^4 operations can be achieved when we have too highly connected classes. Usually such classes are not suitable for partitioning and this rarely happens in practice. So if we limit the maximum internal and external connectivity of the processed classes, we will significantly decrease their count and will not decrease the accuracy of the algorithm significantly.

Let us consider several runs of the algorithm on different versions of Ant [8] and compare the real speed of the algorithm.

Table 1. Times to perform the algorithm on different versions of Ant

Version	Number of nodes in the call graph	Number of edges in the call graph	Time (s)
1.3	16578	27443	8.32
1.5	32991	55687	27.86
1.7	52394	88782	67.69

As can be seen from the figure, the time for each run had increased by not more than the square of the number of elements of the graph. Too far from $O(n^4)$, which is expected by the algorithm, but near to $m.n.\log(m)$. Meanwhile, even the speed for Ant 1.7 is good enough to make the algorithm suitable for practical application.

5. Results and experiments. In this part we will analyze the result of running the algorithm on the same three versions of Ant – 1.3, 1.5 and 1.7. We chose Ant for testing because its sources are freely available and we have the sources for several of its releases. This ensures that our results can be reproduced at any time and gives us the additional benefit of analyzing the development of the analyzed software over time.

We set the values of the ***groupWeaks*** and ***groupSingles*** parameters of the clustering algorithm to false, the value of margin to 20, and the value of ***minClusterSize*** to 5. All other parameters are left with their default values.

As a result we got 10 classes suitable for refactoring for Ant 1.7. Only one of these classes has low internal connectivity. As it can be seen on the table above, some of the classes have a little more than 20 elements. So if the value of ***minSize*** is increased to 30, there will be only 5 classes left.

Ant 1.5 has 9 classes suitable for partitioning when using the same values for the parameters. Two of them have low internal connectivity. For all the classes that have high internal connectivity the algorithm proposes that they are split into two modules.

Things are different in Ant 1.3 – there are only 4 classes proposed for refactoring, but two of them are partitioned into three and four subclasses, respectively.

Table. 2. Classes suitable for refactoring in Ant 1.7

Class name	**Number of elements**
EjbcHandler	22
BorlandGenerateClient	21
Depend	40
TarBuffer	32
JLink	22
Concat	43
PropertyHelper	28
ConditionTest	53
ZipFile	24
Checksum	40

There are some interesting facts that we observed after performing all the executions:

The "TarBuffer" class is considered as suitable for refactoring since Ant 1.3. has not been refactored till Ant 1.7. At the same time it stays at the size of 22 elements. This shows that this class doesn't change over time and no one has paid attention that there might be problems there.

The "Jlink" class is considered for refactoring in Ant 1.3, but is skipped in Ant 1.5, only to appear again in Ant 1.7. with the same size. After analyzing the source code, we saw that the reason for this behavior is that in Ant 1.5 one of the methods have been complicated, causing the union of the two clusters.

The result of changing the values of ***minSize*** and ***minUnconnectedSize*** are predictable so there is no need to consider them separately. The effect of changing the value of ***internalConnectivityMargin*** is much more interesting. Decreasing these values doesn't change the number of the resulting classes, because the internal connectivity of all the classes considered having low connectivity is near zero. The increase of the value of ***clusterSizeDeviation*** from 2 to 3 increased the number of the discovered classes for Ant 1.3 from 4 to 6 and for Ant 1.5 from 9 to 12.

The effect of changing the parameters of the clustering algorithm is also significant. Changing the value of ***minClusterSize*** from 5 to 10 adds two more classes to the results of Ant 1.5. As a common rule, settings that allow partitioning of the classes into more parts would lead to significant increase of the number of the resulting classes, especially when combined with larger values of clusterSizeDeviation.

6. Displaying the results. The approach which we chose for displaying the result of the algorithm consists of two parts.

In the first part all elements in the original call graph are grouped in clusters corresponding to the classes proposed for refactoring by the algorithm. This approach gives the user the ability to have general view of the results as shown on Fig. 3.

Each circle in the figure above corresponds to a class that is suitable for splitting according to the presented algorithm. The sizes of the circles depend on the number of elements in the corresponding classes. The locations of the classes depend on the initial locations of the elements of the class in the call graph and are random in the general case.

In the second part of the visualization, the elements belonging to each of the discovered

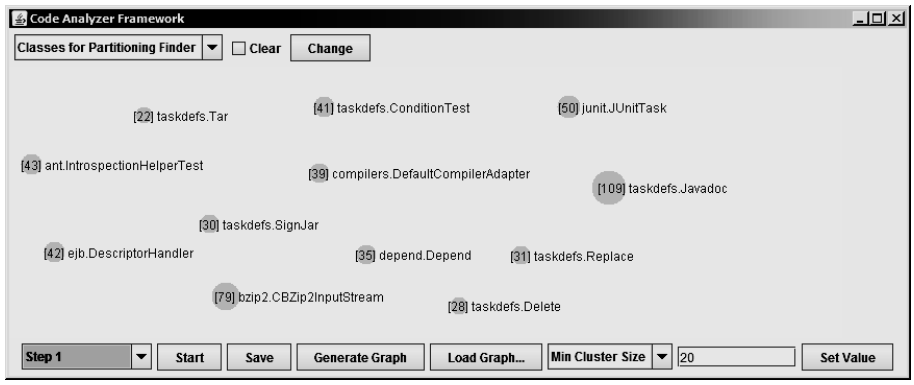

Fig. 3. A general view of the results of running the algorithm

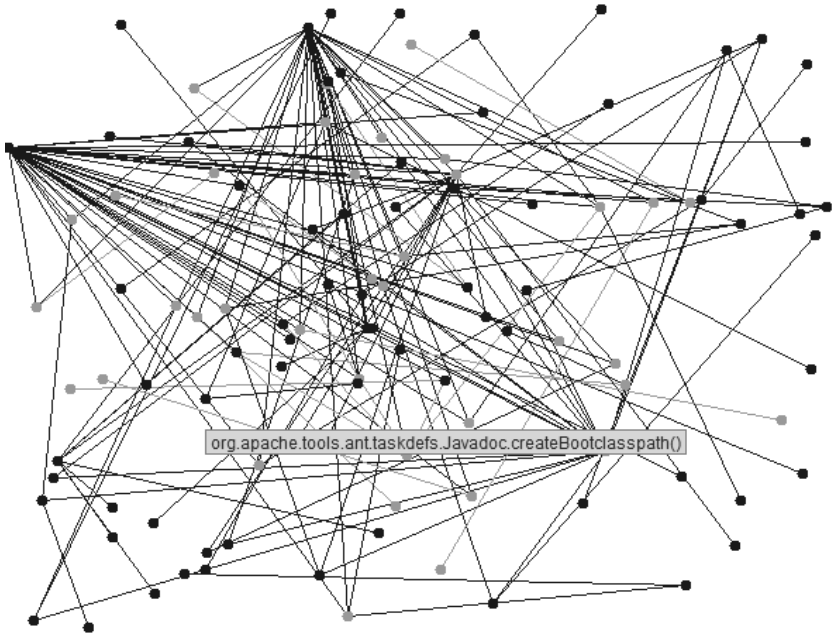

Fig. 4. The suggested partitioning of the JavaDoc class

classes are filtered from the call graph (the elements of each class are filtered in separate graph). The elements of each proposed subclass are colored differently. The names of the nodes are shown when right clicking on the desired node. Showing them on the graph would make the result unusable.

In the Figure 4 it is proposed that the JavaDoc class is split into two classes – one containing the elements corresponding to the light dots and the other containing the elements corresponding to the dark dots. The positions of the dots in the visualization correspond to their positions in the original call graph. In the general case they are

random but this depends on the call graph creation algorithm. The names of the elements are not shown in the visualization, but may be seen when hovering over an element as shown on the figure above. Alternatively the list of elements contained in each subclass may be displayed.

In order to distinguish the classes chosen because of their low internal connectivity in the similar figure, their elements are colored black. For these classes the algorithm doesn't propose a suitable partitioning, but rather recommends to the developer to pay special attention to them.

The proposed visualization allows the user to see both the general results of the algorithm and the concrete results and suggestion for refactoring for each class, so it is easier for the user to decide whether the proposed changes are good enough.

7. Conclusion. The proposed algorithm gives the user the ability to automatically detect classes suitable for partitioning in products containing a lot of classes. Because of its high speed the algorithm is suitable for daily use. The proposed visualization gives both general and specific view of the proposed changes so the developers can easily take decision what to change and what not.

Acknowledgments. The authors gratefully acknowledges the useful notes from the anonymous reviewer.

REFERENCES

[1] M. Fokaefs et al. JDeodorant: Identification and Application of Extract Class Refactorings. ICSE, Waikiki, 2011.

[2] G. Bavota, A. De Lucia, R. Oliveto. Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, **84** (2011), No 3, 397–414.

[3] F. Simon, F. Steinbruckner, C. Lewerentz. Metrics based refactoring. Fifth European Conference on Software Maintenance and Reengineering, 2001, 30–38.

[4] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[5] B. Ryder. Constructing the call graph of a program. *IEEE Trans. Software Eng.*, **5** (1979), 216–226.

[6] J. Gosling et al. The Java Language Specification, 2014.

[7] T. Cholakov, D. Birov. A Compound Algorithm for Clustering Software. *Math. and Education in Math.*, 43 (2014), 157–166.

[8] Apache Ant User Manual, 11 19, 2014, `http://ant.apache.org/manual/`.

Todor Plamenov Cholakov
e-mail: todortk@abv.bg
Dimitar Yordanov Birov
e-mail: birov@fmi.uni-sofia.bg
Faculty of Mathematics and Informatics
University of Sofia
5, James Bourchier Blvd
1164 Sofia, Bulgaria

ОТКРИВАНЕ НА КЛАСОВЕ, ПОДХОДЯЩИ ЗА РАЗДЕЛЯНЕ НА ЧАСТИ

Тодор Чолаков, Димитър Биров

В тази статия разглеждаме алгоритъм и инструмент за откриване на класове, подходящи за разделяне на два или повече по-малки и по-добре капсулирани класа. Даваме оценка на сложността и бързодействието на предложения алгоритъм и демонстрираме експерименти върху софтуерни продукти с отворен код. За целите на експериментите с алгоритъма разработихме специален инструмент. Той позволява на програмиста да има както общ поглед върху резултатите, така и да получи детайлна информация за предложените по отношение на всеки клас промени. Въпросният инструмент се доказа като полезен за анализ на приложения от практиката.