periodica polytechnica

Electrical Engineering
52/3-4 (2008) 153–162
doi: 10.3311/pp.ee.2008-3-4.04
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

BEFRIEND – a benchmark for evaluating reverse engineering tools

Lajos Jenő Fülöp / *Péter* Hegedűs / *Rudolf* Ferenc

Received 2008-07-22

Abstract

Reverse engineering tools analyze the source code of a software system and produce various results, which usually point back to the original source code. Such tools are e.g. design pattern miners, duplicated code detectors and coding rule violation checkers. Most of the time these tools present their results in different formats, which makes them very difficult to compare.

In this paper, we present work in progress towards implementing a benchmark called BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe) with which the outputs of reverse engineering tools can be easily and efficiently evaluated and compared. It supports different kinds of tool families, programming languages and software systems, and it enables the users to define their own evaluation criteria. Furthermore, it is a freely available web-application open to the community. We hope that in the future it will be accepted and used by the community members to evaluate and compare their tools with each other.

Keywords

Benchmark · reverse engineering tools · tool evaluation · code clones · design patterns

Acknowledgement

This research was supported in part by the Hungarian national grants RET-07/2005, OTKA K-73688, TECH_08-A2/2-2008-0089 and by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

Lajos Jenő Fülöp
University of Szeged, Department of Software Engineering, Hungary
e-mail: flajos@inf.u-szeged.hu

Péter Hegedűs
Rudolf Ferenc
University of Szeged, Department of Software Engineering, Hungary

1 Introduction

Nowadays, the development of large software systems comprises several steps. The general tendency is that due to the tight deadlines programmers tend to leave certain steps out of the development process. This usually means omitting the appropriate system documentation, which can make the maintenance and comprehension of the software rather difficult. This problem can be alleviated with the automatic recognition of certain source code characteristics. The design pattern [12] recognizing tools – which by recognizing design patterns can make the source code more comprehensible and transparent – are good examples of this. Several of these tools have been introduced in literature, and so far they have proved to be rather efficient. Despite all this, it would be difficult to state that the performance of design pattern recognizing tools is well-defined and well-known as far as accuracy and the rate of the recognized patterns are concerned. In order to obtain more information about these characteristics, the results of the tools require in-depth evaluation. So far, this has been quite difficult to achieve because for the comparison of different tools a common measuring tool and a common set of testing data are needed. To solve this problem, we developed the DEEBEE (DEsign pattern Evaluation BEnchmark Environment) benchmark system in our previous work [10].

The current work introduces the further development of the DEEBEE system which has become more widely applicable by generalizing the evaluating aspects and the data to be indicated. The new system is called BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe). With BEFRIEND, the results of reverse engineering tools from different domains recognizing the arbitrary characteristics of source code can be subjectively evaluated and compared with each other. Such tools are e.g. bad code smell [9] miners, duplicated code detectors, and coding rule violation checkers.

BEFRIEND largely differs from its predecessor in five areas: (1) it enables uploading and evaluating results related to different domains, (2) it enables adding and deleting the evaluating aspects of the results arbitrarily, (3) it introduces a new user interface, (4) it generalizes the definition of sibling relationships [10], and (5) it enables uploading files in different formats

by adding the appropriate uploading plug-in. BEFRIEND is a freely accessible online system available at http://www.inf.u-szeged.hu/befriend/.

We will proceed as follows: In the following section, we will provide some background needed to understand the main concepts behind BEFRIEND. In Section 3, we will describe the use of the benchmark on concrete examples and scenarios. Next, in Section 4, we will show the results of a small experiment demonstrating the usefulness of the benchmark. Afterwards, in Section 5, we will discuss some works that are related to ours. We will close our paper with conclusions and directions for future work in Section 6.

2 Background

In this section, we present the theoretical concepts that are indispensable for the understanding of BEFRIEND. BEFRIEND serves the evaluation of tools working on source code, which hereafter will be called *tools*. The tools can be classified into domains. A *domain* can be a tool family searching e.g. for design patterns, code clones and bad smells, or rule violations. Without the sake of completeness, design pattern searching tools are e.g. DPD [7], Columbus [2] and Maisa [8], and duplicated code searching tools are e.g. Bauhaus [3], CCFinder [6] and Simian [19]. The tools in a given domain produce different results which refer to one or more positions in the analyzed source code. We refer to these positions as result *instances*. The found instances may include further elements in certain domains. These are called *participants*. For example, in the case of a Strategy design pattern instance, several ConcreteStrategy *participants* may occur. In each *instance*, the *participants* can be typed according to *roles*. In the case of the Strategy design pattern, the *roles* are Context, Strategy, and ConcreteStrategy. For the evaluation of tools several *evaluation criteria* can be defined. With the help of the evaluation criteria, we can evaluate the found instances of the tools.

Many times it may happen that several instances can be grouped, which can largely speed up their evaluation. For example, if two clone detecting tools together find 500 clone pairs (most clone detecting tools find clone pairs), then by grouping them, the number of clone pairs can be reduced to a fraction of the original instance number. In another case, if one of the clone detectors finds groups of instances (e.g. 30), and the other one finds clone pairs (e.g. 400), the reason for the latter tool to find more instances is that its output is defined differently. Relying on this, it can be said that without grouping, the interpretation of tool results may lead to false conclusions.

However, grouping is a difficult task since the tools very rarely label the same source code fragment. This can have a number of reasons, e.g.:

- The tools use different source code analyzers, which may cause differences between the instances.
- One tool gives a wider scale of instances than the other. For example, in the case of code clone searching tools, one tool finds a whole class as one clone, while the other finds only two methods in the class.
- One tool labels the opening brackets of a block, while the other does not.

2.1 Siblings

In order to group instances, their relation needs to be defined. If two instances are related to each other, they are called *siblings*. Basically, three facts determine the existence of the sibling relation between two instances:

- the matching of their source code positions
- the minimal number of matching participants
- domain dependent name matching

By using these three facts, instances can be connected. In the followings, we examine these three cases in detail.

Source code positions Sibling relations are mostly determined by the matching of the source code positions. The *contained*, the *overlap* and the *ok* relations defined by Bellon et al. [4] can be well applicable for matching. We adopted the *contained* and *ok* relations with a little modification as a *contain* relation, because these two relations had originally been used to compare clone instances where every instance contained exactly two participants.

Let P and Q be participants with the same roles. If the roles are not the same then, for example, a ConcreteStrategy participant could be incorrectly connected with a Context participant. So, the *contain* and the *overlap* relations are defined between P and Q in the following way:

$$contain(P, Q) = \max\left(\frac{|P \cap Q|}{|P|}, \frac{|P \cap Q|}{|Q|}\right)$$

$$overlap(P, Q) = \frac{|P \cap Q|}{|P \cup Q|}$$

$$\text{where } P \cdot role = Q \cdot role$$

In the case of the *contain* and the *overlap* relations, the set operations between the participants are applied to the source code lines of P and Q. These two relations use the positions of the participants, they will be used together in the following as $match_p$:

$$match_p(P, Q) = contain(P, Q) \oplus overlap(P, Q)$$

In the case of $match_p$, we denote either the contain or the overlap relation (exclusive or). The match_p relation has a value between 0 and 1[1]. The match_{pb} relation denotes if P and Q

[1] We project this value into the interval from 0 to 100 on the graphical user interface for the sake of easier understanding and use.

have a match_p value above a given *bound* or not:

$$match_{pb}(P, Q, bound) = \begin{cases} 1 & \text{if } match_p(P, Q) \geqslant bound \\ 0 & \text{otherwise} \end{cases}$$

With the use of the $match_{pb}$ relation, the $match_i$ relation between the instances can be defined. The $match_i$ relation denotes how many times the $match_{pb}$ relation has the value of 1 for P and Q for a given *bound*. The $match_i$ relation is defined as follows:

$$match_i(I, J, bound) = \sum_{P \in I} \sum_{Q \in J} match_{pb}(P, Q, bound)$$

where I and J are two instances.

Minimal number of matching participants. Sometimes it is not enough to match only one participant between two instances. For example, in the case of design patterns, the Abstract Factory pattern should have two participants, Abstract Factory and Abstract Product, for matching. For this reason, it is important to determine the minimal number of common participants between two siblings. Let us denote the minimal number with m. The *sibling relation* between two instances I and J with parameters m and *bound* is defined in the following way:

$$sibling(I, J, bound, m) = \begin{cases} \text{true} & \text{if } match_i(I, J, bound) \geqslant m \\ \text{false} & \text{otherwise} \end{cases}$$

The instances can be grouped based on the sibling relation. A *group* contains instances that have a true sibling relation in couples. By using groups, the evaluation of the tools is easier, and the statistical results are better. In BEFRIEND the users can customize the sibling relations by choosing between the *contain* and the *overlap* relations arbitrarily, giving the *bound* and m parameters, and optionally selecting the *roles* for matching.

Domain dependent name matching. In certain domains, the roles are not so important (e.g. code duplications have only one role called *clone*). However, if a domain contains several roles, some roles can be more important than the others. For example, in the case of a Strategy design pattern, basically the Strategy participant determines an instance and therefore, the sibling relation should be based on this participant. In the case of such domains the $match_{pb}$ relation has to be modified. Let *roles* be a set that denotes the roles that are the basis of the sibling relations between the instances. The $match_{pb}$ is redefined as $match_{pb'}$:

$$match_{pb'}(P, Q, bound, roles) = \begin{cases} match_{pb}(P, Q, bound) & \text{if } P \cdot role \in roles \\ 0 & \text{otherwise} \end{cases}$$

In the case of domains where roles are important, the $match_{pb}$ relation has to be replaced with $match_{pb'}$ in the formula of $match_i$.

Extensions. With the previous definitions, it is possible that a group contains two or more instances which are not siblings because the *contain* and the *overlap* relations are not transitive. Now, we do not allow these cases, the groups have to be transitive. After all, it can distort the statistics and the comparison because some instances can be repeated across the groups. Therefore, we want to deal with this problem in the future in more detail. Several solutions, for example, a second bound parameter which would denote the minimal rate of the sibling relations between each instance in a group, might be introduced.

3 Scenarios of use

In this section, we present the use of the system with the help of some scenarios illustrated with pictures. In each case, first we give a general description about the discussed functionality, and afterwards we demonstrate it with a concrete example. The examples are continuous, they are built on each other, and they present the use of the benchmark.

3.1 Setting up the database

In this scenario, we present how the user can create a new domain and evaluation criterion, how he can upload data in the system, and how he can set siblings. The functions that help to do the necessary settings can be found under the *Settings* and *Upload* menus.

Creating a new domain. As a first step, a new domain has to be created according to the data to be uploaded. For the creation of a new domain, the *Domain settings* panel has to be used.

Fig. 1. Creating a new domain

Example: We will upload the results of a duplicated code detecting software. First, with the help of the *Domain settings* panel, we create a new domain which is called *Duplicated Code* (see Fig. 1). As a result, the actual domain is set to the newly created Duplicated Code. If we have created more than one domain, we can select the domain we would like to activate from the *Select active domain* drop-down list.

Creating new evaluation criteria. In order to be able to evaluate the uploaded data, appropriate evaluation criteria are

needed. The user can create an arbitrary number of criteria for every domain. On the basis of this, the uploaded instances can be evaluated. In one evaluation criterion, one question has to be given to which an arbitrary number of answers can be defined. Similarly to the domain, we can create a new criterion under the *Settings* menu. When creating a new criterion, the following data should be given:

- The title of the evaluation criterion (Title).
- The question related to the criterion (Question).
- The possible answers to the question (Choice). To each answer a percentage ratio should be added to indicate to what extent the given question is answered. Relying on the answers of the users, the benchmark can calculate different statistics on the basis of this ratio.
- Should the *precision* and *recall* values be calculated?

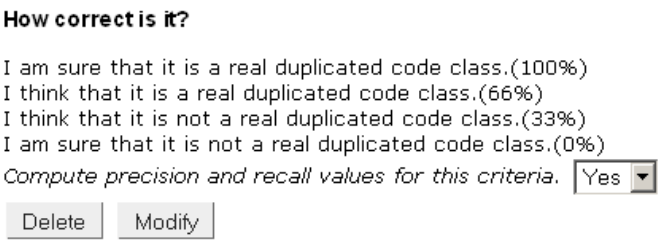

Fig. 2. Correctness criteria

Example: We will create an evaluation criterion, *Correctness*, for the previously created *Duplicated Code* domain. Write number 4 (meaning that we will have four possible answers) in *The number of choices within the created criteria* field in the *Evaluation criteria* panel and click on the *Create new criteria* button. After filling out the form that appears, by clicking on the *Submit* button the criterion appears on the setting surface (see Fig. 2). The *Correctness* criterion is used to decide to what extent a code clone class comprises cloned code fragments. For the criterion, we have also set that the precision and recall values should be calculated.

During the evaluation, we will use two other criteria: One of them is *Procedure abstraction* with the related question: *Is it worth substituting the duplicated code fragments with a new function and function calls?* And the possible answers are:

- Yes, we could easily do that. (100%)
- Yes, we could do that with some extra effort. (66%)
- Yes, we could do that but only with a lot of extra effort. (33%)
- No, it is not worth doing that. (0%)

With this, we define how much effort would be needed to place the duplicated code fragments into a common function. The easier it is to introduce a function, the more useful the found instance is on the basis of this criterion. This is an important indicator because a tool that might not be able to find all the clones but is able to find most of the easily replaceable ones is worth much more regarding refactoring.

The third criterion is *Gain* with the related question: *How much is the estimated gain of refactoring into functions?* The answers:

- The estimated gain is remarkable. (100%)
- The estimated gain is moderate. (50%)
- There is no gain at all. (0%)

It is important to evaluate how much benefit could be gained by placing the code fragments into a function. It might occur that something is easily replaceable, but it is not worth the trouble since no benefit is gained.

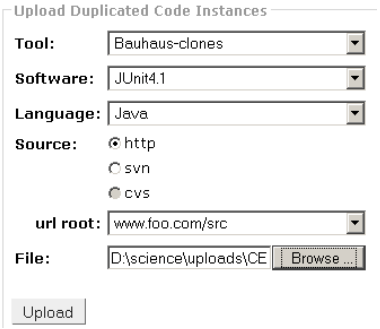

Fig. 3. Upload

Uploading data into the benchmark. When the user has created the appropriate domain and the required evaluation criteria, he has to upload the instances produced by the tools to be evaluated. The upload page can be found under the *Upload* menu. The format of the files to be uploaded is completely arbitrary, the only condition is that the evaluated tool should provide a BEFRIEND plug-in to perform the uploading. The benchmark provides a plug-in mechanism where by implementing an interface the output format of any tool can be uploaded. Currently, the benchmark has plug-ins for the output formats of the following tools: Columbus, CCFinderX, Simian, PMD and Bauhaus.

For uploading, the following data are needed: the name of the tool, the name and programming language of the analyzed software. If a data has not occurred earlier, we can add the new data to the list by selecting the *New* item. The uploaded data refer to a certain source code, so it is important to give some kind of access to these sources. Currently, the benchmark supports two access methods, *http* and *svn* (*cvs* will also be implemented in the near future).

Example: As the continuation of the evaluation process, we upload the results produced by the *clones* command line program of the Bauhaus tool into the system. First, click on the *Upload* menu, which will take you to the upload page (see Fig. 3). After giving the required data and the path of the file containing the results, click on the *Upload* button to perform the uploading.

Sibling setting. Siblings can enable that the instances with similar properties are handled together during evaluation. Settings can be done under the *Settings* menu, on the *Siblings settings* panel (see Fig. 5). The user can choose the *Contain* or the *Overlap* relation. In the *Path matching bound* field, the minimum matching bound of the participants should be given, while in the *Min. number of matching participants* field, the number of participants whose matching we demand from two sibling instances should be given. All setting configurations should be named; the system saves the different settings. Thus, if we want to use an already saved setting in the future, we can load it by selecting it from the *Load shortcut* menu. The system will not rerun the sibling grouping algorithm.

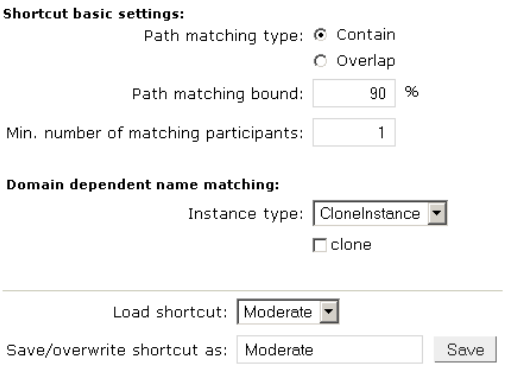

Fig. 4. Sibling settings

Example: As the last step of the scenario, we set the sibling parameters for the uploaded results (see Fig. 5). For linking, we use the *Contain* relation where the matching bound is set at 90%. The reason for choosing this bound is that this way the instances can be linked even in the case of short code clones (10 lines). The ratio of the *Min. number of participants to match* is 1, which means that it is enough to have two matching participants to link two clone instances. We save this configuration as *Moderate*.

3.2 Data evaluation

The basic task of the benchmark is to enable the visualization and evaluation of the uploaded data. In this section, we summarize these functions.

Query view. The uploaded instances can be accessed through the *Evaluation* menu. As a first step, the *Query view* appears (see Fig. 6), which helps the user define the aspects on the basis of which the uploaded instances are listed. There are four possible aspects: *Tool*, *Software*, *Language* and the active domain (e.g. *Duplicated Code*). At the bottom of the view, there is a *Connect siblings* check box. If this box is marked, the instances appear in groups, not individually.

To every aspect, different values deriving from the data of the already uploaded instances can be added. For example, when selecting one particular tool from the tool menu only the instances found by the given tool will appear.

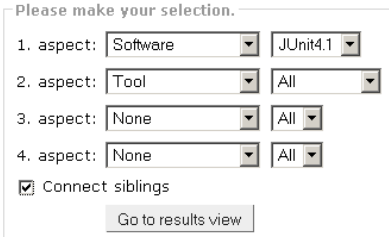

Fig. 5. Query view

With further settings, even narrower instance sets can be defined. If the user would like to skip an aspect when filtering, that aspect should be set to the *None* value. If he would like to see all the items related to the selected aspects, the *All* value should be used. The sequence order of aspect selection is arbitrary.

Example: Select the *Evaluation* menu. The *Query view* page appears (see Fig. 6). In the Fig. 6, the first aspect is *Software* whose value is set to *JUnit4.1*. The second aspect is *Tool* whose value is *All*, the third and fourth aspects are not used. We have also activated the *Connecting siblings* check box. By clicking on the *Go to results view* button, we will get to the view of instances satisfying the given aspects (see Fig. 6).

Results view. The columns of the table correspond to the aspects defined in the *Query view*, with the exception of the last column. Here, the instance identifiers can be seen. By clicking on the identifiers, the *Individual instance view* or the *Group instance view* appears depending on whether the user has set the grouping of siblings in the *Query view*.

Example: In the first column of the table, JUnit4.1 can be seen according to the aspects set in the Query view previously (see Fig. 6). Since all the tools were selected, the second column of the table contains every tool found in the database. The third column comprises the identifiers of the duplicated code instances found by the tools. It can be seen that not all identifiers are present in the table since we have set the grouping of siblings in the Query view, so here the groups appear with the smallest instance identifiers.

Group instance view and Individual instance view. The Instance view is used to display and evaluate instances. BEFRIEND supports two kinds of instance views, *Group instance view* and *Individual instance view*. While the *Group instance view* can display more instances simultaneously, the *Individual instance view* displays the participants of only one instance in every case. Apart from this difference, the two views are basically the same.

In the *Results view*, the *Instance view* can be accessed by clicking on an instance identifier. If the user has set the grouping of siblings in the *Query view* (see Fig. 6), the system automatically applies the *Group instance view*. Otherwise it applies the *Individual instance view*.

Group instance view: The *Participants* table comprises the

Software	Tool	Duplicated Code ids									
JUnit4.1	Bauhaus-clones	#2	#4	#5	#6	#7	#8	#9	#10	#15	#16
		#17	#18	#19	#21	#22	#23	#24	#25	#26	#28
		#29	#31	#32	#33	#34	#40	#43	#44	#45	#46
		#49	#50	#51	#52	#53	#54	#3	#59	#60	#62
		#41	#64	#69							
	Simian	#31	#7								
	PMD	#7	#60	#31	#17	#45					
	CCFinderX	#31	#17	#43	#588	#589	#44	#33	#593	#34	#23
		#23	#610	#60	#614	#18	#619	#57	#623	#4	#69
		#29	#63	#52	#50	#626	#7	#640	#641	#15	#54
		#644									
	Columbus	#624	#646	#17	#7						

Fig. 6. Results view

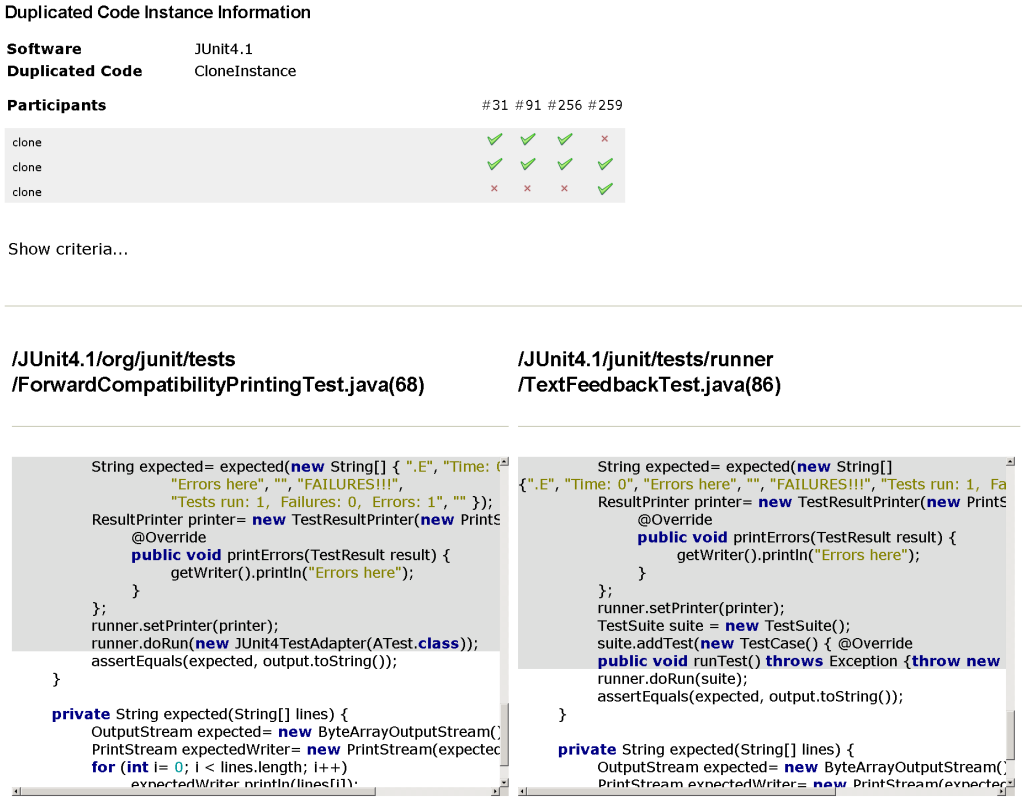

Fig. 7. Group instance view

participants of the instances; the different participants of all the siblings appear. The table has as many columns as the number of instances in the group. Each column corresponds to a given instance whose identifiers are indicated at the top of the column. In the intersection of a row of participants and a column of instances, either a green $\surd$ or a red $\times$ symbol appears. The green $\surd$ means that the instance in the given column comprises the participant in the row. The $\times$ symbol means that this particular instance of the group does not comprise such a participant. By clicking on the green $\surd$ with the right mouse button, a pop-up menu appears with the help of which the participant's source code can be displayed. It can be selected from the menu whether the source code should be displayed on the left or on the right side of the surface under the table. This is very useful because e.g. the participants of the duplicated code instances can be seen next to each other.

The evaluation criteria and, together with them, the instance votes can be accessed by clicking on the *Show criteria...* link under the table of participants (see Fig. 7). The statistics of the instances can be accessed by clicking on the *stat* link above the evaluation column corresponding to a given instance. By moving the mouse above any of the green $\surd$ symbols, a label appears with the source code name of the given participant (if it has one). Comments can also be added to the instances. An instance can

be commented by clicking on the instance identifier appearing above the Participants table. This window even comprises the information that reveals which tool found the given instance.

Example: Click on instance #31 in the *Results view* created in the previous example (see Fig. 6). The *Group instance view* appears (see Fig. 7). Right-click on any of the green √ symbols, and select *Open to left* in the appearing pop-up menu. Right-click on another green √ symbol, and select *Open to right* in the menu. This way the source code of two clone participants will be displayed beside each other. After examining the participants of all the four instances belonging to this group, the evaluation criteria can be displayed by clicking on the *Show criteria...* link. Here, the instances can be evaluated.

Statistics view. This view can be reached from the *Query view* by clicking on a link other than an instance identifier. Here, relying on the structure seen in the *Results view*, the user gets some statistics according to the evaluation criteria, based on the previous user votes (see Fig. 8). One table that comprises the vote statistics referring to all of the concerned instances belongs to each evaluation criterion. In the first column of the table, the instance identifiers appear. The identifier of the grouped instances is red, while that of the others is blue. All categories can be set on the user interface; how to aggregate the statistics of the grouped instances: on the basis of the average, maximum, minimum, deviation or median of the group instance ratio. Beside the identifiers, five columns can be found where the five previously mentioned statistics of the instance votes can be seen. These five values are also calculated for each column (that is for each statistics value). If we have set in the *Settings* menu that the precision and recall values should be calculated, they will also appear under the table corresponding to the criterion.

Aspect	Mean	Deviation	Min	Max	Median
#32	66.0%	0.0%	66.0%	66.0%	66.0%
#33	83.0%	24.04%	66.0%	100.0%	83.0%
#34	33.0%	46.67%	0.0%	66.0%	33.0%
#40	83.0%	24.04%	66.0%	100.0%	83.0%
#43	16.5%	23.33%	0.0%	33.0%	16.5%
#44	33.0%	46.67%	0.0%	66.0%	33.0%
Mean	**52.1%**	**13.39%**	**42.63%**	**67.88%**	**52.1%**
Deviation	**26.92%**	**16.69%**	**31.85%**	**18.67%**	**26.92%**
Min	**0.0%**	**0.0%**	**0.0%**	**33.0%**	**0.0%**
Max	**100.0%**	**46.67%**	**100.0%**	**100.0%**	**100.0%**
Median	**66.0%**	**0.0%**	**66.0%**	**66.0%**	**66.0%**

Summary

Number of instances:	43
Number of evaluated instances:	43
Number of instances above the threshold:	27
Precision:	62.79%
Total number of instances:	56
Total number of evaluated instances:	56
Total number of instances above the threshold:	32
Recall:	84.38%

Fig. 8. Bauhaus correctness statistics

Example: After having evaluated all the instances found by the tools, we examine the statistic value we get for the votes of each tool. Let us go back from the *Group instance view* (see Fig. 7) to the *Results view* (see Fig. 6) by clicking on the *Back to previous view* link. Here, we get the statistic values of all the tools by clicking on 'JUnit4.1'. According to the 3 criteria, three tables belong to each tool. In the *Correctness* table of the Bauhaus-clones tool, the statistics of the instance votes can clearly be seen (see Fig. 8). Furthermore, it can also be seen that some instances are grouped; these are marked in red by the system. Under the table the precision and recall values are also displayed. The *Number of instances* shows the number of instances found by the tool. The *Number of evaluated instances* is the number of evaluated instances, while the *Number of instances above the threshold* is the number of instances whose votes are above the set threshold. This threshold can be controlled by setting the *Threshold for calculating precision and recall* value, which is by default 50%. This value determines which instances can be accepted as correct instances. On the basis of this, we can calculate the precision value, which is 62.79% in this case. The *Total number of instances* is the total number of instances found by all tools, the *Total number of evaluated instances* is the total number of evaluated instances, while the *Total number of instances above the threshold* is the number of instances with votes above the threshold. Relying on these values, the recall value can be calculated, which is 84.38% in the case of Bauhaus.

Comparison view. The system comprises yet another interesting view, the Comparison view. In order to make this view accessible, we have to start from a statistics view that comprises several tools. The comparison view compares the instances found by each tool. For all the possible couplings, it defines the instance sets found by two tools, and the difference and intersection of these sets. This view helps to compare the tools with each other.

A = JUnit4.1/Bauhaus-clones
B = JUnit4.1/CCFinderX

A	#23	#5	#60	#17	#10	#31	#8	#29	#54	#33
	#52	#50	#45	#7	#21	#41	#40	#15	#18	#24
	#53	#16	#32	#9	#28	#51	#4			
B	#29	#60	#15	#54	#33	#644	#593	#52	#619	#589
	#50	#7	#17	#43	#4	#31	#610			
A - B	#40	#23	#5	#9	#18	#45	#24	#28	#21	#16
	#10	#32	#51	#41	#8	#53				
B - A	#644	#593	#619	#589	#43	#610				
A & B	#29	#60	#54	#33	#52	#15	#50	#7	#17	#4
	#31									

Fig. 9. Comparison view

Example: In the *Statistics view* loaded in the previous example, click on the *Switch to comparison view* and the *Comparison view* will appear (see Fig. 9). The comparison of tools is carried out in couples, and here, we also have the opportunity to link siblings the same way as in the other views (here the grouped instances are marked in red as well).

4 Experiment

In this section, we summarize the results of the experiment made with BEFRIEND. We would like to emphasize that the aim of the experiment was *the demonstration of the capabilities of our system* rather than the evaluation of the different tools. During the evaluation five *duplicated code* finder tools were assessed on two different open source projects, *JUnit* and *NotePad++*. The tools used in the experiment were *Bauhaus* (*clones* and *ccdiml*), *CCFinderX*, *Columbus*, *PMD* and *Simian*.

More than 700 duplicated code instances were evaluated by two of us. For the evaluation, three evaluation criteria were used: *Correctness*, *Procedure abstraction* and *Gain* (see Section 3.1). The results of the tools on the two software are shown in Table 1 and Table 2.

Tab. 1. Results on NotePad++

Criteria	Bauhaus ccdiml	Columbus	PMD	Simian
Precision	100.0%	96.15%	62.5%	61.43%
Recall	5.06%	28.09%	64.61%	48.31%
Proc. abstr.	62.87%	65.59%	48.16%	48.12%
Gain	55.95%	53.37%	33.88%	34.88%

The *precision* and *recall* values presented in the tables are calculated based on the Correctness criteria. In the rows of *Procedure abstraction* and *Gain*, the average values derived from the votes given for the instances are shown according to the criteria. We would like to note that a *threshold* value is needed for calculating the precision and recall. The instances above this threshold are treated as true instances (see Section 3.2). We used the default 50% threshold value, but the threshold can be adjusted arbitrarily. For example, in case of three voters, if two of them give 66% to an instance, while the third one gives 33%, the average of the three votes is 55%. In such cases, it is reasonable to accept the instance as a true instance since two of the three voters accepted it, while only one rejected it.

Tab. 2. Results on JUnit

Criteria	Bauhaus clones	CCFinder	Columbus	PMD	Simian
Precision	62.79%	54.84%	100.0%	100.0%	100.0%
Recall	84.38%	53.13%	12.5%	15.63%	6.25%
Proc. abstr.	48.31%	44.23%	79.0%	73.0%	66.25%
Gain	29.36%	30.98%	62.5%	62.5%	62.5%

CCFinderX is missing from Table 1 because it produced a very high number of hits with the parameters we used (it found over 1000 duplicated code instances containing at least 10 lines). Due to the lack of time, we were able to evaluate only the four other tools on NotePad++.

We had some important experience during the evaluation. In the case of JUnit, PMD and Simian produced a very similar result as Columbus but our experience is that in general, the *token* based detectors (Bauhaus-clones, CCFinderX, PMD, Simian) produce substantially larger number of hits than the *ASG* based tools (Bauhaus-ccdiml, Columbus). This is partly due to the fact that while the ASG based tools find only the instances of at least 10 lines, which are also syntactically coherent, the token based detectors mark the clones that are in some case shorter than 10 lines in such a way that they expand the clone with several of the preceding and succeeding instructions (e.g. with '}' characters indicating the end of the block). On the grounds of this experience we would like to expand the evaluation with a new criterion in the near future. The new criterion would apply to the accuracy of the marking of a clone instance. It would also define what portion of the marked code is the real clone instance.

Based on the *Gain* and *Procedure abstraction* values of certain tools, we can say that the ASG based detectors find less but mostly *more valuable* and *easily refactorable* clone instances. On the contrary, the token based tools find more clone instances producing a *more complete* result.

We would like to note that we also imported the design pattern instances we had evaluated with DEEBEE in our previous work [10] into our system. Besides these, the system contains coding rule violation instances found by PMD and CheckStyle as well. The evaluation of these instances is also in our future plans.

5 Related work

In this section, first we introduce Sim's work and later we analyze the evaluating and comparing techniques related to the different domains.

Sim et al. [17] collected the most important aspects, properties and problems of benchmarking in software engineering. They argued that *benchmarking has a strong positive effect on research*. They gave a definition for benchmarking: "a test or set of tests used to compare the performance of alternative tools or techniques." A benchmark has preconditions. First, there must be a minimum level of maturity of the given research area. Second, it is desirable that diverse approaches exist. The authors defined seven requirements of successful benchmarks: accessibility, affordability, clarity, relevance, solvability, portability and scalability. Sim gives a more detailed description and examples in her Ph.D. thesis [18].

Nowadays, more and more papers introducing the evaluation of reverse engineering tools are published. These are needed because the number of reverse engineering tools is increasing and it is difficult to decide which of these tools is the most suitable to perform a given task.

Design patterns. Petterson et al. [13] summarized problems during the evaluation of accuracy in pattern detection. The goal was to make accuracy measurements more comparable. Six major problems were revealed: design patterns and variants, pattern instance type, exact and partial match, system size, precision and recall, and control set. A control set was "the set of correct pattern instances for a program system and design pattern." The

determination of the control sets is very difficult, therefore solutions from natural language parsers are considered. One good solution is *tree banks*. Tree banks could be adapted by establishing a large, manually validated pattern instances database. Another adaptable solution is the idea of pooling process: "The idea is that every system participating in the evaluation contributes a list of n top ranked documents, and that all documents appearing on one of these lists are submitted to manual relevance judgement." The process of constructing control sets has the following problems. They are not complete in most software systems, and on a real scale software system a single group is not able to determine a complete control set. They stated that *community effort is highly required* to make control sets for a set of applications.

Duplicated code. Bellon et al. [4] presented an experiment to evaluate and compare clone detectors. The experiment involved several researchers who applied their tools on carefully selected large C and Java programs. The comparison shed light on some facts that had been unknown before, so both the strengths and the weaknesses of the tools were discovered. Their benchmark gives a standard procedure for every new clone detector.

Rysselberghe et al. [16] compared different clone searching techniques (string, token and parse tree based). For the comparison, they developed reference implementations of the different techniques instead of using existing tools. During the evaluation, they used certain questions, some of which were corresponding to the criteria introduced in BEFRIEND. Such a question was e.g.: "How accurate are the results?". The different techniques were tested on 5 small and medium size projects by using the evaluating questions. In another article, they compared the reference implementations of the clone searching techniques on the basis of refactoring aspects [15]. These aspects were the following: suitable, relevance, confidence and focus.

Burd et al. [5] evaluated 5 clone searching tools on the university project GraphTool. They also faced the problem of how to link the instances. For this purpose, they used a simple overlap method. For evaluation, they used the well-known precision and recall values. In addition, they presented different statistics, some parts of which are even now supported by BEFRIEND (e.g. intersection, difference).

Coding rule violations. Wagner et al. [21] compared 3 Java bug searching tools on 1 university and 5 industrial projects. A 5-level severity scale, which can be integrated into BEFRIEND, served as the basis for comparison. Based on the scale, "Defects that lead to a crash of the application" is the most serious one, while "Defects that reduce the maintainability of the code" is the less serious one. The tools were compared not only with each other, but with reviews and tests as well. We note that both the reviews and the tests can be loaded into BEFRIEND by writing the appropriate plug-ins. In two other articles [20, 22] they also examined the bug searching tools.

Ayewah et al. [1] evaluated the FindBugs tool on three large scale projects, SUN JDK 1.6, Google and GlassFish. During the evaluation they applied the following categories in the case of JDK 1.6: Bad analysis, Trivial, Impact and Serious.

Rutar et al. [14] evaluated and compared five tools: FindBugs, JLint, PMD, Bandera and ESC/Java. The evaluation was carried out on 5 projects (Apache Tomcat 5.019, JBoss 3.2.3, Art of Illusion 1.7, Azureus 2.0.7 and Megamek 0.29). They observed that the rate of overlap among the tools was very low, the correlation among the hits of the tools was weak. This leads to the conclusion that rule violation searching tools are in a serious need of such an evaluating and comparing system as BEFRIEND.

A tool demonstration about BEFRIEND [11] was accepted by the 15th Working Conference on Reverse Engineering. It was only a short report of two pages, which was not enough to introduce BEFRIEND completely. This paper is a big extension of the tool demonstration: sibling connections are detailed, BEFRIEND is presented with concrete scenarios and some experimental results are also shown, etc.

6 Conclusion and future work

We have developed BEFRIEND from the benchmark for evaluating design pattern miner tools called DEEBEE. During the development of BEFRIEND, we were striving for full generalization: an arbitrary number of domains can be created, the domain evaluation aspects and the setting of instance siblings can be customized, etc. For uploading the results of different tools, the benchmark provides a plug-in mechanism. We already applied BEFRIEND for three reverse engineering domains: design pattern mining tools, code clone mining tools, and coding rule violation checking tools. The evaluation results stored in DEEBEE have been migrated to BEFRIEND, and in the code clones domain we have demonstrated the benchmark with further examples.

In the future, we would like to examine further reverse engineering domains, prepare the benchmark for these domains and deal with the possible shortcomings. There are also a number of possibilities for the further development of BEFRIEND. Without the sake of completeness, some of these are: instance grouping according to further aspects, e.g. in the case of design patterns, creational, structural and behavioural patterns [12], in the case of code clones, the Type I, II and III [4], while in the case of coding rule violations, e.g. General and Concurrency [14]. Creating further statistic views and querying instances on the basis of the statistic results.

Ayewah et al. [1] mentioned that "there is not significant agreement on standards for how to evaluate and categorize warnings." This statement can easily be extended to every domain. We often face this problem in the case of reverse engineering tools. In the long term, BEFRIEND could offer a solution to this problem, this is why we would like to see more and more

people using it. As a result, a standard evaluation method could be created.

This work is the first step to create a generally applicable benchmark that can help to evaluate and compare many kinds of reverse engineering tools. In the future, we will need the opinion and advice of reverse engineering tool developers in order for the benchmark to achieve this aim and satisfy all needs. BEFRIEND is freely available and public on the link http://www.inf.u-szeged.hu/befriend/

References

1 **Ayewah N, Pugh W, Morgenthaler J, Penix J, Zhou Y**, *Evaluating static analysis defect warnings on production software*, Paste '07: Proceedings of the 7th acm sigplan-sigsoft workshop on program analysis for software tools and engineering, ACM, 2007, 1–8, DOI 10.1145/1251535.1251536, (to appear in print).

2 **Balanyi Z, Ferenc R**, *Mining Design Patterns from C++ Source Code*, Proceedings of the 19th international conference on software maintenance (icsm 2003), IEEE Computer Society, September 2003, 305–314, DOI 10.1109/ICSM.2003.1235436, (to appear in print).

3 *The Bauhaus Homepage*, available at `http://www.bauhaus-stuttgart.de`.

4 **Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E**, *Comparison and Evaluation of Clone Detection Tools*, Ieee transactions on software engineering, September 2007, 577-591, DOI 10.1109/TSE.2007.70725.

5 **Burd E, Bailey J**, *Evaluating clone detection tools for use during preventative maintenance*, Proceedings of the 2th international workshop on source code analysis and manipulation (scam 2002), IEEE Computer Society, 2002, 36-43, DOI 10.1109/SCAM.2002.1134103, (to appear in print).

6 *The CCFinder Homepage*, available at `http://www.ccfinder.net/`.

7 *The Design Pattern Detection tool Homepage*, available at `http://java.uom.gr~nikos/pattern-detection.html`.

8 **Ferenc R, Gustafsson J, Müller L, Paakki J**, *Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa*, Acta Cybernetica, **15**, (2002), 669-682.

9 **Fowler M, Beck K, Brant J, Opdyke W, Roberts D**, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Pub Co, 1999, ISBN ISBN 0-201-48567-2.

10 **Fülöp L J, Ferenc R, Gyimóthy T**, *Towards a Benchmark for Evaluating Design Pattern Miner Tools*, Proceedings of the 12th european conference on software maintenance and reengineering (csmr 2008), IEEE Computer Society, April 2008, DOI 10.1109/CSMR.2008.4493309, (to appear in print).

11 **Fülöp L J, Hegedűs P, Ferenc R, Gyimóthy T**, *Towards a Benchmark for Evaluating Reverse Engineering Tools*, Tool demonstrations of the 15th working conference on reverse engineering (wcre 2008), October 2008, DOI 10.1109/WCRE.2008.18, (to appear in print).

12 **Gamma E, Helm R, Johnson R, Vlissides J**, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley Pub Co, 1995, ISBN ISBN 0-201-63361-2.

13 **Pettersson N, Löwe W, Nivre J**, *On evaluation of accuracy in pattern detection*, First international workshop on design pattern detection for reverse engineering (dpd4re'06), 2006 October, `http://cs.msi.vxu.se/papers/PLN2006a.pdf`.

14 **Rutar N, Almazan C B, Foster J S**, *A comparison of bug finding tools for java*, Issre '04: Proceedings of the 15th international symposium on software reliability engineering, IEEE Computer Society, 2004, 245–256, DOI 10.1109/ISSRE.2004.1, (to appear in print).

15 **Rysselberghe F V, Demeyer S**, *Evaluating clone detection techniques*, Proceedings of the international workshop on evolution of large scale industrial software applications, 2003., 2003, `citeseer.ist.psu.edu/vanrysselberghe03evaluating.html`.

16 ———, *Evaluating clone detection techniques from a refactoring perspective*, 19th international conference on automated software engineering (ase'04), IEEE Computer Society, 2004, 336–339.

17 **Sim S E, Easterbrook S, Holt R C**, *Using Benchmarking to Advance Research: A Challenge to Software Engineering*, Proceedings of the twenty-fifth international conference on software engineering (icse'03), IEEE Computer Society, 2003May, 74–83, DOI 10.1109/ICSE.2003.1201189, (to appear in print).

18 **Sim S E**, *A theory of benchmarking with applications to software reverse engineering*, Ph.D. Thesis, 2003.

19 *The Simian Homepage*, available at `http://www.redhillconsulting.com.au/products/simian/`.

20 **Wagner S, Deissenboeck F, Aichner M, Wimmer J, Schwalb M**, *An evaluation of two bug pattern tools for java*, Proceedings of the 1st ieee international conference on software testing, verification and validation (icst 2008), ACM, 2008, 1–8, DOI 10.1109/ICST.2008.63, (to appear in print).

21 **Wagner S, Jurjens J, Koller C, Trischberger P**, *Comparing bug finding tools with reviews and tests*, In proceedings of 17th international conference on testing of communicating systems (testcom'05), Springer, 2005, 40–55, DOI 10.1007/11430230-4, (to appear in print).

22 **Wagner S**, *A literature survey of the quality economics of defect-detection techniques*, Isese '06: Proceedings of the 2006 acm/ieee international symposium on empirical software engineering, ACM, 2006, 194–203, DOI 10.1145/1159733.1159763, (to appear in print).