Evaluating and Improving Reverse Engineering Tools

Summary of the Ph.D. Thesis

of

Lajos Jenő Fülöp

Supervisor:
Dr. Tibor Gyimóthy

Ph.D. School of Computer Science
Institute of Informatics
University of Szeged

Szeged
2011

Introduction

Maintaining legacy systems incur significant costs. Most of the time, re-engineering such a system is a better choice than rewriting it from scratch [9]. Re-engineering consists of two stages, namely reverse engineering information from the current system and, based on this information, (forward) engineering the system into a new form. Successful re-engineering demands a really reliable reverse engineering of the legacy system because any decision made during the forward engineering phase will be based on this information. It motivated us to develop a method which extends and improves one of our reverse engineering tools, and to develop benchmarks and perform experiments on evaluating reverse engineering tools.

In this study we deal with *design pattern miners*, *duplicated code detectors* and *rule violation checkers*. Design pattern mining tools help one to better understand the system and its components. Duplicated code detector tools discover risky copied code fragments that could carry the same bugs and make the maintenance of the system difficult. Rule violation checkers audit typical programmer errors in the source code. These tools provide important information about the legacy system, but their results may contain false positives.

We enhanced our design pattern miner tool, which is a component of the Columbus framework [6]. We used machine learning methods to further refine the pattern mining by marking the pattern candidates returned by the matching algorithm as either true or false [34]. We also compared three design pattern miner tools (Columbus, Maisa and CrocoPat) from three aspects, namely differences between the hits, their speed and memory requirements [35]. Furthermore, we present experiments performed on a newly developed benchmark (DEEBEE) for evaluating and comparing design pattern miner tools. With the help of this benchmark, the accuracy of two design pattern miner tools were evaluated on reference implementations of design patterns and on two software systems [36, 37, 38]. We also introduced DPDX, a common exchange format for design pattern miner tools. Here, we propose a well-defined and extendible metamodel that addresses a number of limitations of current tools. Then, the proposed metamodel is implemented in an XML-based language [39, 40]. After, we introduced a new version of the DEEBEE system, called BEFRIEND, which has become more useful since we generalized the evaluation aspects and the type of the evaluated tools. With BEFRIEND, the results of reverse engineering tools that recognize arbitrary aspects of source code can be subjectively evaluated and compared with each other [41, 42].
Now I will state five main results in the thesis. These are:

1. **The improvement of an existing design pattern miner tool.**
2. **Performance evaluation of design pattern miner tools.**
3. **Validation of design pattern miner tools.**
4. **Common exchange format of design pattern miner tools**
5. **Validation of reverse engineering tools.**

In the following sections I will briefly present these results and emphasize my own contributions at the end of each section.

1 Improvement of an existing design pattern miner tool

The problem with the more common approaches to pattern recognition (based on pattern matching) is that they are inherently too lax in the sense that they produce many false results, in which some code fragments are identified as pattern candidates that share only the *structure* of the pattern description. Here we present a machine learning method and results of experiments that tell us how to improve the results of design pattern miner tools so as to decide whether they are correct or not. We applied the design pattern mining approach of our Columbus framework [4]. We carried out experiments on StarWriter (containing over 6,000 classes), the text editor of the StarOffice suite [28]. Using the pattern-matching algorithm of Columbus we first found several pattern candidates that were further filtered using machine learning methods to provide more accurate results.

The learning process

In the following we will give an overview of the concrete steps of the learning process we developed.

1. *Predictor value calculation.* In the original pattern mining process [4], Columbus analyzes the source code and creates an Abstract Semantic Graph (ASG) representation. Afterwards, the design pattern miner component of Columbus (CAN2Dpm) finds design pattern candidates that conform to the actual DPML (Design Pattern Markup Language) file, which describes the structure of the pattern looked for. Each design pattern has features (predictors) that are not related to its structural description. We retrieve this kind of information from the ASG - saving them to a CSV file (predictor table) - and use them as input for the learning system.

2. *Manual inspection.* Here, we examine the source code manually to decide whether the design pattern candidates are false candidates or not. Then we extend the predictor table file with a new column containing the results of the manual inspection.

3. *Machine learning.* We perform the training of the machine learning systems(C4.5 [25] and Backpropagation [7]). The outputs of these are models which contain the acquired knowledge.

4. *Integration.* Lastly, we integrate the results of machine learning into Columbus.

Figure 1 graphically describes this process. The original elements of the mining process of Columbus are denoted by straight lines and empty boxes, while the new parts introduced by the current study are denoted by dashed lines and filled boxes.

Experiments

We performed experiments with two design patterns, Adapter Object and Strategy. To assess the accuracy of the learning process we applied the method of three-fold cross-validation, which means that we divided the predictor table file into three equal parts and performed the learning process three times. We defined the learning accuracy score in each case as the ratio of the number of correct decisions of the learning systems (compared to the manual classification) over the total number of candidates. We calculated the average and standard deviation (shown in parentheses) using the three testing results and got the scores shown in Table 1.

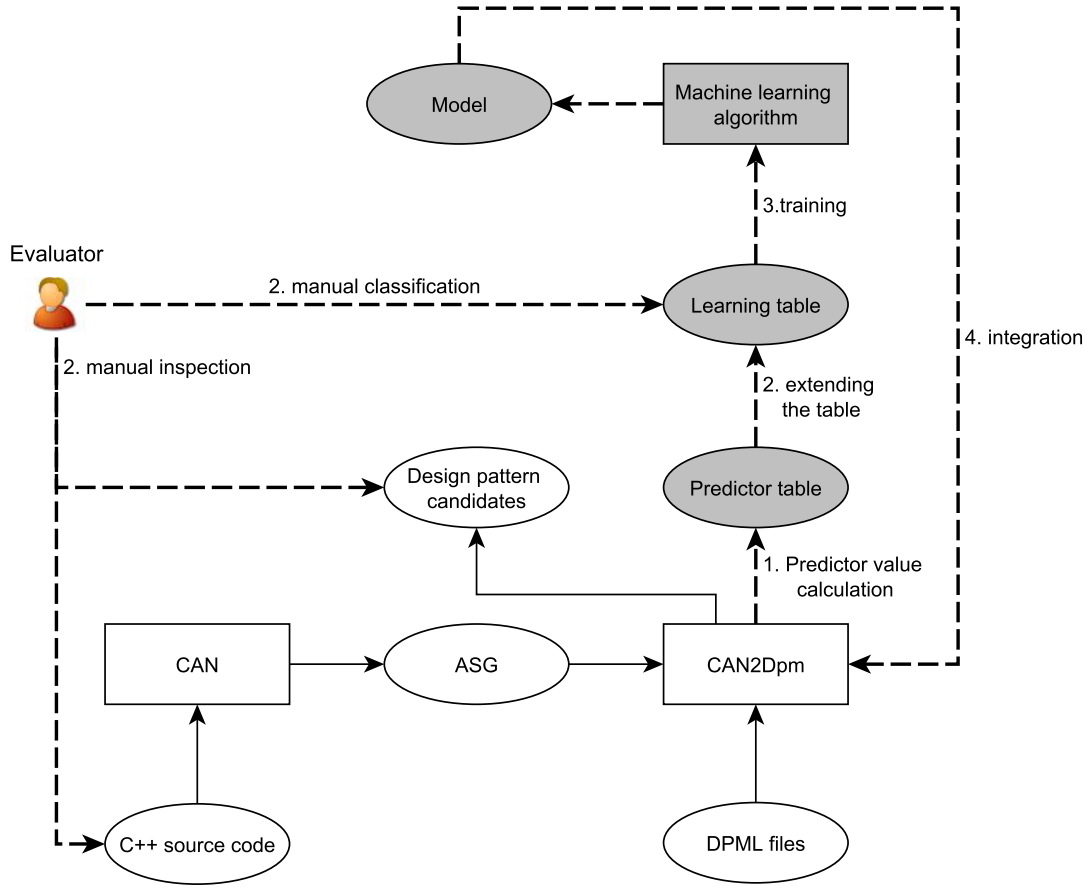

Figure 1: The learning process

Design Pattern	Decision Tree	Neural network
Adapter Object	66.70% (21.79%)	66.70% (23.22%)
Strategy	90.47% (4.13 %)	95.24% (4.12 %)

Table 1: Average accuracy with standard deviation based on three-fold cross validation

Our goal was to filter out false candidates from the results provided by our structure-based pattern miner algorithm [4]. In our experiments we achieved learning accuracy scores of 67–95% and with the model obtained we were able to filter out 86% of the false candidates of the Adapter Object design pattern, and 94% of the false candidates of the Strategy pattern.

Own contribution

The author performed the experiments with the Strategy design pattern and manually tagged the results of the design pattern mining tool in the case of the Strategy design pattern. The results of this thesis are published in [34].

2 Performance evaluation of design pattern miner tools

In this study we compare three design pattern miner tools, namely Columbus, Maisa and CrocoPat. We chose these tools because it is possible to prepare a common input for them with Columbus. Our previous work enabled us to provide the input for Maisa [12], while in the case of CrocoPat we created a new plug-in for Columbus which is able to prepare the appropriate input. We illustrate this process in Figure 2.

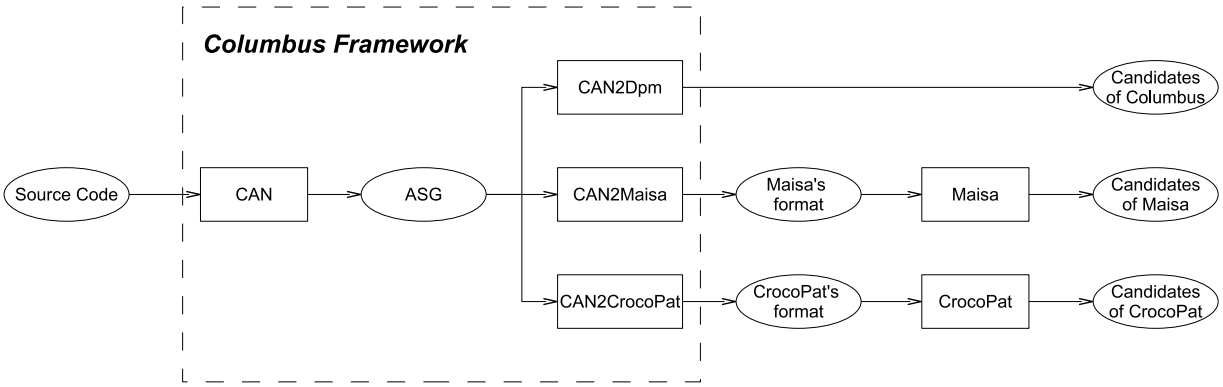

Figure 2: Common framework

Approach

Now we present a comparative approach of the given design pattern mining tools with the following viewpoints.

- *Differences between the design pattern candidates found.* Sometimes, tools report the same design pattern candidate differently. These differences might be due to several reasons. The tools might use different techniques to define a design pattern and the representation of the results might not be the same. We will try to discover the possible reasons for the differences experimentally.
- *Speed.* Speed is measured by the amount of the time taken by the tool to mine the given design pattern.
- *Memory usage.* We measured the maximal memory required for the design pattern mining task.

Experiments

We made a comparison on four open source small-to-huge systems (DC++, WinMerge, Jikes and Mozilla) to make the benchmark results independent of system characteristics like size, complexity and application domain. All the tests were run on the same computer, so the measured values were independent of the hardware and hence the results are comparable. Our test computer had a 3 GHz Intel Xeon processor with 3 GB memory.

Differences between the design pattern candidates found

In essence, the design pattern candidates found would be the same in most of the cases if we could disregard the following common causes of the differences.

Different definitions of design patterns. We found that there were some specific reasons why the tools discovered different pattern candidates. The main one was that in some cases a design pattern description overlooked a participant as in the case of the Builder pattern in Maisa. Here the pattern definition did not contain the Director participant, hence the candidates discovered by Maisa were not the same as those found by the other two. Such differences could be intentional and ad-hoc as well, therefore it would be difficult to standardize and to eliminate them.

Precision of pattern descriptions. Another difference was how precise the pattern descriptions actually were. For example, in the case of Jikes the differences in the total number of Adapter Class candidates found were due to the fact that CrocoPat and Columbus defined the Target as abstract while Maisa did not.

Differences in algorithms. We found differences in the design pattern miner algorithms as well. For example, Columbus and Maisa counted the repeated candidates with certain classes in common only once, but CrocoPat counted each occurrence. Such differences should be handled and standardized in a fair comparison of candidate correctness.

Speed.

Overall, we concluded that the best tool from a speed perspective is CrocoPat, but in some cases Columbus was faster. Columbus can be applied in the case of small- or medium-sized systems and in the case of complex design patterns like Visitor. As for CrocoPat, it can used in the case of a larger system or in the case of a simpler design pattern like Template Method.

Memory usage.

The results here showed that the memory usage strongly depends on the size of the projects analyzed and it is independent of the given design patterns. In the case of Columbus, the required memory was very large compared to the other two. This is due to the fact that Columbus is a general reverse engineering framework and design pattern detection is just one of its many features. For this reason it uses an ASG representation, which contains all the information about the source code. Note that with CrocoPat and Maisa the required memory was smaller because their inputs only contained information about the source code necessary for pattern detection. We conclude that, in the terms of memory requirements, Maisa's performance was the best.

Own contribution

The author developed the Columbus-CrocoPat exporter and integrated it into the Columbus framework. He also defined several design patterns in the representation language (RML) of CrocoPat. Furthermore, the author performed the experiments presented in this thesis. The author also participated in finding a concrete definition for the comparison-and-evaluation approach.

3 Validation of design pattern miner tools

We developed a publicly available benchmark called DEEBEE (DEsign pattern Evaluation BEnchmark Environment) for evaluating and comparing design pattern miner tools. Our benchmark is general, being language, software, tool and pattern independent. With this benchmark the accuracy (precision and recall) of the tools can be validated by anyone.

Benchmark

Figure 3 shows an overview of the benchmark. First, design pattern miner tools and developers discover design patterns from the source code. Afterwards, design pattern miner tools generate their results in a tool-specific format, which have to be converted into the input format of DEEBEE (which is a CSV file).

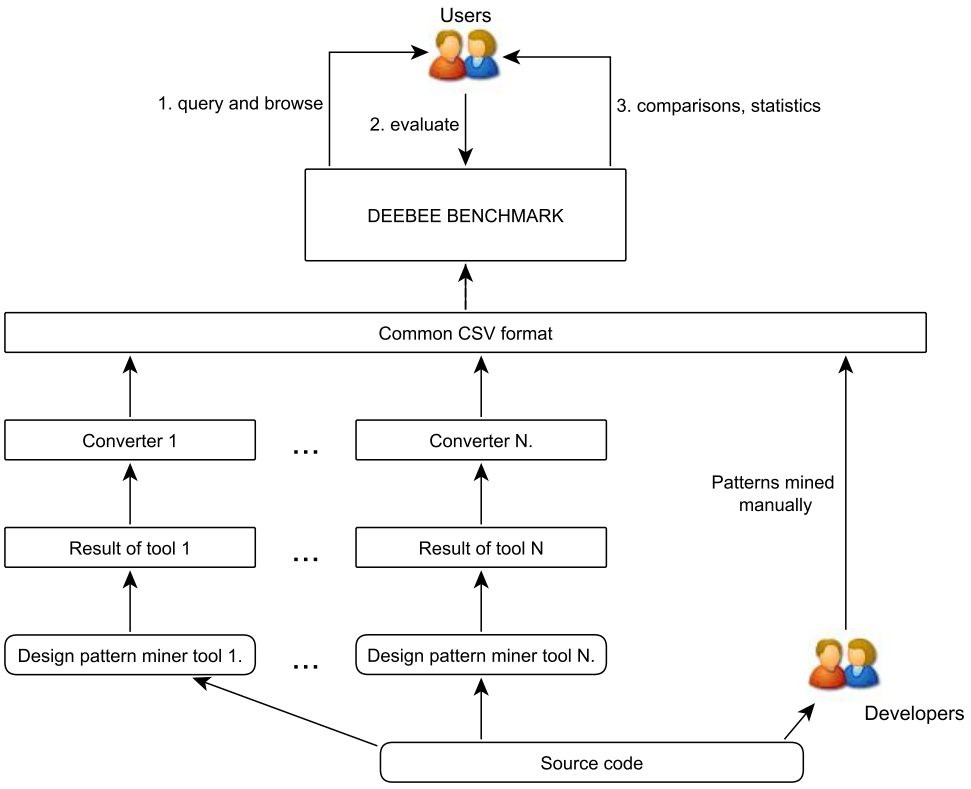

Figure 3: Overview of DEEBEE

Developers manually discover design pattern candidates and then store their results in the DEEBEE CSV file format. In the next step, the candidates (CSV files) are uploaded into the benchmark. After, the candidates can be queried and browsed, evaluated and compared via the online interface of DEEBEE. Furthermore, the benchmark is able to automatically generate statistics (e.g. precision and recall) based on the evaluations given by the users. These functionalities significantly ease the evaluation and comparison process of design pattern miner tools.

As shown in the second thesis, the results of the different design pattern miner tools may differ for several reasons. In DEEBEE we propose a method to handle this. We labelled the same but differently

reported pattern candidates as *siblings*. The identification of siblings is based on the *fundamental participants* of design pattern candidates. For example, in the case of the State pattern [13] the fundamental participant plays the role of the State class.

The benchmark contains 1,274 design pattern candidates from three C++ software systems (Mozilla [21], NotePad++ [22] and FormulaManager [29]), three Java software systems (JHotDraw [17], JRefactory [18] and JUnit [19]) and C++ reference implementations of design patterns. The uploaded design pattern candidates are recovered by three design pattern miner tools: Columbus (C++) [4], Maisa (C++) [23] and Design Pattern Detection Tool (Java) [30].

Experiments

To compare different tools, *reference implementations* of design patterns based on a book by Gamma et. al. [13] were created by us. With these reference implementations the basic capabilities of C++ pattern miner tools can be evaluated and compared. Since the reference implementations contain disjoint implementations of the design patterns in an artificial context, we developed a program called FormulaManager where each design pattern occurs in a real context at least once. In addition, pattern instances from NotePad++ recovered by professional software developers were added to the benchmark.

We evaluated and compared two design pattern miner tools, namely *Maisa* and *Columbus*, with the help of DEEBEE. The tools were evaluated on *reference implementations*, on *FormulaManager*, and on *NotePad++*. The results are shown in Table 2.

System	Reference impl.		NotePad++		FormulaManager	
Tool	Columbus	Maisa	Columbus	Maisa	Columbus	Maisa
Precision	100%	80.00%	62.50%	16.67%	52.27%	80%
Recall	58.33%	33.33%	29.41%	11.76%	71.88%	25%

Table 2: Results

Own contribution

The author developed the benchmark (except the instance view). He also defined and implemented the uploading format of the benchmark including the sibling and grouping mechanism. The author performed the experiments with Maisa and Columbus, and uploaded their results into the benchmark. He also participated in designing the architecture of the benchmark, in determining the evaluation aspects, in manually tagging the results of the tools and in designing its use cases.

4 Common exchange format of design pattern miner tools

We propose to address the limitations of output formats of current design pattern detector (DPD) tools by introducing a common exchange format for them, called DPDX, based on a well-defined and extendible metamodel. This format should aid the comparison [35], fusion [20], visualization [10], and validation [37] of the outputs of different DPD tools.

Requirements

We define the following core requirements that the common exchange format must fulfil to address the limitations of current DPD tools outputs and serve as the basis for a federation of tools:

1. **Specification.** The exchange format must be specified formally to allow DPD tool developers to implement appropriate generators, parsers, and/or converters.
2. **Reproducibility.** The tool and the program to be analyzed must be explicitly reported to allow researchers to reproduce the results.
3. **Justification.** The format must include explanations and scores expressing confidence in the results produced by the tool.
4. **Completeness.** The format must be able to represent program constituents at every level of role granularity described in design pattern literature.
5. **Identification of role players.** Each program constituent playing a role in a design motif must be unambiguously identified.
6. **Identification of candidates.** Each candidate must be unambiguously identified and reported only once.
7. **Comparability.** The format must allow one to report the motif definitions assumed by a tool and the applied analysis methods to allow other tool users to compare results.
8. **Language-independence.** (optional) The common exchange format should abstract language specific concepts so that it can be used to report candidates identified in programs written in arbitrary imperative programming languages (including object-oriented languages).
9. **Standard-compliance.** (optional) The specification should be consistent with existing standards so that it can be easily adapted, maintained, and evolved.

Metamodels

Three meta-models together specify the DPDX format: the meta-model of design pattern schemata, the meta-model of program element identifiers and the meta-model of DPD results. The schema meta-model allows the tools to report the schema of the patterns they search for; the program element meta-model allows the tools for identifying the program elements of the source code playing some role in the pattern instance; and the result meta-model describes the detected pattern candidates themselves.

Figure 4 shows how these models are related. Here the results are instances of the result meta-model. Their main part is the mapping of roles and relations (from schemata) to the program elements (in the program element model). Candidates are targets of mappings like this. Note that candidates may overlap; that is, program elements can play a role in different pattern schemata, as illustrated by the overlap of one of the Singleton candidates with one of the Decorator candidates.

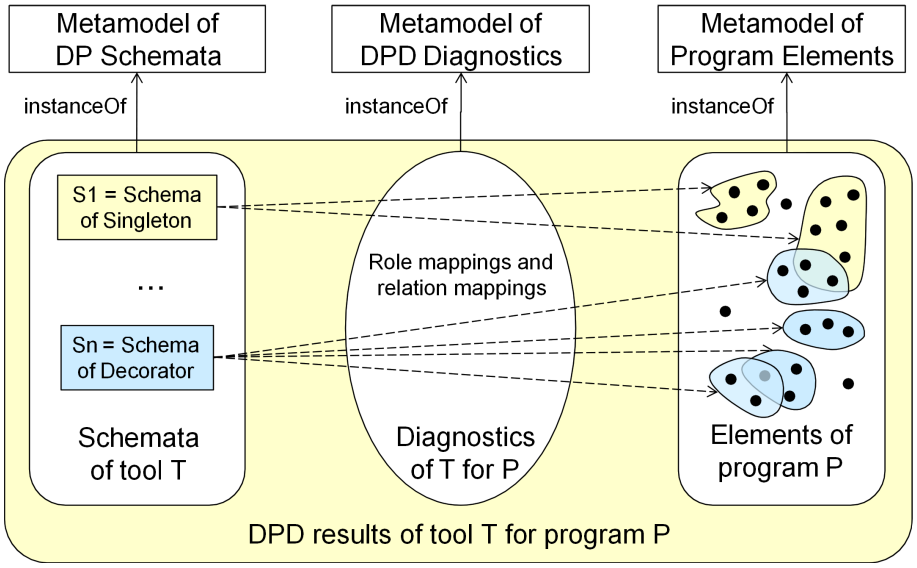

Figure 4: Relation between schemata, diagnostics and instances

Implementation

For long-term maintainability, the implementations of the meta-models should rely as much as possible on emerging or de-facto standards. Therefore we shall base our common exchange output format on XML. The implementation of DPDX consists of the realization of the three meta-models. To keep the implementation simple, we have adhered as much as possible to the following general principles for mapping meta-models to XML: (1) classes of the meta-models are mapped to XML tags; (2) attributes of the meta-model elements are mapped to attributes of the XML elements; (3) aggregation between the elements of the meta-models is represented by the parent-children nesting technique of XML; (4) an element that can be referred to by another element has an 'id' attribute, and the element that would like to refer to this element has an attribute to refer it; (5) an association with target cardinality greater than 1 is represented by a group element included with individual referencing elements.

Own contribution

The author developed the initial versions of the schema metamodel implementation and described the Maisa tool. He also participated in the substantial improvement and finalization of the initial ideas (concepts, metamodel, implementation) in their eventual form.

5 Validation of reverse engineering tools

We further developed the DEEBEE system to help make it more suitable by generalizing the evaluating aspects and the data to be evaluated and compared. The new system is called **BEFRIEND** (BEnchmark For Reverse engInEering tools workiNg on source coDe). BEFRIEND largely differs from its predecessor (DEEBEE) in five aspects.

1. **Domains.** DEEBEE supports the evaluation and comparison of the results of design pattern miner tools. With BEFRIEND, the results of reverse engineering tools from different domains recognizing the arbitrary characteristics of source code can be evaluated and compared with each other. Such tools include design pattern detectors, duplicated code detectors and coding rule violation checkers. In BEFRIEND, a user is able to set the active domain, and every further action (e.g. listing candidates) on the user interface will be in this active domain.

2. **Evaluation aspects.** BEFRIEND allows the adding and deleting of the evaluating aspects of the results in an arbitrary way, while DEEBEE has fixed evaluation aspects. On the basis of this, the uploaded candidates can be evaluated. In one evaluation criterion, one question has to be given to which an arbitrary number of answers can be defined. For each answer a percentage ratio should be set to indicate to what extent the given question has been answered. Based on the replies by users, the benchmark can calculate different statistics using this ratio.

3. **User interface.** The user interface of DEEBEE was also improved by us during the development of BEFRIEND. For example, in DEEBEE, the instance view just displays one code fragment but BEFRIEND displays two code fragments (see Figure 5). For instance, it eases the comparison of copied code fragments detected by duplicated code detector tools.

4. **Grouping mechanism.** BEFRIEND generalizes the definition of grouping mechanism (sibling relationships) to tackle the problems of other domains, not just design pattern mining, e.g. for duplicated code detectors where fundamental participants cannot be used as a basis for grouping the same results, unlike DEEBEE. Three things determine the existence of the sibling relation between two candidates in BEFRIEND. These are the matching of their source code positions, the minimal number of matching participants, and domain dependent name matching. The settings of sibling relations can be aligned in BEFRIEND for each available domain separately.

5. **Plug-in oriented architecture.** DEEBEE has a special CSV format for uploading, and a design pattern miner tool has to convert his output into this format. BEFRIEND allows the uploading of files in different formats by introducing a plug-in oriented architecture. In this way, it permits the uploading of the results of a new tool by implementing the appropriate plug-in.

Experiments

We applied BEFRIEND to three reverse engineering domains, namely design pattern mining tools, code clone mining tools, and coding rule violation checking tools. In the code clones domain we performed some experiments with the benchmark. Five *duplicated code* finder tools were assessed on two different open source projects called *JUnit* and *NotePad++*. For the evaluation, three evaluation criteria were used. The *Correctness* criterion is used to decide to what extent a code clone group comprises cloned

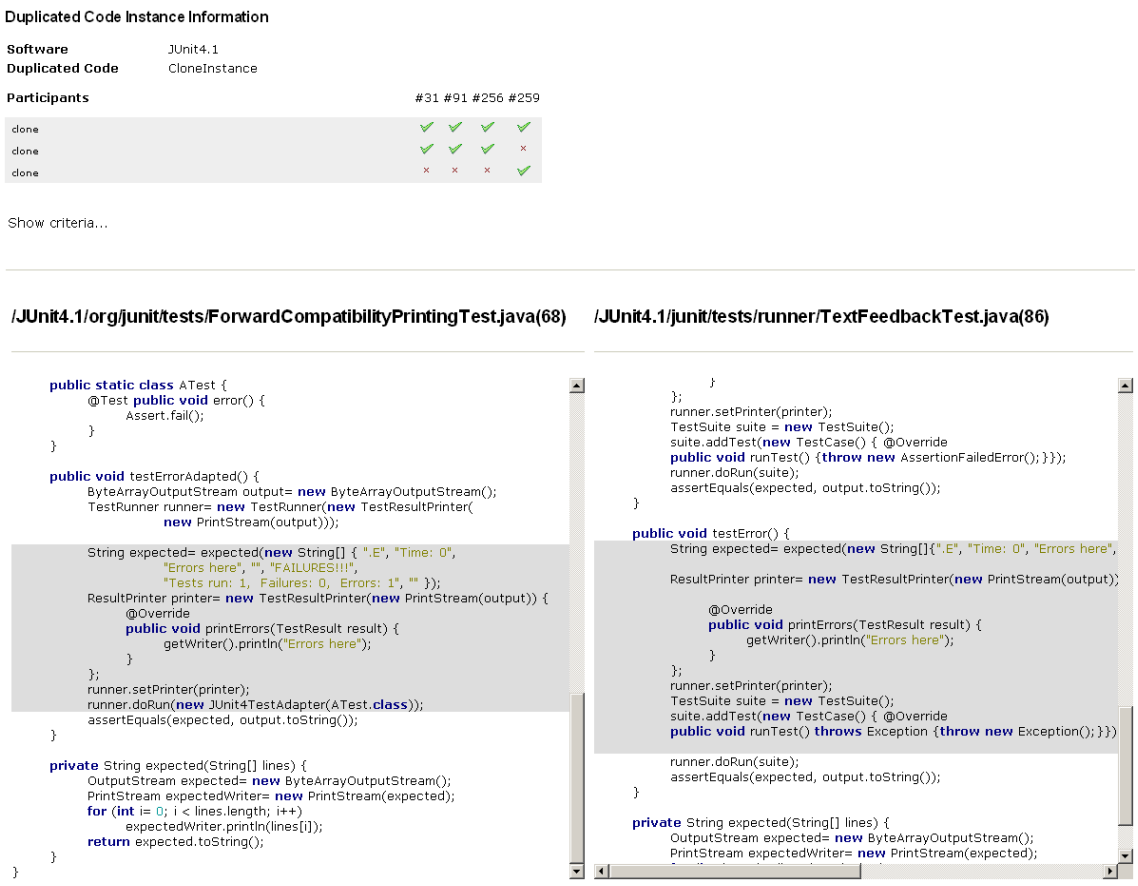

Figure 5: Group instance view

code fragments; precision and recall scores are calculated based on votes for this criterion. The second criterion is *Procedure abstraction* with the related question *'Is it worth substituting the duplicated code fragments with a new function and function calls?'*. The third criterion is *Gain* with the related question *'How much is the estimated gain of refactoring into functions?'* The results for JUnit are shown in Table 3.

Tool	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%

Table 3: Results on JUnit

Own contribution

The author adopted and generalized the theory of sibling relations and provided the corresponding implementation. He also participated in defining the terminology, manually evaluating the candidates using the benchmark and in the presentation of the benchmark's architecture.

Conclusions

The main contributions of this work are summarized as follows. First, we employed machine learning methods to further refine the results of our design pattern miner tool. In this study, we showed that a machine learning-based method can be successfully applied for filtering out false positives of a reverse engineering tool. Previously, several interesting approaches were published [32, 33, 16, 1] to improve the results of design pattern miner tools, but no one has tried to improve the results as we do here.

In our second study, we evaluated and compared three design pattern miner tools in a set of experiments. During these experiments, the common differences of tool results were collected, and it was learned which tool should be used in certain circumstances in terms of speed and memory consumption. Based on the experiences of the second study, we developed DEEBEE, a benchmark for evaluating and comparing design pattern miner tools, and performed some experiments using it. The benchmark is general, being language, software, tool and pattern independent. With this benchmark the accuracy of the tools can be validated by anyone. Previously, several studies were published [24, 15, 11, 2, 14] about the evaluation and comparison of design pattern miner tools and about a review, but such a benchmark like DEEBEE did not exist (e.g. with an automatic grouping).

We also introduced an XML-based output format (DPDX) for design pattern miner tools. The proposed format is based on a well-defined and extendible metamodel that addresses the limitations of formats of design pattern miner tools. This format should aid the comparison [35], fusion [20], visualization [10], and validation [37] of the outputs of different design pattern miner tools.

Lastly, we developed BEFRIEND, a benchmark that can be used for evaluating and comparing reverse engineering tools. We applied BEFRIEND to three reverse engineering domains, namely design pattern mining tools, duplicated code detector tools, and coding rule violation checking tools. In the duplicated code domain we performed experiments with the benchmark using five *duplicated code* finder tools. Several papers were published [5, 27, 8, 31, 3, 26] about the evaluation and comparison of different kind of reverse engineering tools, but a general benchmark like BEFRIEND did not exist previously.

Table 4 summarizes which publications cover which results of the thesis.

$\mathcal{N}o.$	[34]	[35]	[36]	[37]	[38]	[39]	[40]	[41]	[42]
1.	•								
2.		•							
3.			•	•	•				
4.						•	•		
5.								•	•

Table 4: The relation between the thesis topics and the corresponding publications.

Acknowledgements

First, I would like to thank my supervisor Dr. Tibor Gyimóthy who helped me by providing useful ideas, comments and interesting research directions. I would like to thank my article co-author and mentor, Dr. Rudolf Ferenc, for guiding my studies and teaching me a lot of indispensable things about research. Without his valuable advice and hints I would never have acquired the research–oriented attitude that I have. My many thanks also go to my colleagues and article co-authors, namely Dr. Árpád Beszédes, Tibor Bakota, Dr. István Siket, Dr. Judit Jász, Péter Siket, Péter Hegedűs, Dr. Lajos Schrettner, Dr. Tamás Gergely, Dr. László Vidács, György Hegedűs, Dr. Günter Kniesel, Alexander Binun, Dr. Alexander Chatzigeorgiou, Dr. Yann-Gaël Guéhéneuc, Dr. Nikolaos Tsantalis, Gabriella Kakuja-Tóth, Hunor Demeter, Csaba Nagy, Ferenc Fischer, Árpád Ilia, Ádám Zoltán Végh, Róbert Rácz, Lóránt Farkas, Fedor Szokody, Zoltán Sógor, Gábor Lóki, János Lele, Tamás Gyovai, Tibor Horváth and János Pánczél. I would also like to thank the anonymous reviewers of my papers for their useful comments and suggestions. And I would like to express my thanks to David P. Curley for reviewing and correcting my work from a linguistic point of view. I would like to thank my mother for her continuous support and encouragement. Last, but not least, my heartfelt thanks goes to my wife Márti for providing a vital, affectionate and supportative background during the time spent writing this work.

Lajos Jenő Fülöp, 2011

References

[1] Giuliano Antoniol, Roberto Fiutem, and L. Cristoforetti. Using Metrics to Identify Design Patterns in Object-Oriented Software. In *Proceedings of the Fifth International Symposium on Software Metrics (METRICS98)*, pages 23–34. IEEE Computer Society, November 1998.

[2] Francesca Arcelli, Stefano Masiero, Claudia Raibulet, and Francesco Tisato. A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition. In *Proceedings of the 15th Australian Software Engineering Conference (ASWEC'05)*, pages 677–691. IEEE Computer Society, February 2005.

[3] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[4] Zsolt Balanyi and Rudolf Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, September 2003.

[5] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. In *IEEE Transactions on Software Engineering*, Volume 33, pages 577–591, September 2007.

[6] Árpád Beszédes, Rudolf Ferenc, and Tibor Gyimóthy. Columbus: A Reverse Engineering Approach. In *Proceedings of the 13th IEEE Workshop on Software Technology and Engineering Practice (STEP 2005)*, pages 60–69. IEEE Computer Society, September 2005.

[7] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[8] Elizabeth Burd and John Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Proceedings of the 2th International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 36–43. IEEE Computer Society, 2002.

[9] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Object-Oriented Reengineering Patterns. Square Bracket Associates, 2008.

[10] Jing Dong, Sheng Yang, and Kang Zhang. Visualizing Design Patterns in Their Applications and Compositions. *IEEE Trans. Softw. Eng.*, 33:433–453, July 2007.

[11] Jing Dong, Yajing Zhao, and Tu Peng. A Review of Design Pattern Mining Techniques. *the International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, pages 823–855, 2008.

[12] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybernetica*, 15:669–682, 2002.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[14] Yann-Gaël Guéhéneuc. P-MARt: Pattern-like Micro Architecture Repository. `http://www-etud.iro.umontreal.ca/~ptidej/yann-gael/Work/Publications/Documents/EuroPLoP07PRa.doc.pdf` .

[15] Yann-Gaël Guéhéneuc, Kim Mens, and Roel Wuyts. A Comparative Framework for Design Recovery Tools. In *Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 123–134. IEEE Computer Society, March 2006.

[16] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting Design Patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 172–181. IEEE Computer Society, 2004.

[17] The JHotDraw Homepage.
`http:/www.jhotdraw.org`.

[18] The JRefactory Homepage.
`http:/jrefactory.sourceforge.net/`.

[19] The JUnit Homepage.
`http:/www.junit.org`.

[20] Günter Kniesel and Alexander Binun. Standing on the Shoulders of Giants – A Data Fusion Approach to Design Pattern Detection. In *17th IEEE International Conference on Program Comprehension (ICPC'09)*. IEEE Computer Society, 2009.

[21] The Mozilla Homepage.
`http://www.mozilla.org`.

[22] The NotePad++ Homepage.
`http://notepad-plus.sourceforge.net/`.

[23] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A.I. Verkamo. Software Metrics by Architectural Pattern Mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pages 325–332, 2000.

[24] Niklas Pettersson, Welf Löwe, and Joakim Nivre. On Evaluation of Accuracy in Pattern Detection. In *First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE'06)*, October 2006.

[25] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[26] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256. IEEE Computer Society, 2004.

[27] Filip Van Rysselberghe and Serge Demeyer. Evaluating Clone Detection Techniques from a Refactoring Perspective. In *19th International Conference on Automated Software Engineering (ASE'04)*, pages 336–339. IEEE Computer Society, 2004.

[28] The StarOffice Homepage.
`http://www.sun.com/software/star`.

[29] The source code of FormulaManager.
`http://www.sed.hu/src/FormulaManager/`.

[30] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design Pattern Detection Using Similarity Scoring. In *IEEE Transactions on Software Engineering*, Volume 32, pages 896–909, Nov 2006.

[31] Stefan Wagner, Jan Jurjens, Claudia Koller, and Peter Trischberger. Comparing Bug Finding Tools with Reviews and Tests. In *Proceedings of 17th International Conference on Testing of Communicating Systems (TestCom'05)*, pages 40–55. Springer, 2005.

[32] Lothar Wendehals. Improving Design Pattern Instance Recognition by Dynamic Analysis. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, May 2003.

[33] Lothar Wendehals. Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams. In *Proceedings of the 6th Workshop Software Reengineering (WSR2004)*, pages 63–64, May 2004.

Listed publications

[34] Rudolf Ferenc, Árpád Beszédes, Lajos Fülöp, and János Lele. Design Pattern Mining Enhanced by Machine Learning. In *Proceedings of the 21th International Conference on Software Maintenance (ICSM 2005)*, pages 295–304. IEEE Computer Society, September 2005.

[35] Lajos Jenő Fülöp, Tamás Gyovai, and Rudolf Ferenc. Evaluating C++ Design Pattern Miner Tools. In *Proceedings of the 6th International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 127–136. IEEE Computer Society, September 2006.

[36] Lajos Jenő Fülöp, Árpád Ilia, Ádám Zoltán Végh, and Rudolf Ferenc. Comparing and Evaluating Design Pattern Miner Tools. In *Proceedings of the 10th Symposium on Programming Languages and Software Tools (SPLST 2007)*, pages 372–386. Eötvös Loránd University, Faculty of Informatics, June 2007.

[37] Lajos Jenő Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. Towards a Benchmark for Evaluating Design Pattern Miner Tools. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pages 143–152. IEEE Computer Society, April 2008.

[38] Lajos Jenő Fülöp, Árpád Ilia, Ádám Zoltán Végh, Péter Hegedűs, and Rudolf Ferenc. Comparing and Evaluating Design Pattern Miner Tools. *Journal of ANNALES Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 31:167–184, 2009. Department of Computer Algebra, Eötvös Loránd University.

[39] Günter Kniesel, Alexander Binun, Péter Hegedűs, Lajos Jenő Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhéneuc, and Nikolaos Tsantalis. DPDX – A Common Exchange Format for Design Pattern Detection Tools. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pages 232–235. IEEE Computer Society, March 2010.

[40] Günter Kniesel, Alexander Binun, Péter Hegedűs, Lajos Jenő Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhéneuc, and Nikolaos Tsantalis. A common exchange format for design pattern detection tools. Technical report IAI-TR-2009-03, ISSN 0944-8535, CS Department III, University of Bonn, Germany, October 2009.

[41] Lajos Jenő Fülöp, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. Towards a Benchmark for Evaluating Reverse Engineering Tools. In *Tool Demonstrations of the 15th Working Conference on Reverse Engineering (WCRE 2008)*, pages 335–336. IEEE Computer Society, October 2008.

[42] Lajos Jenő Fülöp, Péter Hegedűs, and Rudolf Ferenc. BEFRIEND - a Benchmark for Evaluating Reverse Engineering Tools. *Journal of Periodica Polytechnica, Electrical Engineering*, 52/3-4:153–162, 2008. Budapest University of Technology and Economics.