Evaluating and Improving Reverse Engineering Tools

Lajos Jenő Fülöp

Department of Software Engineering
University of Szeged

Supervisor: Dr. Tibor Gyimóthy

June 2011
Szeged, Hungary

A thesis submitted for the degree of doctor of philosophy of the University of Szeged

University of Szeged
Ph.D. School in Computer Science

Preface

Developers tend to leave some important steps and actions (e.g. properly designing the system's architecture, code review and testing) out of the software development process, and use risky practices (e.g. the copy-paste technique) so that the software can be released as fast as possible. However, these practices may turn out to be critical from the viewpoint of maintainability of the software system. In such cases, a cost-effective solution might be to re-engineer the system.

Re-engineering consists of two stages, namely reverse-engineering information from the current system and, based on this information, forward-engineering the system to a new form. In this way, successful re-engineering significantly depends on the reverse engineering phase. Therefore, it is vital to guarantee correctness, and to improve the results of the reverse engineering step. Otherwise, the re-engineering of the software system could fail due to the bad results of reverse engineering.

The above issues motivated us to develop a method which extends and improves one of our reverse engineering tools, and to develop benchmarks and to perform experiments on evaluating and comparing reverse engineering tools.

Lajos Jenő Fülöp, 2011

"To discover new continents,
you must be willing to lose sight of the shore."
Brian Tracy

Acknowledgements

This quote above expresses well how a person might feel when he starts his research work. I have been very lucky during my journey to the *"new continent"* because several people helped me with their comments, ideas and suggestions and this improved not just my work, but also opened my mind and broadened my knowledge. I am really grateful to all those who helped my journey and I would not be where I am now without you all.

First, I would like to thank my supervisor Dr. Tibor Gyimóthy who helped me in my work 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

Contents

Preface iii

Acknowledgements v

List of Figures xii

List of Tables xiv

1 Introduction 1
- 1.1 Summary by chapters . . . 4
- 1.2 Summary by results . . . 6

2 Background 11
- 2.1 Reverse engineering . . . 11
- 2.2 Terminology . . . 14
- 2.3 Columbus framework . . . 16

I A proposed method for improving design pattern mining 19

3 Improvement of an existing design pattern miner tool 21
- 3.1 The learning process . . . 22
- 3.2 Predictors . . . 23
  - 3.2.1 Adapter object . . . 24
  - 3.2.2 Strategy . . . 25
- 3.3 Machine learning approaches used . . . 26
- 3.4 Results . . . 27
  - 3.4.1 Adapter object candidates investigation . . . 28
  - 3.4.2 Strategy candidates investigation . . . 29
  - 3.4.3 Learning efficiency . . . 30
- 3.5 Summary . . . 32

II Evaluation of design pattern miner tools 33

4 Performance evaluation of design pattern miner tools 35
4.1 Framework . . . 35
4.1.1 CrocoPat . . . 35
4.2 A comparative approach . . . 36
4.3 Results . . . 38
4.3.1 Discovered pattern candidates . . . 39
4.3.2 Pattern mining speed . . . 44
4.3.3 Memory requirements . . . 47
4.4 Summary . . . 48

5 Validation of design pattern miner tools 49
5.1 Benchmark . . . 50
5.1.1 Architecture . . . 50
5.1.2 Fundamental participants and siblings . . . 52
5.1.3 Upload file format. . . . 54
5.1.4 Benchmark contents . . . 55
5.2 Usage scenarios . . . 56
5.2.1 Browsing the database . . . 56
5.2.2 Evaluating and comparing tools . . . 61
5.2.3 Adding a new tool . . . 62
5.3 Experiments performed . . . 64
5.3.1 Reference implementations . . . 67
5.3.2 NotePad++ . . . 69
5.3.3 FormulaManager . . . 71
5.4 Evaluation of the benchmark . . . 73
5.5 Summary . . . 74

6 Common format for design pattern miner tools 75
6.1 Background . . . 76
6.1.1 Motivation . . . 76
6.1.2 Requirements . . . 76
6.1.3 State of the art . . . 77
6.2 DPDX concepts . . . 79
6.2.1 Specification . . . 79
6.2.2 Reproducibility . . . 80
6.2.3 Justification . . . 80
6.2.4 Completeness . . . 80
6.2.5 Identification of role players . . . 80
6.2.6 Language independence . . . 82
6.2.7 Identification of candidates . . . 82

6.2.8 Comparability . . . 85
6.3 DPDX meta-models . . . 85
6.3.1 Schema metamodel . . . 85
6.3.2 Program element metamodel . . . 87
6.3.3 Result metamodel . . . 88
6.4 DPDX implementation . . . 89
6.4.1 Implementation details . . . 90
6.4.2 Integration and visualization . . . 91
6.5 Summary . . . 91

III Evaluation of reverse engineering tools 93

7 Validation of reverse engineering tools 95
7.1 Background . . . 95
7.1.1 Sibling relation . . . 96
7.2 Use scenarios . . . 101
7.2.1 Setting up the database . . . 101
7.2.2 Data evaluation . . . 105
7.3 Experimental results . . . 110
7.4 Summary . . . 112

8 Conclusions 115

Appendices 119

Appendix A Related Work 119
A.1 Design pattern mining . . . 119
A.2 Improvement of design pattern mining . . . 121
A.3 Evaluation of reverse engineering tools . . . 122

Appendix B DPDX 127
B.1 Output formats of DPD tools . . . 127
B.2 DPDX attribute values . . . 131
B.3 DPDX implementation examples . . . 134

Appendix C Summary 139
C.1 Summary in English . . . 139
C.2. Summary in Hungarian . . . 142

Bibliography 145

List of Figures

2.1 The reengineering process. 12
2.2 Concept map of the most frequent terms of the thesis. 14
2.3 The State design pattern. 15
2.4 The Columbus framework 17

3.1 The learning process 23
3.2 The Adapter Object design pattern 24
3.3 The Strategy design pattern 25

4.1 Common framework 36
4.2 Factory Method pattern in RML 37

5.1 Architecture of Trac 50
5.2 Overview of DEEBEE 51
5.3 State example 53
5.4 CSV file for the T1-2 candidate of the previous example 55
5.5 Functionalities of the benchmark 57
5.6 Query view of DEEBEE 57
5.7 Results view of DEEBEE 58
5.8 Instance view with highlighted source code in DEEBEE 59
5.9 Instance view statistics in DEEBEE 60
5.10 Statistics view of DEEBEE 61
5.11 Comparison view of DEEBEE 63
5.12 Adding the results of a new tool to DEEBEE 64
5.13 Framework for design pattern mining 65
5.14 Reference implementation of Adapter Object 66
5.15 Visitor candidate mined by Maisa from reference implementations 68
5.16 Comparison of candidates in reference implementations 70

6.1 A federation of design pattern detection, visualization and assessment tools cooperating via the common exchange format. 76
6.2 Named and unnamed elements example 81
6.3 Illustration of candidates 84

6.4 Relation between schemata, diagnostics and instances 85
6.5 Metamodel of design pattern schemata 86
6.6 Sample of design pattern schemata 86
6.7 Metamodel of program element identifiers and optional source locations . 87
6.8 Representation of the invocation a.f(d,c) from the code example in Figure 6.2 88
6.9 Metamodel of the design pattern detection results 88
6.10 A decorator instance from the java.io package of the JDK with subclasses implemented by us (OutputStreamWriter and CharCountBufferWriter) . . 89
6.11 Sample of the design pattern detection results 89

7.1 Example code for contain and overlap functions 97
7.2 Problem of the non-transitivity feature of the sibling relation 101
7.3 Creating a new domain 102
7.4 Correctness criteria 103
7.5 Uploading data into BEFRIEND 104
7.6 Sibling settings 105
7.7 Query view 106
7.8 Results view 106
7.9 Group instance view 107
7.10 Bauhaus correctness statistics 109
7.11 Comparison view 110

B.1 Output format of SPQR 127
B.2 Output format of Fujaba 128
B.3 Output format of Maisa 128
B.4 Output format of SSA 128
B.5 Output format of Columbus 129
B.6 Output format of PINOT 129
B.7 Output format of Ptidej 129
B.8 Input format of DEEBEE 129
B.9 Format of PMART 130
B.10 Implementation of the schema metamodel 134
B.11 Implementation of the program metamodel - first part 135
B.12 Implementation of the program metamodel - second part 136
B.13 Implementation of the result metamodel 137

List of Tables

1.1 The relation between the thesis topics and the corresponding publications. 9

3.1 Pattern candidates . . . 27
3.2 Some predictor values for Adapter Object . . . 28
3.3 Some predictor values for Strategy . . . 29
3.4 Average accuracy with standard deviation based on three-fold cross validation . . . 31
3.5 Learning statistics . . . 31

4.1 About the size of each project studied . . . 38
4.2 DC++ candidates . . . 39
4.3 WinMerge candidates . . . 40
4.4 Jikes candidates . . . 41
4.5 Mozilla candidates . . . 42
4.6 Initialization times . . . 44
4.7 Initialization time - exporters . . . 44
4.8 WinMerge times . . . 45
4.9 Jikes times . . . 46
4.10 Mozilla times . . . 46
4.11 Memory requirements in megabytes . . . 47

5.1 Distribution of uploaded candidates . . . 56
5.2 Number of design pattern candidates found for Reference Implementations 67
5.3 Number of design pattern candidates found for NotePad++ . . . 71
5.4 Number of design pattern candidates found for FormulaManager . . . 72

6.1 Tools and requirements satisfied by their output formats . . . 78

7.1 Contain and overlap values for the previous example . . . 99
7.2 Results on NotePad++ . . . 111
7.3 Results on JUnit . . . 111

B.1 DP-Miner result . . . 127
B.2 Attribute values of DPDX . . . 131

B.3 Property values 131
B.4 Program element hierarchy 133

To my wife,

Márti.

"You can't just ask customers what they want and then try to give that to them. By the time you get it built, they'll want something new."

Steve Jobs

Chapter 1

Introduction

The development of software systems usually includes their specification, design, implementation, testing, deployment and maintenance. The general tendency is that software companies tend to leave some of the steps (e.g. requirements analysis, specification, design and testing) out of the software development process to release the software as fast as possible due to the tight deadlines. The problem is that these skipped steps may turn out to be crucial from the viewpoint of deployment and maintainability of the software system.

For example, a fledgling software company is forced to release its first products as soon as possible to gain some advantage in the market over its competitors. In this way, the first releases are very frequent and fast, and if the company survives the initial difficulties it will rapidly grow into a medium-sized company. However, by that time the maintenance (e.g. satisfying new requests, fixing bugs, adapting to new platforms and environments) of the software becomes very expensive because nobody knows the concrete requirements, the architecture, and the key components of the system. Furthermore, several bugs may be reported by customers because there were insufficient resources to test the software during the preceding releases. These reported bugs could seriously harm the company's reputation. The situation is frequently made worse when the original developers leave the company and then nobody knows how the system was implemented. In this way, a new software system can become a legacy system after a relatively short period of time [25]. However, the company has to manage this crisis to survive in the market.

The company has two possibilities to manage the evolved crisis. It can either redevelop the system from scratch or re-engineer the current legacy system.

Redeveloping a system is a really expensive and risky solution for several reasons. First of all, the legacy system has to be maintained before the new system is released to the customers. Maintaining the legacy system will cost as much as the original development because the customers should not feel any kind of effects. In this way, the company gets into a similar situation as during the implementation of the legacy system. It will have

insufficient resources to collect the requirements, describe the specification accurately, design the architecture and plan the test cases of the new system. It means that the company will get into the same trap as it was in in the case of the legacy system: just implementing the new system without due care means that the current problems and crisis will appear two or three years later. However, in certain cases it is better to re-implement a system from scratch, but this decision should be preceded by an in-depth evaluation of the current (legacy) system to see whether it has been re-engineered or not.

Re-engineering a legacy software systems was briefly summarized by Demeyer et. al. [25]: *"The goal of reengineering is to reduce the complexity of a legacy system sufficiently that it can continue to be used and adapted at an acceptable cost."* Most of the time, re-engineering a legacy system is unambiguously a better choice than rewriting it from scratch for several reasons. First of all, several components of a legacy system are not critical while others are modified very rarely. Some components of the system are tested and, typically, most of them are the critical components of the system. Furthermore, the system is currently being used and functioning. On top of this, Demeyer et. al. proposed several patterns [25] that provide solutions to re-engineering legacy systems incrementally and continuously in such a way that the customers of the system do not notice anything. Therefore, in most cases it is much more cost-effective to re-engineer a legacy system than to rewrite it from scratch.

Re-engineering consists of two stages, namely reverse engineering information from the current (legacy) system and, based on this information, (forward) engineering the system into a new form. Reverse engineering applies techniques like analyzing the source code, interviewing developers, browsing the existing documentation, discovering the design and architecture, and identifying the problematic and the key parts of the system. However, most of the time the documentation is completely missing or if it exists it is quite outdated, and developers either do not know the system because they have only worked for the company for a short time or they are afraid of providing the necessary information (e.g. bugs caused by them) about the system. Therefore only the source code of the legacy system can be regarded as an objective starting point of reverse engineering. In this thesis, by the term *reverse engineering* we mean the *reverse engineering of just the source code*.

Re-engineering constructs the system in a new form[1] based on the results of reverse engineering. The re-engineered form of the system should have lower maintainability costs. Successful re-engineering demands a really precise and really reliable reverse engineering of the legacy system because any kind of decision and activity during the forward engineering phase is based on this information. Hence it is very important to ensure correctness, and to improve the results of the reverse engineering step, otherwise

[1] Refactoring is a well-known technique that applied in the field of software engineering.

the whole re-engineering project could be unsuccessful due to the false results of reverse engineering. It motivated us to develop a method which extends and improves one of our reverse engineering tools, and to develop benchmarks and to perform experiments on evaluating and comparing reverse engineering tools.

Reverse engineering tools handle (i) source code parsing and extract an abstract model from it and (ii) perform some exploratory operations on the abstract model. In this thesis we will focus on the *latter type of reverse engineering tools*. Namely, we will 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. By revealing patterns from the source code it is easier to understand how the corresponding classes and methods interact and to work out why they communicate with each other and what business logic is implemented by them. Duplicated code detector tools discover risky copied code fragments. These fragments could carry the same bugs and make the maintenance of the system difficult, because all the fragments should be modified at the same time. Rule violation checkers audit typical programmer errors in the source code. One such error might be comparing two Java *String* objects with the = operator instead of the *equals* method. These tools provide important information about the legacy system, but their results may contain false positives.

Up until now, research teams have focused on the improvement and evaluation of the tools. Petterson et al. [63], for instance, summarized problems during the evaluation of accuracy in *design pattern detection*, Bellon et al. [10] performed tests to evaluate and compare *duplicated code detectors* and Wagner et al. [88] compared three Java *coding rule checkers*. Surprisingly, no author has yet proposed a general framework or benchmark for comparing and evaluating the results of reverse engineering tools like design pattern miners, duplicated code detectors and rule violation checkers. In this thesis, we will focus on these; especially on design pattern mining, how to improve and how to evaluate design pattern miner tools. In the third part of the thesis we propose a benchmark for reverse engineering tools like design pattern miners, duplicated code detectors and rule violation checkers, and show how it can be applied to real-life cases.

1.1 Summary by chapters

The thesis contains three main parts. The first part (Chapter 3) presents a method for improving design pattern mining and also the results of experiments using our design pattern miner tool. The second part (Chapters 4, 5 and 6) discusses techniques, formats and experiments concerning the evaluation and comparison of design pattern miner tools. After, the third part of the thesis (Chapter 7) describes our general framework, which is capable of evaluating and comparing reverse engineering tools.

Part I: An improvement method for design pattern mining

In the third chapter we shall briefly describe our method for improving design pattern mining. We utilized machine learning methods to further refine our pattern miner tool by marking the pattern candidates returned by the matching algorithm as either true or false. We defined predictors for two selected design patterns namely *Adapter Object* and the *Strategy*. We performed our experiments on *StarWriter* [80].

Part II: Evaluation of design pattern miner tools

The fourth chapter describes a comparison and performance evaluation of three design pattern miner tools, namely Columbus, Maisa and Crocopat. Here we provide a comparative approach for design pattern miners. The approach examines the performance indicators (time, memory) and differences found between the results. Afterwards, the three tools are evaluated based on this comparative approach. In this chapter, we do not focus on the correctness issue of the results, but just systematize the common differences among tools.

Chapter 5 discusses the evaluation and comparison of design pattern miner tools concerning the correctness of the results. Here we elaborate on DEEBEE (DEsign pattern Evaluation BEnchmark Environment), an online framework for evaluating and comparing results of design patter miner tools. When comparing the results, DEEBEE handles problems systematized previously in Chapter 3 by relating and grouping similar results of the tools. We will describe the interface of DEEBEE through meaningful usage scenarios and afterwards we will present the results of our experiments. Lastly, the benchmark is evaluated with regard to the requirements proposed by Sim et al. [74].

In the sixth chapter, we present a common output format, DPDX, for design pattern detector tools. A common output format is vital for a uniform evaluation and comparison of the results. Here we define requirements for the common output format and evaluate the current format of design pattern miner tools according to these requirements. Then, we define metamodels of the format that contains the schema metamodel, program metamodel and results metamodel. After, we provide an XML-based implementation of this metamodel.

Part III: Evaluation of reverse engineering tools

In the seventh chapter, we present BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe), an online framework for evaluating and comparing reverse engineering tools. BEFRIEND is a generalized version of DEEBEE. Here we provide the theoretical background needed for handling and grouping the results of different reverse engineering tools. Then, we apply the BEFRIEND interface in representative usage scenarios. After, we will present the results of our experiments performed using the benchmark.

In the last chapter we present conclusions and provide a short summary of the results obtained. In this chapter we also outline possible directions for future work.

Lastly, we round off with appendices that contain related works, details of the DPDX format and a brief summary of the principal results of the thesis in English and Hungarian.

1.2 Summary by results

The main contributions of this work are summarized as follows. First, we introduce a new technique for improving design pattern mining. Afterwards, we evaluate design pattern miners in terms of their speed, memory consumption and the differences in their results. Furthermore, we developed a publicly available online benchmark (DEEBEE) which supports the evaluation and comparison of design pattern miners by considering their correctness and completeness. We also propose a common exchange format for design pattern miner tools that is based on a well-defined metamodel. Lastly, we generalize DEEBEE in BEFRIEND, which supports the evaluation and comparison of reverse engineering tools as well.

We state five key results in the thesis and the contributions of the author are clearly shown in the listed results. As the thesis consists of three parts, the results are also presented in three parts.

Part I: An improvement method for design pattern mining

The result of the first part of the thesis is a machine learning-based method for improving design pattern mining. This is described in Chapter 3 in detail.

1. **Improvement of an existing design pattern miner tool.**
   Here we enhanced our design pattern miner tool called CAN2Dpm, which is a component of the Columbus framework [13]. 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. In other words, we further *filtered* the pattern candidates returned via the original matching algorithm by eliminating false hits. Our approach in a nutshell is to analyze the candidates returned by the matching algorithm, taking into account various aspects of the candidate code fragment and its neighbourhood, such as whether a participant class has a parent or not, or how many new methods a participant class defines besides a participating method. The information associated with these aspects is referred to as *predictors*, whose values can be used in a machine learning system for decision making. We employ a conventional learning approach; that is, we first manually tag the candidates as true or false and calculate the values of the predictors on the candidates. Then we load these into some learning system (we conducted our experiments using two methods, namely a decision tree-based approach and a neural network approach). This in turn provides a model that incorporates the acquired knowledge, which can later be used for pattern mining in unknown systems. In the current work we test the models with the cross-validation method. We performed our experiments on *StarWriter* [80] as the subject system for pattern mining and we searched for the *Adapter Object* and the *Strategy* design patterns.

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.

Part II: Evaluation of design pattern miner tools

The second part of the thesis focuses on our experiences gained and tools developed for the evaluation of design pattern miners. These are elaborated on in Chapters 4 and 5. This part also introduces a common exchange format (in Chapter 6) for design pattern miner tools, which is needed to readily compare, evaluate, fuse and visualize the results of tools in a standard way.

2. **Performance evaluation of design pattern miner tools.**
   Our aim here was to compare three design pattern miner tools; namely Columbus, Maisa and CrocoPat. We chose these tools because it was possible to prepare a common input for them with our front end called Columbus. Earlier work by us enabled us to provide the input for Maisa [34], while in the case of CrocoPat we created a new plug-in for Columbus which can generate the right format. Next, the tools were compared from three aspects: differences between the hits, their speed and memory requirements. We think that these are important aspects for a design pattern miner tool. We did not examine whether a design pattern hit was true or false, but just concentrated on structural hits and differences.

   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 and obtained the results 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.**
   Here we present experiments performed on a newly developed benchmark for evaluating and comparing design pattern miner tools. The benchmark is quite general from the viewpoint of the mined software systems, programming languages, uploaded design pattern candidates, and design pattern miner tools. With the help of this benchmark, the accuracy of two design pattern miner tools (Columbus and Maisa) were evaluated on reference implementations of design patterns and on two software systems called NotePad++ and FormulaManager. Here, design pattern instances in NotePad++ were also discovered by hand, so both the precision and recall scores were calculated by the benchmark automatically. In addition, we developed a software system called FormulaManager to test the tools on a program where each design pattern was implemented in a real-life context.

   With the help of the benchmark one can evaluate design pattern miner tools, which will hopefully lead to better quality design pattern miner tools in the future. With

this benchmark the results of a pattern miner tool should be quicker and easier to classify.

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**
   Here we present DPDX, a common exchange format for design pattern miner tools. First we define the requirements for the output format of a design pattern miner tool. Then, we examine the formats of existing tools based on the above-defined requirements. After, taking into account the identified problems of the existing formats, we analyze the requirements in detail to see how they may be fulfilled completely by the proposed format. Based on these experiments we propose a well-defined and extendible metamodel that addresses a number of limitations of current tools. The proposed metamodel is implemented in an XML-based language that can be easily adapted by existing and future tools, providing a means for improving accuracy and recall scores when evaluating, comparing and combining their findings.

   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.

Part III: Evaluation of reverse engineering tools

The results of the third part of the thesis include BEFRIEND, a general benchmark for reverse engineering tools. This is elaborated on in Chapter 7.

5. **Validation of reverse engineering tools.**
   Here we introduce the further development of the DEEBEE system, which has become more useful since we generalized the evaluation aspects and the type of the evaluated tools. 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 that recognize arbitrary aspects of source code can be subjectively evaluated and compared with each other. Such tools include design pattern miners, duplicated code detectors and coding rule violation checkers. BEFRIEND greatly differs from its predecessor in five aspects:

$\mathcal{N}o.$	[94]	[95]	[96]	[97]	[98]	[99]	[100]	[101]	[102]
1.	•								
2.		•							
3.			•	•	•				
4.						•	•		
5.								•	•

Table 1.1: The relation between the thesis topics and the corresponding publications.

(1) it permits uploading and evaluating results related to different domains; (2) it permits adding and deleting the evaluating aspects of the results in an arbitrary way; (3) it has a new user interface; (4) it generalizes the definition of sibling relationships; and (5) it permits uploading files in different formats by adding the appropriate uploading plug-in.

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.

Lastly, Table 1.1 summarizes which publications cover which results of the thesis.

"An investment in knowledge pays the best interest."
Benjamin Franklin

Chapter 2

Background

In this chapter we will provide the necessary background for chapters 3 to 8. First, in Section 2.1 we shall briefly introduce the concept of reverse engineering and then in Section 2.2 we will clarify the terminology used in the thesis. After, in Section 2.3 we will present the Columbus framework that will be applied several times later on.

2.1 Reverse engineering

Previously we discussed reverse engineering in the Introduction through a meaningful example of a company. In this example the importance of reverse engineering was also demonstrated. In this section we collect and present some formal definitions and terms used in reverse engineering.

Chikofsky and Cross [21] formally defines re-engineering in the following way:

> "*Reengineering* is the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form." [21]

As we said in the Introduction, re-engineering is based on two processes which are also defined by Chikofsky and Cross:

> "*Reverse engineering* is the process of analyzing a subject system to (a) identify the system's components and their interrelationships and (b) create representations of the system in another form or at a higher level of abstraction." [21]
>
> "*Forward engineering* is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system." [21]

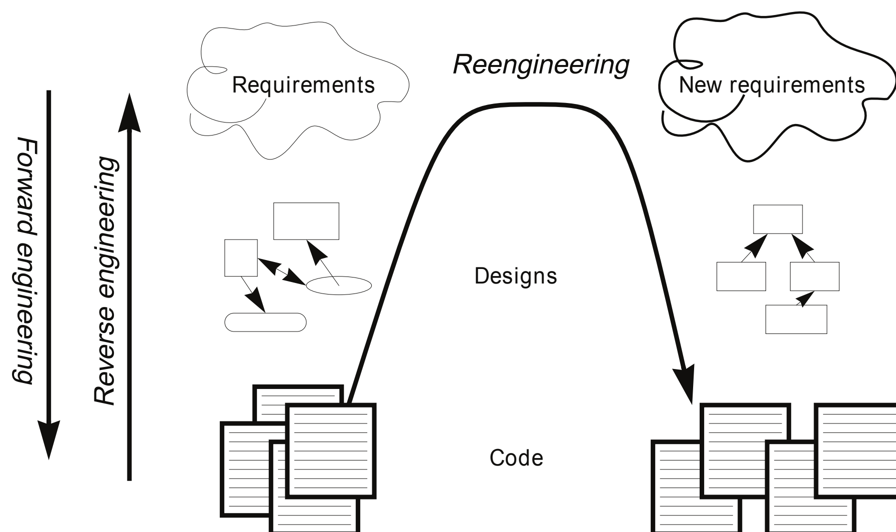

Figure 2.1: The reengineering process.

In [25], Demeyer et al. demonstrated in diagram form how reverse engineering and forward engineering work together in the re-engineering process (see Figure 2.1). Reverse engineering constructs a model from the source code. It represents information about the source code, which, for example, helps us to understand the system (e.g. design pattern mining) and to discover concrete problems in the system (e.g. rule violation checkers and duplicated code detectors). Based on the information provided by the reverse engineering process and on the new requirements of the system, the original design can definitely be improved. For example, a duplicated code detector tool discovers copied clone fragments that should be fixed. In this case, the problem is first fixed at the design level: the transformed model introduces a new method representing the common functionality and the originally copied clone fragments are replaced with method calls to the new common method. Afterwards, the source code is updated based on the transformed model.

Reverse engineering tools basically deal with two tasks. The first task is source code parsing and extracting an abstract model from the source code, while the second one is performing some exploratory operations on the abstract model. In this thesis we will deal with the latter task of reverse engineering. *Reverse engineering tools* in this thesis mean these kind of tools. Namely, we will deal with design pattern miners, duplicated code detectors and rule violation checkers. These three areas will now be described.

Design pattern miners The correct use of *design patterns* in software development is a commonly used premise for the good quality of the design in terms of - among other things - reusability and maintainability. Well-established and documented design patterns exist in various areas of software development. One of the most commonly recognized pattern catalogues was compiled by Gamma *et al.* [39], which describes patterns used

in object-oriented analysis and design. Here, we will make use of these patterns. Most of the systems contain occurrences of design patterns, irrespective of whether they are introduced by the designer intentionally or unwittingly. Whatever the case, knowing about the instances of patterns in a software system may be of great help during the software maintenance phase (for example, to better understand the system's structure and workings). Unfortunately, in many cases the pattern usage is poorly documented, so during the reverse engineering process the automatic or semi-automatic procedures used for discovering design patterns in undocumented software can be a great help if we wish to re-engineer the software. There are several tools available for discovering patterns in Java source code like Ptidej [42, 65] and Fujaba [37], and tools also exist for mining patterns from C++ code like Columbus [8, 32]. However, the automatic recognition of design patterns is indeterministic because design patterns convey concepts not concrete solutions. Gamma et al. [39] also proposed concrete solutions, but these solutions may appear in the source code with another intent as well. Therefore, tools mining design patterns produce a certain amount of false results which need to be filtered out.

Duplicated Code Detectors Most developers know about the copy-paste method. The copy-paste technique means copying a certain code fragment and pasting it into another part of program code. The problem with this technique is that if the original code fragment contains bugs (errors) then the copied fragment will contain these bugs as well, hence the number of bugs immediately increases in the system. In addition, when the copied fragment has to be modified then the copied fragments also have to be modified, but it usually not known where the copied fragments are located. However, developers tend to use this technique frequently because of tight deadlines. They use it because it seems quick and effective at first glance, but it carries a potential danger: every copied code fragment has to be modified consistently in the future. In this way, this technique is a potential pitfall for the developer and for the company as well. To address this problem, several duplicated code detector tools have been published (e.g. Bauhaus [9], CCFinder [19] and Simian [78]). However, detecting duplicated code is not a trivial task. Copied fragments are frequently modified inconsistently, which will make detection difficult and problematic. Thus in practice duplicated code detector tools may produce some false results and miss some real clones.

Coding Rule Checkers Coding rules describe rules for programming. These rules are various and some rules relate to the coding style, while others relate to critical coding mistakes, which can cause runtime errors. For example, *Basic Ruleset* of *PMD* [64] contains the *EmptyCatchBlock* rule: *"Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported."* A *coding rule violation* occurs if the programmer does not follow a certain rule and he violates it. Coding rule checker

tools discover these violations in the given system. Several rule checkers exist such as PMD [64], CheckStyle [20], FindBugs [35] and FxCop [38].

The above are just examples of different aspects of areas of reverse engineering. We will elaborate on these areas, especially that of the evaluation and comparison of design pattern miner tools.

2.2 Terminology

In this section the terminology that will be used throughout the thesis is clarified and summarized. Based on the corresponding publications the clarification of terminology should be a part of chapters 6 and 7. However, it is much better to gather and present these terms at the beginning of the thesis to make everything clearer. Figure 2.2 shows the most frequent terms used in the thesis.

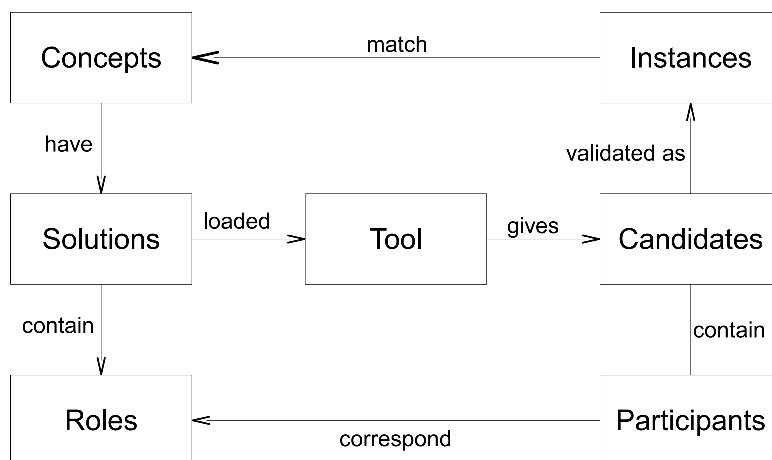

Figure 2.2: Concept map of the most frequent terms of the thesis.

Reverse engineering tools can be classified into *domains*. A *domain* groups tools with the same aim e.g. design pattern miners, duplicated code detectors, or rule violation checkers.

Each *domain* contains *concepts* which describe a concrete problem. For example, in the case of the design pattern miner *domain*, one *concept* is the State design pattern. The concepts of this pattern were summarized by Gamma et al. [39] in terms of its intent, a motivation example and its applicability.

Every *concept* has at least one *solution*. For example, in the case of the design pattern miner *domain*, Gamma et al. [39] proposed a structure for the State design pattern (see Figure 2.3). However, this structure can implemented in several ways, which we refer to as *solutions* (e.g. aggregation between Context and State can be implemented with a simple reference or with a standard container). Moreover, certain implementations (solutions) can significantly differ from this implementation (e.g. Context and State are represented in one common class).

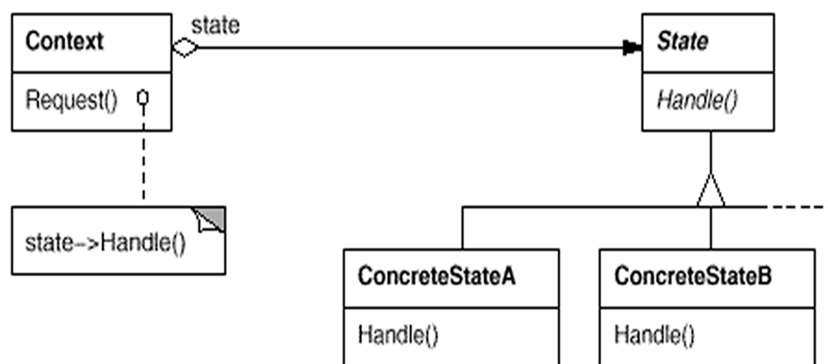

Figure 2.3: The State design pattern.

A *solution* contains *roles*. For example, the proposed structure of the State pattern contains the class level *roles* Context, State, and ConcreteState.

The tools in a given domain produce different results that refer to one or more positions in the source code being analyzed. We refer to these positions as result *candidates*. A certain tool discovers *candidates* based on the *solution* that was the basis of the discovery process. However, in certain cases a discovered *candidate* does not correspond to the original *concepts*. In such cases we refer to the *candidate* as a *false positive* or *false hit*. Otherwise, if the *candidate* corresponds to the original *concepts* then we will refer it as a *true positive*, a *true hit* or an *instance* of the original *concept*. Human inspection is required to decide whether a *candidate* matches the *concepts* and represents an *instance* or not. For example, Gamma et al. [39] proposed exactly the same structure for State and Strategy patterns, which cause these patterns to have the same *candidates*. Only human inspection can decide whether a certain *candidate* is a Strategy or a State *instance*. Candidates and instances may be categorized in the following way:

- *True Positives* (TP): instances found by the tool (correctly).
- *False Positives* (FP): false candidates reported by the tool (incorrectly).
- *True Negatives* (TN): false candidates not reported by the tool (correctly).
- *False Negatives* (FN): instances not found by the tool (incorrectly).

Based on these aspects, the following two well-known statistics can calculated. The *precision* score is defined as $\frac{TP}{TP+FP} \times 100$, which means the ratio of correctly identified instances over all found candidates. The *recall* score is defined as $\frac{TP}{TP+FN} \times 100$, which means the ratio of correctly identified instances over all existing instances.

The candidates found include other elements which are called as *participants*. *Participants* correspond to the *roles* of the current *solution*.

The design pattern miner domain contains several concepts (design patterns), while the duplicated code detector domain contains just one (duplicated code). Similar to the design pattern miner domain, the rule violation checker domain also contains several concepts (empty catch block, broken null check, etc. [64]). However, *roles* and *participants* are only used for the design pattern mining domain.

Design patterns In this thesis we mainly focus on design pattern miner (DPM)[1] tools. This is why we need to introduce additional terms and definitions for this area [99],[100].

A *design pattern* describes a recurring design problem. A design pattern includes at least four parts: a name, a problem, a structure, and the consequences of applying the proposed structure (see Gamma et al. [39]). The implemented *solution* of a design pattern is a *design motif* (see [44]), which describes a prototypical set of classes and/or objects collaborating to solve the design problem. A motif typically describes several *roles*, which must be fulfilled by program constituents (types, methods, fields, etc.), their relations (inheritance, subtyping, association, etc.), and/or their collaborations (expressed in terms of code fragments or UML-like sequence diagrams). Roles can be *mandatory or optional* [53].*Mandatory* roles(e.g. 'State' in the 'State' motif) represent the essence of a motif. *Optional roles* might not be present in some instances (e.g. 'Context' may be missing). An *instance* of a design pattern *P* is a set of program constituents playing all the mandatory roles (and possibly some or all the optional roles) in the motif of P. Program constituents playing the mandatory roles are called as *fundamental participants*. A *candidate* of a design motif is a set of program constituents meant to form an instance of the motif in the program and, generally, reported by a design pattern miner tool, which typically report candidates that they deem consistent with the design motif P. Although textbooks typically describe just one motif per design pattern in explicit detail, there may be several implementation variants for each pattern, thus several design motifs (solutions) need to be searched for by DPD tools.

2.3 Columbus framework

Some tools (the Columbus design pattern miner component, Maisa, CrocoPat) that will be evaluated here have been integrated into the Columbus reverse engineering framework.

[1] The name design pattern detector (DPD) is a synonym for this.

Moreover, our improvement of the design pattern miner component of the Columbus framework is also an important part of the thesis. Therefore, it is necessary to introduce the Columbus framework.

Columbus was developed in cooperation between FrontEndART Ltd., the University of Szeged and the Software Technology Laboratory of Nokia Research Center. Columbus is able to analyze large C/C++ projects and to extract facts from them. The main motivation for developing the Columbus system was to create a general framework for combining a number of reverse engineering tasks and to provide a common interface for them. Thus, Columbus is a framework tool which supports project handling, data extraction, data representation, data storage, filtering and visualization. All these basic tasks of the reverse engineering process for the specific needs are accomplished by using the appropriate modules (plug-ins) of the system. Some of these plug-ins are provided as basic parts of Columbus, while the system can be extended to meet other reverse engineering requirements as well. This way we have a versatile and readily extendible tool for reverse engineering tasks.

Figure 2.4 shows the architecture of the Columbus framework with some components. First, Columbus (CAN) analyzes the source code and builds the corresponding representation (Abstract Semantic Graph – ASG) from it. Afterwards, the plugins load this ASG and they collect the necessary information by traversing the ASG. Lastly, the tools generate their results.

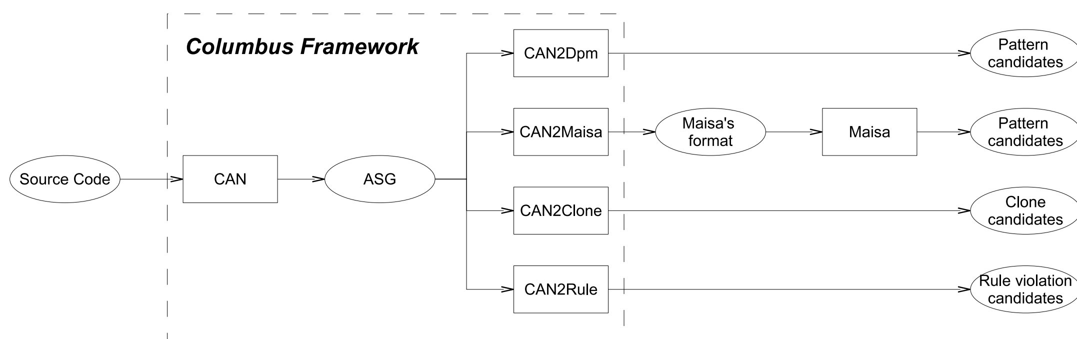

Figure 2.4: The Columbus framework

One of the Columbus's plug-ins is CAN2Dpm, which discovers design patterns. The design patterns were described in DPML (Design Pattern Markup Language) files [8], which store information about the structures of the design patterns. Here CAN2Dpm seeks to match this graph to the ASG using our algorithm described in a previous work [8].

Maisa Maisa is a software tool [62] for the analysis of software architectures developed in a research project at the University of Helsinki. The key idea behind Maisa is to

analyze design level UML diagrams and compute architectural metrics for the early quality assessment of a software system.

Maisa uses constraint satisfaction [56], which is a generic technique that can be applied in a wide variety of tasks (in our case, to mining patterns from software architectures or software code). A constraint satisfaction problem (CSP) is represented as a set of variables and a set of constraints that restrict the values that can be assigned to these variables. The language of Maisa's design pattern description is Prolog-like.

Maisa has also been integrated into the Columbus framework with the introduction of the CAN2Maisa plugin [33]. CAN2Maisa generates the appropriate input file for Maisa. After, Maisa discovers design pattern candidates from this intermediate file.

Part I

A proposed method for improving design pattern mining

"The question of whether computers can think is like the question of whether submarines can swim."
Edsger W. Dijkstra

Chapter 3

Improvement of an existing design pattern miner tool

It is clear that recognizing design patterns is a crucial task of the software maintenance process. In particular, design pattern mining supports the understanding of a software system by exposing the key participants. Moreover, it can partly make up for the missing documentation, which is quite a common problem during software development. However, there is no guarantee that the results produced will be correct.

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. This is due to the fact that the patterns themselves are given using conventional object-oriented concepts, such as class diagrams containing abstract classes and methods, generalization, polymorphism, decoupling concrete responsibilities through references to abstract classes, and so on. This leads to structures that are in many ways quite similar to each other (consider the structures of, say, Bridge vs. Adapter Object[1], Composite vs. Decorator or State vs. Strategy). Furthermore, such common structures may appear even for code fragments that were not designed with the intent of representing any specific design pattern, but make use of the above-mentioned techniques simply as good object-oriented solutions to other problems. The distinction between such true and false results and between different patterns with the same structures can be made only by applying more sophisticated methods that involve a deeper investigation of the implementation details and its environment, i. e. to find its real purpose.

In this chapter we propose a machine learning method and experiments on how to improve the results of design pattern miner tools to decide whether they are correct or

[1] The Adapter design pattern has two variants; namely a *class* version and an *object* version, which differ in the way the adaptation is achieved; one is by using multiple inheritance, while the other is by object composition.

not. We applied the design pattern mining approach of our Columbus framework [8], but each experiment was repeated since both the C++ front end and the pattern mining algorithms have improved quite a lot since then. We carried out experiments on StarWriter (containing over 6,000 classes), the text editor of the StarOffice suite [80]. Using the pattern-matching algorithm of Columbus we first found several hundred pattern candidates that were further filtered using machine learning methods to provide more accurate results.

3.1 The learning process

In the following we will give an overview of the concrete steps of the learning process we developed. It consists of four consecutive steps, which are the following:

1. *Predictor value calculation.* In the original pattern mining process [8], Columbus analyzes the source code and creates an ASG (Abstract Semantic Graph) 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 that are not related to its structural description. We retrieve this kind of information from the source code and use them as input for the learning system. We call these collected values predictors. In this first step we calculate predictor values from the ASG and then save them to a CSV file (predictor table). This file is basically a table containing the predictor values for each design pattern candidate.[2]
2. *Manual inspection.* Here, we examine the source code manually to decide whether the design pattern candidates are instances or false candidates. Then we extend the predictor table file with a new column containing the results of the manual inspection.
3. *Machine learning.* Next, we perform the training of the machine learning systems. The outputs of these systems are model files which contain the acquired knowledge.
4. *Integration.* Lastly, we integrate the results of machine learning (the model files) into Columbus to be able to make smarter decisions by filtering the design pattern candidates. This way, Columbus should report far fewer false positive design pattern candidates to the user.

Figure 3.1 graphically describes this process. The original elements of the design pattern mining process of Columbus are denoted by straight lines and empty boxes, while the new parts introduced by the learning process are denoted by dashed lines and filled boxes.

[2] We classify the predictor values according to their magnitude to achieve better learning results. We divide the values into equal intervals and use the classes corresponding to these intervals as the input for the learning algorithms.

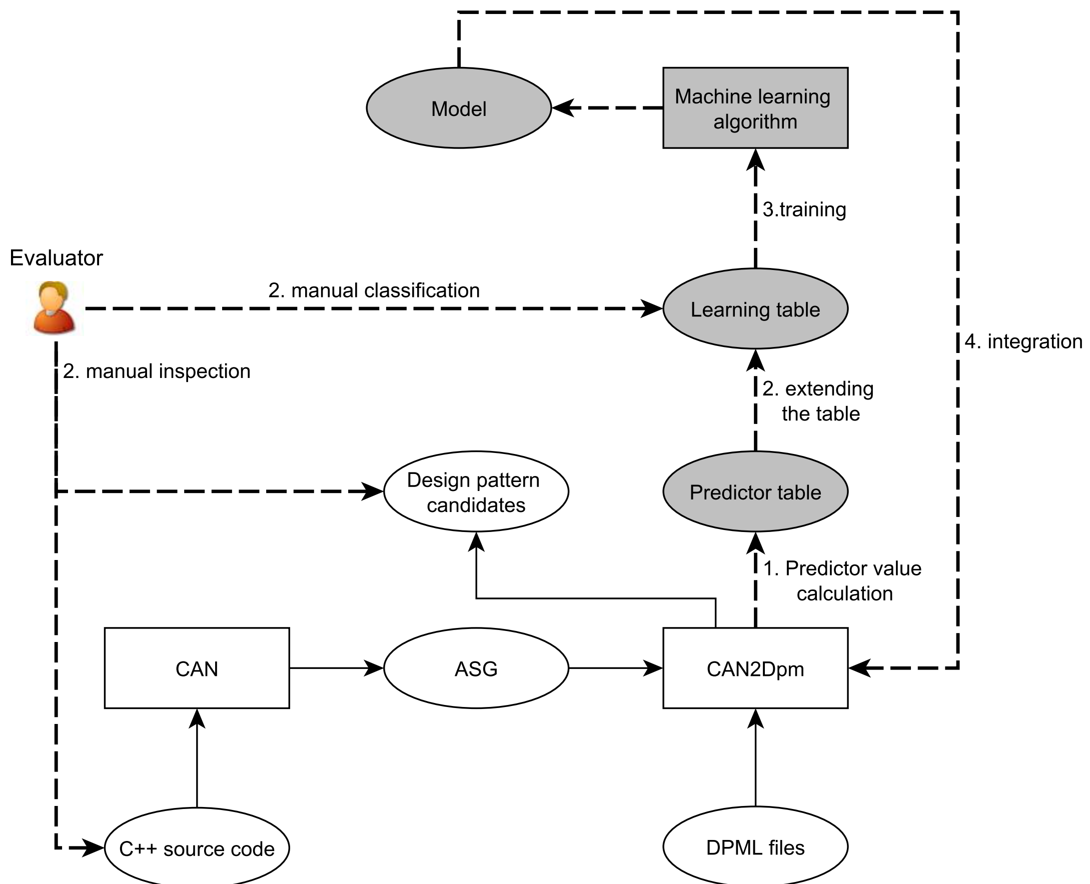

Figure 3.1: The learning process

3.2 Predictors

We chose two design patterns out of 16 patterns handled by the matching algorithm for the experiments. Our final choice was the structural pattern *Adapter Object* and the behavioural *Strategy* pattern since these two occurred most frequently in our experiments. This choice was also appropriate because, after the manual investigation of the candidates, we found that there were enough true and false examples as well. Moreover, the candidates and their contexts were sufficiently different to train the machine learning systems successfully. These two patterns are good examples of how general the structural descriptions of patterns can be in terms of general object-oriented features, and how useful the deeper information can be for their recognition.

In the following we will outline these two design patterns and the predictors that we formulated. We also experimented with other predictors, but these proved to be the most effective.

3.2.1 Adapter object

The aim of the Adapter pattern is to *"convert the interface of a class into another interface that clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces."* [39]

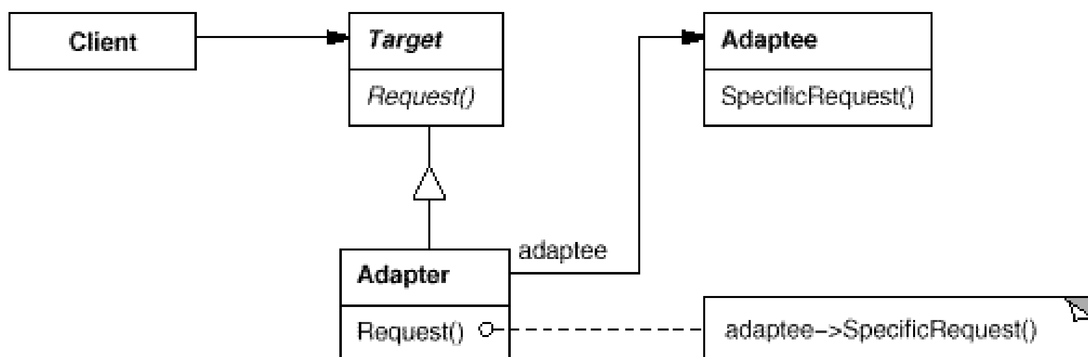

Figure 3.2: The Adapter Object design pattern

The Adapter pattern has four participants (see Figure 3.2). First, the Target class defines the domain-specific interface that Client uses. Client in turn represents the class collaborating with objects that conform to the Target interface. Next, the Adaptee class describes an existing interface that needs adapting, and lastly Adapter is the class that adapts the interface of Adaptee to the Target interface.

There are two forms of the Adapter pattern, namely Class and Object. The former uses multiple inheritance to adapt one interface to another, while the latter uses composition for the same purpose. We employed Adapter Object in our experiments.

It may be seen that this structure, the delegation of a request through object composition, is a quite common arrangement used by object-oriented systems and so only a more detailed analysis may spot the instances of this pattern. Here we identified the following predictors for the Adapter Object design pattern:

- *A1 – PubAdapteeCalls predictor* shows how many public methods of the Adapter candidate class contain a method call to an Adaptee candidate class. Since the main purpose of the Adapter pattern is to adapt the Adaptee class to the Adapter class, we reach the Adaptee objects through the Adapter ones. So most public methods of an Adapter candidate should contain a method call to an Adaptee candidate.
- *A2 – PubAdapteeNotCalls predictor* shows how many public methods of the Adapter candidate class do not contain a method call to an Adaptee candidate class. We assume that the A1 value is greater than A2, because the Adapter's intent is to adapt the functionality of Adaptee so the Adapter must have more public functions that call Adaptee than those that do not.

- *A3 – NotPubAdapteeNotCalls predictor* shows how many non-public methods of the Adapter candidate class do not contain calls to an Adaptee candidate. We assume that if A2 is greater than A1, then the private or protected methods are responsible for calling the Adaptee. Thus, if A2 is larger than A1, then we assume that A3 will be a low number because the non-public methods should call the Adaptee.
- *A4 – MissingAdapteeParameter predictor* shows how many constructors of the Adapter candidate class do not get an Adaptee candidate as a parameter. The Adapter is likely to get the Adaptee object that it will manage via its constructors, so it should be zero or a low value.
- *A5 – AdapteeParameterRatio predictor* shows the ratio of the constructors of the Adapter candidate class that get an Adaptee candidate object as a parameter. We assume that it should be one, or close to one.
- *A6 – NewMethods predictor* shows how many new methods the Adapter candidate class defines. Client uses Adaptee through the interface of the Target class, so since the purpose of the Adapter class is to define the methods of Target, no new methods of its own need to be added.

3.2.2 Strategy

The intent of the Strategy pattern is to *"define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."* [39]

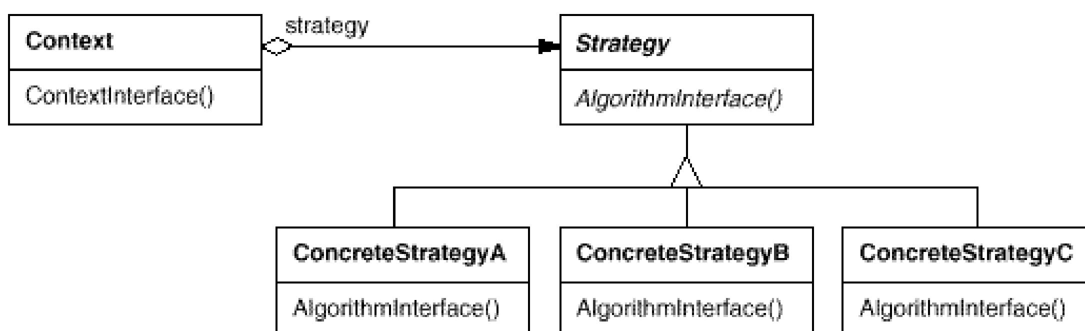

Figure 3.3: The Strategy design pattern

The Strategy pattern has three participants (see Figure 3.3). The Strategy class declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy, which implements the algorithm using the

Strategy interface. The Context class is configured with a ConcreteStrategy object and maintains a reference to a Strategy object.

The fact that the implementation of the algorithm interface is achieved simply by realizing the Strategy interface with inheritance and method overriding suggests that this pattern also requires a more detailed analysis to distinguish its instances from the false candidates. Here we identified the following predictors for the Strategy design pattern:

- *S1 – InheritanceContext predictor* shows the number of children of the Context candidate class. It should be a low value, because otherwise the pattern would be more similar to the Bridge pattern than to Strategy.
- *S2 – IsThereABase predictor* shows the number of parents of the Strategy participant. We assume that Strategy does not have any parents because it provides the interface to change the strategy.
- *S3 – Algorithm predictor* investigates the ConcreteStrategy candidate classes. It represents a value based on the ConcreteStrategy candidates' algorithmical features like the number of loops and recursions. We suppose the more algorithmic features it has, the higher the probability of it being a true ConcreteStrategy candidate (instance).
- *S4 – ConcreteStrategy predictor* shows the number of ConcreteStrategy candidates discovered. If this is very low or one then the main advantage of the Strategy pattern will be lost.
- *S5 – ContextParam predictor* shows the number of methods of the Context candidate which have a Strategy parameter. Usually the Context class forwards the client's requests to Strategy so that the client can select the Strategy object at runtime. Thus Context should have at least one method with a Strategy parameter.
- *S6 – InheritanceStrategy predictor*: shows the number of children[3] of the Strategy candidate class. This value should be close to S4; but in any case it must be smaller.

3.3 Machine learning approaches used

We employed two machine learning systems for acquiring knowledge from the predictor sets discussed in the previous section. Both systems produce a model that will be used for decision making. Besides having the same predictor sets, these two algorithms were also used in the same way so we were able to test both in the same environment.

[3] Note that this value is not the same as that for S4 because the ConcreteStrategy-s may be located deeper in the class hierarchy, and not all children of Strategy are necessarily ConcreteStrategy-s.

The above represent some of the most popular approaches in the area of machine learning, one being a decision tree and the other a neural network. The system we employed for the former was C4.5 (which uses an enhanced version of the ID3 algorithm), while for the latter it was the Backpropagation algorithm.

C4.5 is an enhanced implementation of the ID3 algorithm that was proposed by Quinlan in 1993 [66]. The C4.5 algorithm makes use of a variant of the rule post-pruning method to find high precision hypotheses to the target concept of the learning problem. It generates a classification-decision tree for the given data set by recursively partitioning the data. Training examples are described by attributes (predictor values) whose choice for a given tree node depends on their information gain at each step during the growth of the tree.

The Backpropagation algorithm [16] works with neural networks that are the means for machine learning, whose reasoning concept was borrowed from the workings of the human brain. This algorithm uses more layers of neurons; it gets the input patterns and gives them to the input layers. Then it computes the output layer (the output decision) from the input layer and the hidden (inner) layers. In addition, an error value is also calculated based on the difference between the output layer and the target output pattern (the learning data). The error value is propagated backwards through the network, and the values of the connections between the layers are adjusted in such a way that the next time the output layer is computed the result will be closer to the target output pattern. This method is repeated until the output layer and target output pattern are almost equal or up to some iteration limit.

3.4 Results

Here we will present the results of our experiments concerning the accuracy of the learning methods and their effect on design pattern recognition.

The basic pattern matching-based algorithm of Columbus found 84 candidates of Adapter Object and 42 of the Strategy pattern in StarWriter [80]. Next, we performed a manual inspection of the source code corresponding to the candidates found and provided a classification for each candidate; either as a false or true candidate (instance). Table 3.1 lists some statistics about this classification.

Pattern	Total candidates	False candidates	True candidates
Adapter Object	84	59	25
Strategy	42	35	7

Table 3.1: Pattern candidates

This manually tagged list of candidates was then used as the training set for the learning systems together with the calculated predictor values for each candidate, as described in Section 3.1.

In the following we will first overview our experiences with the investigation of the candidates and their relation to the actual predictor values, and then we will present our results concerning the learning efficiency.

3.4.1 Adapter object candidates investigation

During our investigation of the Adapter Object candidates we found that they can be divided into groups that share some common features. We will present two examples of these groups.

Candidates belonging to the first group all have an Adapter class that references another class through a data member of a pointer type. The referenced class is, however, too simple to be considered as an Adaptee as it has very few members or too few methods of it are used by the Adapter. For example, the Adapter contains a String (which is the Adaptee) that holds the name of the current object, and one of the Adapter's methods needs the length of the String so it calls the corresponding method of String. Clearly, these candidates are not real Adapter Object patterns.

Candidates of the second group have an Adapter class that implements some kind of collection data structure like a set, list or an iterator. These code fragments also have the structure of an Adapter Object design pattern, but their purpose is obviously different, so they are not real patterns either.

After we had classified the candidates as true (instance) or false candidates and the predictor values had been calculated, we investigated whether the actual predictor values support our assumptions about the predictors given in Section 3.2. Table 3.2 shows the predictor values for some typical candidates along with their manual classification results.

Candidates	C1	C2	C3	C4
A1 - PubAdapteeCalls	5	1	0	2
A2 - PubAdapteeNotCalls	4	0	35	14
A3 - NotPubAdapteeNotCalls	0	2	11	52
A4 - MissingAdapteeParameter	0	0	4	2
A5 - AdapteeParameterRatio	1	1	0.33	0
A6 - NewMethods	7	1	26	9
Classification (True/False)	T	T	F	F

Table 3.2: Some predictor values for Adapter Object

Let us look at, say, the C1 candidate, which is a real pattern instance. We can see that more public methods in Adapter call the Adaptee (A1>A2), while there are no non-public

methods that do not call it (A3=0). The values of A4, A5 and A6 also support our assumptions (all Adapter constructors take an Adaptee). Let us take the C3 candidate, which is a false candidate, as another example. It can be seen that the relations between the values of A1, A2 and A3 are exactly the opposite of the true example, i. e. there are few calls from Adapter to Adaptee. A5 and A6 support our assumptions as well. Based on this, we may safely assume that the learning methods probably discovered these relationships as well.

3.4.2 Strategy candidates investigation

First, we will overview some interesting Strategy candidates. There was a class called SwModify, which behaved as Context, while SwClient assumed the role of Strategy. The former had a method called Modify that would call the Modify method of the latter. Inherited (direct or indirect) classes of SwClient defined different Modify methods, so we classified this candidate as an instance. We also noticed that the number of ConcreteStrategy-s was quite high.

The above-mentioned Strategy class called SwClient appeared in another candidate too as the Strategy class, which we eventually treated as a false one for the following reasons. Context was represented by the class SwClientIter that communicated with SwClient through the AlgorithmInterface method called IsA. ConcreteStrategy-s defined this method, but since its purpose was only RTTI (runtime type identification) and not that of a real algorithm, we classified this candidate as false.

There were also some candidates with only one ConcreteStrategy. We decided to classify these as false candidates because, after investigating all other instances, it was obvious that a real Strategy pattern instance should have several ConcreteStrategy-s otherwise its initial purpose is lost.

We investigated the predictor values together with the classifications of Strategy in more depth in order to verify our initial assumptions about the predictors. Table 3.3 shows four example candidates with the predictor values and the classifications.

Candidate	C1	C2	C3	C4
S1 - InheritanceContext	0	0	3	1
S2 - IsThereABase	0	1	3	1
S3 - Algorithm	10.15	13.6	0	0
S4 - ConcreteStrategy	65	10	1	2
S5 - ContextParam	2	1	2	6
S6 - InheritanceStrategy	57	9	2	35
Classification (True/False)	T	T	F	F

Table 3.3: Some predictor values for Strategy

The C1 candidate is classified as an instance (true positive). It can be clearly seen that the S1 and S2 predictors are low, while S3 and S4 are high, as expected. Predictor S5 is greater than 0 and S6 is smaller than S4, so this appears to be a true positive candidate.

C4 is a false candidate, where the S1 and S2 predictors are not zero. Furthermore S3 and S4 are very low, which suggests that this should be a false candidate according to our above assumptions. Lastly, S6 is also much higher than S4, which further supports the belief that this is a false candidate. The manual classification of it was false, so our assumptions about the predictors were again correct.

In the next section, where we present the actual results of the learning efficiency, we will see that the learning methods successfully discovered these features of the predictors and incorporated them into their models.

3.4.3 Learning efficiency

To assess the accuracy of the learning process we applied the method of three-fold cross-validation,[4] which means that we divided the predictor table file into three equal parts and performed the learning process three times. Each time we chose a different part for testing and the other two parts for learning. We define the following basic measures that are required to assess the learning efficiency[5]:

- *True Positives of Learning* (TPL): an instance *correctly classified* by the machine learning model as a true candidate (aka instance).
- *False Positives of Learning* (FPL): a false pattern candidate *incorrectly classified* by the machine learning model as a true candidate (aka instance).
- *True Negatives of Learning* (TNL): a false pattern candidate *correctly classified* by the machine learning model as a false pattern candidate.
- *False Negatives of Learning* (FNL): an instance *incorrectly classified* by the machine learning model as a false candidate.

We measured 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 i.e. $(\frac{TPL+TNL}{TPL+TNL+FPL+FNL}) \times 100$. We also calculated the average and standard deviation (shown in parentheses) using these three testing results and got the scores shown in Table 3.4.

It can be seen that the two learning methods produced very similar results, but the accuracy score was worse in the case of Adapter Object. This is probably due to two

[4]We did not have enough design pattern candidates to perform the usual ten-fold cross-validation method, so we divided the training set into three parts instead of ten.

[5]These measures should not be confused with those defined in Section 2.2.

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

reasons. First, one of the three validation tests produced very bad results, which worsened the overall percentage scores, and second, it seems that we found better predictors for Strategy than for Adapter Object.

However, the real importance of the learning accuracy scores will be appreciated only by investigating in more detail how the application of machine learning improves the accuracy of the design pattern recognition. To do this, we applied the following measures:

- **Specificity** (SPC) This measures how the learning model can filter out false candidates: $\frac{TNL}{TNL+FPL} \times 100$
- **Negative Predictive Value** (NPV) This shows to what extent the prediction of false candidates was wrong: $\frac{TNL}{TNL+FNL} \times 100$
- **Recall** (R) This measures how many of the manually classified true positive candidates (instances) are correctly classified by the learning method: $\frac{TPL}{TPL+FNL} \times 100$.
- **Precision** (P) This shows the degree of correctly classified true positive candidates (instances): $\frac{TPL}{TPL+FPL} \times 100$

Algorithm	Neural Network		Decision Tree	
Pattern	Adapter Object	Strategy	Adapter Object	Strategy
TPL	6	7	5	5
FPL	9	2	8	2
TNL	50	33	51	33
FNL	19	0	20	2
Specificity	84.75%	94.29%	86.44%	94.29%
Negative Pred. Val.	72.46%	100%	71.83%	94.29%
Recall	24%	100%	20%	71.43%
Precision	40%	77.78%	38.46%	71.43%

Table 3.5: Learning statistics

Table 3.5 shows the results obtained for these measures for each learning method. It may be concluded from the table that specificity is really good with both learning methods and design patterns (85–94%), and that few true candidates (instances) were wrongly predicted as false ones, so the NPV of filtering is also quite good.

As for the ability to predict true positive candidates (instances), the results are diverse. Strategy candidates were fairly well classified, but only one fourth to one fifth of Adapter Object instances were found by the learning methods with a modest precision score. This may be accounted for by the fact that the predictors were designed with the intent to filter out false pattern candidates (Specificity).

3.5 Summary

In this chapter we presented an approach whereby significant improvements in accuracy can be achieved in design pattern recognition compared to the conventional structure matching-based methods. The main idea here was to employ machine learning methods in order to refine the results of the structure-based approaches.

Our goal was to filter out false candidates from the results provided by our structure-based pattern miner algorithm [8]. In our experiments we achieved learning accuracy scores of 67–95% and with the model obtained we were able to filter out 51 of the 59 false candidates of the Adapter Object design pattern (out of a total of 84 candidates) and 33 of the 35 false candidates of the Strategy pattern (out of a total of 42 candidates).

The method presented here has a critical part. The manual evaluation of design pattern candidates, which are discovered by a certain tool, is a really time-consuming, dull and taxing task. The person who carries out the assessment has to look for the appropriate source file(s) and locate the appropriate source code fragment(s), and then investigate the fragments together to get the whole picture. Afterwards, he has to record his opinions for later use. However, this part is really important in our procedure, so tool-support for this would be really useful. Among other things and based on this motivation, we developed DEEBEE, an online framework and benchmark that is capable of evaluating and comparing design pattern candidates. DEEBEE will be presented later in Chapter 5.

Part II

Evaluation of design pattern miner tools

"The only source of knowledge is experience."
Albert Einstein

Chapter 4

Performance evaluation of design pattern miner tools

The purpose of this chapter is to 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 front end. Our study was based on the same input data, hence ensuring a fair-minded comparison, because any parsing errors affected all three tools in the same way. The tools were compared with three views in mind: differences between the candidates, speed and memory requirements. We did not analyze whether a design pattern candidate was true or false; we examined these tools only from the viewpoint of structural candidates and differences.

From the results of experiments we found that CrocoPat is generally the fastest, Maisa needs the least memory and Columbus is the fastest in the case of mining very complex patterns.

4.1 Framework

Our previous work enabled us to provide the input for Maisa [34], 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 4.1.

4.1.1 CrocoPat

Beyer et. al. [14, 15] developed a system that is able to work with large graphs effectively. The effectiveness of the system is based on binary decision diagrams which represent the relations compactly. They developed the relation manipulation language (RML) for

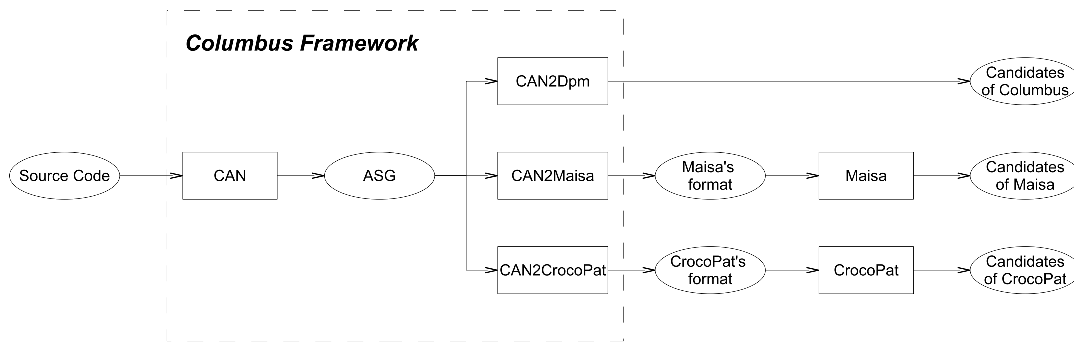

Figure 4.1: Common framework

manipulating n-ary relations and a tool implementation called CrocoPat. CrocoPat is an interpreter, and it executes RML programs. First, CrocoPat reads the input graph in Rigi Standard Format (RSF) [59] from the standard input. Afterwards, CrocoPat reads the RML description and creates a BDD representation from it. After, an RSF output is produced that shows the patterns found in the input graph according to the given RML description. We extended the Columbus framework with a plugin that generates the appropriate input graph for CrocoPat based on the information in the ASG[1].

The RML (Relational Manipulation Language) is very similar to logic programming languages like Prolog, but it contains techniques of imperative programming languages too. Hence, it is very expressive and it can describe design patterns among other structures. Unfortunately, we have not found any design pattern library in RML, so we created the descriptions of the patterns in the RML language for 18 design patterns. Figure 4.2 shows one concrete implementation of these, namely the description of the Factory Method design pattern in CrocoPat's RML language.

4.2 A comparative approach

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

- *Differences between the design pattern candidates found.* In a fair comparison of the tools we have to know and take into consideration why their results are different. However, the differences might be due to several reasons. The different tools might use different techniques to define a design pattern. Most of the time the recognition algorithms are different as well. Furthermore, the representation of

[1]The Columbus framework was introduced in Section 2.3.

```
AbstractClass(X) := CLASS(X) & ABSTRACT(X);
Product(X) := AbstractClass(X);
ConcreteProduct(Cpr,Pr) := CLASS(Cpr) & Product(Pr) &
        TC(INHERITANCE(Cpr,Pr));

Creator(Cr,Pr) := AbstractClass(Cr) & ASSOCIATION(Cr,Pr) & Product(Pr) &
        Cr != Pr;
CreatMethods(Cr,Pr,M) := Creator(Cr,Pr) & HASMETHOD(Cr,M);
CreatorFM(Cr,Pr,FM) := CreatMethods(Cr,Pr,FM) & VIRTUAL(FM) &
        PUREVIRTUAL(FM) & RETURNS(FM,Pr);
CreatorAM(Cr,Pr,AM,FM) := CreatMethods(Cr,Pr,AM) &
        CreatorFM(Cr,Pr,FM) & CALLS(AM,FM);

ConcreteCreator(Ccr,Pr,Cr,Cpr) := CLASS(Ccr) & ASSOCIATION(Ccr,Pr) &
        Product(Pr) & Creator(Cr,Pr) & TC(INHERITANCE(Ccr,Cr)) &
        ConcreteProduct(Cpr,Pr) & Ccr != Pr;
CCreatorFM(Ccr,Pr,Cpr,M) := ConcreteCreator(Ccr,Pr,_,Cpr) &
        HASMETHOD(Ccr,M) & VIRTUAL(M) & !PUREVIRTUAL(M) &
        RETURNS(M,Pr) & CREATES(M,Cpr);

FactoryMethod(Prod,Creat,CProd,CCreat,CreatFM,CreatAM,CcreatFM) :=
       Product(Prod) &
       Creator(Creat,Prod) &
       ConcreteProduct(CProd,Prod) &
       ConcreteCreator(CCreat,Prod,Creat,CProd) &
       CreatorFM(Creat,Prod,CreatFM) &
       CreatorAM(Creat,Prod,CreatAM,CreatFM) &
       CCreatorFM(CCreat,Prod,CProd,CcreatFM) &
       CProd != CCreat &
       Creat != CProd;
```

Figure 4.2: Factory Method pattern in RML

the results might not be the same. Based on this viewpoint, we will try to discover the possible reasons for the differences, and to test our current assumptions.

- *Speed.* Design pattern mining can be a very time consuming task, e.g. because of the cost of a conventional graph matching algorithm. Hence we shall compare the given tools in terms of speed as well. Speed is measured by the amount of the time taken by the tool to mine the given design pattern in the given C++ project.

Section 4.3.2 describes how the speed of the given tools is measured.

- *Memory usage.* Similar to the previous viewpoint, memory usage could be a problematic point of a design pattern miner tool. We measured the maximal memory required for the design pattern mining task. It was not straightforward because the memory usage of CrocoPat is fixed, and only the Columbus source code was available. The memory measuring method applied for the given tools is described in Section 4.3.3.

4.3 Results

We made a comparison on four open source small-to-huge systems, to make the benchmark results independent of system characteristics like size, complexity and application domain. These four real-life, freely available C++ projects are the following.

- *DC++ 0.687.* An open-source client for the Direct Connect Protocol that allows one to share files over the Internet with other users [24].
- *WinMerge 2.4.6.* An open-source visual text file differentiating and merging tool for Win32 platforms [93].
- *Jikes 1.22-1.* A compiler that translates Java source files (as defined in The Java Language Specification) into the byte-coded instruction set and binary format defined in The Java Virtual Machine Specification [47].
- *Mozilla 1.7.12.* All-in-one open source Internet application suite [58]. We used a checkout dated March 12, 2006.

Table 4.1 lists some statistics about the projects we analyzed. The first row shows how many source and header files were analyzed in the evaluated software systems. The second row lists the size of these source and header files in megabytes.

Size info.	DC++	WinMerge	Jikes	Mozilla
No. of files	338	512	74	11,325
Size (MB)	3	5.3	3	127
LOC	12,727	49,809	52,169	1,288,869
No. of classes	68	174	258	5,467

Table 4.1: About the size of each project studied

The last two rows were calculated via the metric plug-in of Columbus, and gives information about the total lines of code (LOC) and the number of classes. By *LOC* we mean every line in source code that is not empty and is not a comment line (also known as "logical lines of code").

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. In the next section we will describe our benchmark results and then analyze them in detail.

Now we will present our results concerning the differences between the design pattern candidates found, the running-time and the memory requirements. Below we will begin with the discovered pattern candidates, and then compare the time requirements for the tools. After, we will list the memory requirements of the design pattern mining tools.

4.3.1 Discovered pattern candidates

In this section we will describe our experiments related to pattern candidates found by the design pattern miner tools that we compared. Unfortunately, Maisa did not contain descriptions of the patterns Bridge, Chain of Responsibility, Decorator, State, Strategy and Template Method (the results for these are marked with dashes in our tables). We examined the differences between the tools manually, checking and comparing the candidates and the description of design patterns. First, we summarize our results on DC++ in Table 4.2.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	0	0	0
Adapter Class	0	2	0
Adapter Object	0	0	0
Bridge	0	-	0
Builder	0	0	0
Chain Of Responsibility	0	-	0
Decorator	0	-	0
Factory Method	0	0	0
Iterator	0	0	0
Mediator	0	0	0
Prototype	0	0	0
Proxy	0	0	0
Singleton	0	0	0
State	14	-	14
Strategy	14	-	14
Template Method	0	-	0
Visitor	0	0	0

Table 4.2: DC++ candidates

This was a small software package, so it did not contain too many design pattern candidates. Maisa found two Adapter Classes, while CrocoPat and Columbus found none. This is due to the fact that the definition of the Adapter Class in Maisa differed

from those in Columbus and CrocoPat. In Maisa, the Target participant class was not abstract and the Request method of the Target class was not pure virtual, while in Columbus and CrocoPat these features were requested. We examined the two Adapter Class candidates in Maisa, and we found that the Targets were not abstract in these cases and the Request operations were not pure virtual. Columbus and CrocoPat found 14 State and 14 Strategy design pattern candidates. The reason for the identical number of candidates is that the State and Strategy patterns have the same static structure, so their description in the tools were the same as well [8].

Table 4.3 shows the results of using the tools in the case of WinMerge. Maisa found two more Adapter Objects in WinMerge than Columbus. In the first case the difference was due to the fact that the Request method of a participant Adapter Object class was defined virtual in Columbus, while Maisa did not have this precondition. In the second case the pattern found in Maisa had a Target participant that was not abstract, which was a requirement in Columbus. When we relaxed the description of this pattern in Columbus, it found these two candidates too. The best solution would be if an exact definition existed for this pattern in both tools. CrocoPat found six Adapter Object candidates, while Columbus found only three.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	0	0	0
Adapter Class	0	0	0
Adapter Object	3	5	6
Bridge	0	-	0
Builder	0	1	0
Chain Of Responsibility	0	-	0
Decorator	0	-	0
Factory Method	0	0	0
Iterator	0	0	0
Mediator	0	0	0
Prototype	0	0	0
Proxy	0	0	0
Singleton	0	0	0
State	3	-	10
Strategy	3	-	10
Template Method	2	-	42
Visitor	0	0	0

Table 4.3: WinMerge candidates

The reason was that when Columbus found pattern candidates with certain participant classes in common, then *Columbus treated them as the same pattern candidate*. For example, if a design pattern (e.g. Strategy) contains a class with child classes (where these child classes could be of arbitrary number) then every repeated child class (e.g. ConcreteStrategy-s) and every repeated method (e.g. repeated AlgorithmInterface methods in each ConcreteStrategy) appears as a new candidate in the case of CrocoPat, while

Columbus grouped these candidates into one big candidate based on the common (e.g. Strategy) class. This is a very important difference between Columbus and CrocoPat, so we will stress this difference several times.

Maisa found a Builder in WinMerge but the two other tools did not, because in Maisa the Builder pattern representation did not contain the Director participant while the other two tools contained it. In the case of State, Strategy and Template Method the differences were because Columbus counted pattern candidates participating with certain classes in common only once, as in the case of Adapter Object.

Next, we will describe our experiments on design pattern candidates found in Jikes (see Table 4.4). Maisa found an Adapter Class, while Columbus and CrocoPat did not. The reason was the same as in the case of DC++, namely that in Maisa the Target participant class was not abstract and the Request method of the Target class was not pure virtual, but in Columbus and CrocoPat these features were required. In the case of Adapter Object Maisa failed to find a lot of candidates, while Columbus and CrocoPat discovered a lot of design pattern candidates. It appeared as if CrocoPat found more candidates because Columbus counted repeated pattern candidates with certain classes in common only once. In actual fact, these tools *found the same pattern candidates*.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	0	0	0
Adapter Class	0	1	0
Adapter Object	78	10	94
Bridge	0	-	0
Builder	0	1	0
Chain Of Responsibility	0	-	0
Decorator	0	-	0
Factory Method	0	0	0
Iterator	0	0	0
Mediator	0	4	0
Prototype	84	0	84
Proxy	53	74	66
Singleton	0	0	0
State	170	-	334
Strategy	170	-	334
Template Method	4	-	4
Visitor	0	23	0

Table 4.4: Jikes candidates

In Maisa, the Builder pattern representation did not contain the Director participant, so Maisa found an incomplete Builder candidate in Jikes. CrocoPat did not find any Mediator in Jikes, while Maisa found four. This is because Maisa described Mediator in a very special way, so that it contained a Mediator with two Colleagues, but Concrete Mediators were overlooked. The description of Mediator in CrocoPat required a Mediator

abstract class with a child ConcreteMediator class as well. In the case of Proxy, each tool discovered the same 53 candidates, but CrocoPat also counted repeating patterns with different methods and classes in 13 cases (66 altogether). Maisa found 21 more candidates more (74 altogether) because it did not need an abstract Proxy participant class in the Proxy design pattern. In the case of State and Strategy, it seems that Columbus found fewer design pattern candidates, but it counted each repeated pattern with certain classes in common only once. Maisa found 23 Visitor patterns which the other two did not. This is due to the permissive description of this pattern in Maisa.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	5	1	9
Adapter Class	0	59	0
Adapter Object	65	57	247
Bridge	880	-	1100
Builder	0	11	0
Factory Method	0	67	0
Mediator	0	2	0
Prototype	83	25	901
Proxy	0	1	0
Singleton	8	0	20
State	722	-	7662
Strategy	722	-	7662
Template Method	279	-	522
Visitor	0	30	0

Table 4.5: Mozilla candidates

Table 4.5 shows the results of our experiments with Mozilla. A lot of design pattern candidates were found, as in the case of State, where CrocoPat found 7662 and Columbus discovered 722 candidates. This huge difference was due to the fact that the design pattern candidates found were not grouped by CrocoPat, while Columbus grouped them. In the case of Adapter Class, the reasons for the differences were the same as in Jikes and in DC++ examined earlier. Columbus did not count repeated candidates in the case of Adapter Object, so it actually found the same candidates as CrocoPat, but Maisa missed some because of its different pattern description. CrocoPat and Columbus found the same candidates of the Bridge pattern, but Columbus counted the candidates with certain classes in common only once. Maisa found incomplete Builder candidates again, because the description of this pattern did not contain the Director participant class. Maisa found Factory Method candidates, while the other two did not. This is because the other two defined Factory Method with an abstract Product and an abstract Creator participant class, while Maisa did not require these participants to be abstract. CrocoPat did not find any Mediator candidate in Mozilla, but Maisa discovered two candidates. This is because Maisa described Mediator in a very special way, so it contained a Mediator with two Colleagues, but Concrete Mediators were missing. In the case of Prototype,

Singleton, State, Strategy and Template Method the differences were again due to the fact that CrocoPat counted each repeated pattern candidate, while Columbus counted these repeated ones with certain classes in common only once.

We have not explained every discrepancy, but we offered some of the more common reasons for each. 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. For example, the results of Maisa in WinMerge for the Builder pattern differed from those of CrocoPat and Columbus for just this reason.

- *Precision of pattern descriptions.* Another difference was how precise and strict the pattern descriptions were. For example, in the case of Jikes the differences in the numbers 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. Columbus and Maisa counted the repeated candidates with certain classes in common only once, but CrocoPat counted each occurrence.

The differences between the tools can be summarized in two points:

- *Intentional feature* is integrated into the tool to achieve better results. These features should not be modified in a comparison. For example, in Maisa the missing participant could be an *intentional feature* to discover more pattern instances.

- *The ad-hoc nature* of the tools creates unnecessary and confusing differences. These features should be standardized so as to have a common form or solution to facilitate a comparison. For example, in Maisa the missing participant might be ad-hoc as well.

The connection between above-mentioned differences and the intentional or ad-hoc features is the following. Differences caused by *different definitions of design patterns* could be intentional and ad-hoc as well. In this way, it is hard to determine whether it was ad-hoc or intentional. Hence it would be difficult to standardize and to eliminate this difference among the tools. However, based on our experiments it would be useful to have some kind of common design pattern definition repository, which would be worth studying. Differences caused by *precision of pattern descriptions* are quite similar to the previous one, and they arise for the same reasons.

Lastly, certain kinds of *differences in algorithms* could be standardized in a comparison of design pattern miner tools. For example, Columbus and Maisa counted the repeated candidates with certain classes in common only once while CrocoPat counted each occurrence. This difference should be handled and standardized in a fair comparison of candidate correctness. In chapters 5 and 7 a grouping method will be presented that can handle this difference.

4.3.2 Pattern mining speed

Now we will compare the speed of the three design pattern miner tools. We wanted to measure just the search time for patterns, hence we divided the running time into two parts, namely an initialization part and a pattern mining part.

Subject system	Columbus	Maisa	CrocoPat
DC++	00:00:03	00:00:00	00:00:03
WinMerge	00:00:08	00:00:03	00:00:11
Jikes	00:00:11	00:00:06	00:00:12
Mozilla	00:05:33	00:01:32	00:03:12

Table 4.6: Initialization times

Table 4.6 contains the initialization time of the tools (hh:mm:ss). For Maisa and CrocoPat we also measure the time required for exporting from the ASG representation into the tool-specific format (see Table 4.7).

Exporter Name	CAN2Maisa	CAN2CrocoPat
DC++	00:00:06	00:00:05
WinMerge	00:01:05	00:00:13
Jikes	00:00:13	00:00:10
Mozilla	00:06:11	00:05:47

Table 4.7: Initialization time - exporters

Tables 4.8, 4.9 and 4.10 only contain the pattern mining times. These times were measured in the following way:

- *Columbus.* We extended Columbus to report time statistics for the pattern mining procedure. We only considered the graph matching time; we did not include the time taken for ASG to load.
- *CrocoPat.* For CrocoPat, we made a small tool which executed CrocoPat and measured its running time. We measured the time needed for each pattern mining procedure for each subject software system. Next, we also measured the time for the subject systems with an empty RML program, because this way we could

measure the time necessary to reserve the memory and to prepare the BDD representation (initialization time). After, we subtracted the initialization time from the full running time for each result, and this way obtained the pattern matching times.

- *Maisa.* Maisa created statistics for each pattern mining procedure, which contained information about the time necessary for pattern mining and for the initialization, so we used these generated statistics.

First, we will list our results for DC++. In this case the required time was very small for each assessed pattern miner tool (lower than one second), hence they can be treated as being practically identical. This is due to the small size of the DC++ system, hence the design pattern candidates were discovered very quickly in this system by all three tools.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	00:00:00	00:00:02	00:00:07
Adapter Class	00:00:00	00:00:00	00:00:07
Adapter Object	00:00:00	00:00:17	00:00:08
Bridge	00:00:00	-	00:00:08
Builder	00:00:00	00:00:24	00:00:09
Chain Of Responsibility	00:00:00	-	00:00:08
Decorator	00:00:00	-	00:00:09
Factory Method	00:00:00	00:00:02	00:00:01
Iterator	00:00:00	00:00:21	00:00:01
Mediator	00:00:00	00:00:24	00:00:04
Prototype	00:00:00	00:00:02	00:00:08
Proxy	00:00:00	00:00:22	00:00:14
Singleton	00:00:00	00:00:00	00:00:07
State	00:00:03	-	00:00:08
Strategy	00:00:03	-	00:00:08
Template Method	00:00:01	-	00:00:06
Visitor	00:00:00	00:00:00	00:00:05

Table 4.8: WinMerge times

In the case of WinMerge (see Table 4.8), Columbus was the best tool. This is due to the fact that the Columbus design pattern mining algorithm filters out several classes at the beginning of the process and WinMerge has a relatively small number of classes. In contrast, with Jikes and Mozilla, Columbus could not filter out as many classes at the beginning of the process so it achieved worse results than those of the other two.

The time requirements for discovering patterns in Jikes is shown on Table 4.9. Columbus was very fast in the case of larger patterns (e.g. Abstract Factory, Visitor), because it could filter out [8] a lot of class candidates at the beginning of the discovering process. However, Columbus was slower in the case of smaller patterns (e.g. Template Method) because here a lot of class candidates remained for the detailed discovery process. The time requirements for CrocoPat and Maisa were rather similar.

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	00:00:00	00:00:06	00:00:09
Adapter Class	00:00:00	00:00:04	00:00:07
Adapter Object	00:00:09	00:00:11	00:00:07
Bridge	00:00:00	-	00:00:06
Builder	00:00:08	00:00:59	00:00:07
Chain Of Responsibility	00:00:00	-	00:00:07
Decorator	00:00:00	-	00:00:18
Factory Method	00:00:00	00:00:04	00:00:02
Iterator	00:00:00	00:00:55	00:00:07
Mediator	00:00:00	00:01:05	00:00:12
Prototype	00:04:18	00:00:04	00:00:07
Proxy	00:00:00	00:01:03	00:00:13
Singleton	00:00:00	00:00:00	00:00:11
State	00:04:48	-	00:00:12
Strategy	00:04:48	-	00:00:12
Template Method	00:03:55	-	00:00:06
Visitor	00:00:00	00:00:06	00:00:14

Table 4.9: Jikes times

Design Pattern	Columbus	Maisa	CrocoPat
Abstract Factory	00:02:32	00:18:21	00:13:43
Adapter Class	00:00:06	00:18:34	00:14:34
Adapter Object	00:04:41	03:07:03	00:13:42
Bridge	04:50:29	-	00:17:20
Builder	01:39:09	04:09:22	00:14:02
Chain Of Responsibility	00:00:07	-	00:13:33
Decorator	00:00:18	-	00:27:40
Factory Method	00:03:03	00:15:56	00:00:02
Iterator	00:00:07	03:54:12	00:14:19
Mediator	00:48:03	04:19:07	00:18:10
Prototype	01:24:55	00:14:21	00:25:49
Proxy	00:00:07	04:41:47	00:27:10
Singleton	00:00:02	00:00:00	00:13:17
State	04:09:20	-	00:20:22
Strategy	04:09:20	-	00:20:22
Template Method	00:14:46	-	00:13:27
Visitor	00:00:07	00:06:56	00:20:45

Table 4.10: Mozilla times

Table 4.10 lists our results for Mozilla. In most cases, CrocoPat produced the best results, but in certain cases Columbus and Maisa were faster. Columbus was slow when it was only able to filter out a small number of class candidates at the beginning of the discovery process. The CSP algorithm of Maisa was also slow here.

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.

4.3.3 Memory requirements

Now we will compare the memory usage of the three design pattern miner tools. We measured the memory requirements of each design pattern mining procedure, but we will summarize our results in one table because we found them to be very similar.

The memory measurement method for the three systems was carried out in the following way:

- *Columbus.* We extended the tool at the source code level so that it reported data about its memory usage automatically.
- *Maisa.* Maisa did not report its memory usage, so we measured it by simply monitoring its peak memory usage on the task manager.
- *CrocoPat.* CrocoPat's memory usage is constant and can be set as a command line parameter. Therefore, we executed CrocoPat between 1 megabyte to 200 megabytes of reserved memory for each pattern mining process (it is based on a binary search algorithm). After, we took the smallest case where CrocoPat still terminated normally.

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. This is true for each pattern miner tool, as can be seen in Table 4.11.

Subject system	Columbus	Maisa	CrocoPat
DC++	37 (19)	10-11	2-3
WinMerge	71 (32)	13-14	10-11
Jikes	51 (26)	13-17	10-14
Mozilla	866 (330)	60-71	125-175
Average	256 (102)	24-28	37-51

Table 4.11: Memory requirements in megabytes

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 (including detailed facts about statements and expressions not needed for design pattern detection)

for all kinds of tasks. Right now, for technical reasons, the design pattern miner plug-in of Columbus does not work without the ASG (although it does not need it), but we would like to fix this problem. Hence, we also measured the memory needed by Columbus without the ASG and listed these values in parentheses in Table 4.11.

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.

After viewing Table 4.11 we may conclude that, in the terms of memory requirements, Maisa's performance was the best.

4.4 Summary

In this chapter we compared three pattern miner tools, namely Columbus, Maisa and CrocoPat. We compared them in terms of differences of pattern candidates, speed and memory consumption. We provided a common input for the tools by analyzing the source code with the front end of Columbus and with plug-ins for generating the required files for the tools. This way, as a "side effect" of this study, we extended our Columbus Reverse Engineering Framework with a plug-in for CrocoPat. From the results of experiments we found that CrocoPat is generally the fastest, Maisa needs the least memory and Columbus is the fastest in the case of mining very complex patterns.

The findings here are three-fold. First of all, we experimentally learned and tested our preliminary assumptions about the causes of the differences between the results of the tools. These results support the grouping mechanism that will be presented in chapters 5 and 7. Second, it is clear which tool should be used in certain circumstances in terms of speed or memory consumption. Columbus is the fastest in the case of complex patterns or in the case of small or medium-sized systems. CrocoPat is the fastest in the case of simpler patterns or in the case of large sized systems, while Maisa can used if just a small amount of memory is available.

During this study we addressed the problem of comparing the results of different kinds of tools. Different result formats have to be processed and the appropriate source code fragments have to be located and the results of a comparison need to be stored. We performed this manually, which was a quite dull and time-consuming task. As a solution to this laborious manual work, we will introduce DEEBEE in the next chapter.

> *"The art and science of asking the right questions*
> *is the source of all knowledge."*
> *Thomas Berger*

Chapter 5

Validation of design pattern miner tools

The problem with most approaches of pattern recognition (based on pattern matching) is that they are insufficiently strict in the sense that *they produce many false results*, in which some code fragments are identified as pattern candidates that have only the structure of the pattern description. The pattern miner tools usually employ different definitions of patterns, different algorithms for detecting these patterns, and they present their results in different output formats, as shown in the previous chapters. Another major problem is that *there is no standard measurement system or a test database* to evaluate and compare results produced by design pattern miner tools.

These problems make it difficult to evaluate and compare design pattern miner tools. We also observed that there was a desire in conferences and publications for a means of evaluating patterns easily and effectively [63]. Hence, *we developed a publicly available benchmark*, 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.

In this chapter our benchmark will be introduced and experiments will be described that evaluated and compared two design pattern miner tools, namely *Maisa* and *Columbus*. The tools were evaluated on reference implementations of design patterns, on a software system called FormulaManager, which contains every design pattern in a real context, and on an open source software system called NotePad++. The reference implementations and FormulaManager were implemented by us. In addition, pattern instances from NotePad++ recovered by professional software developers were added to the benchmark.

5.1 Benchmark

We shall use the well-known issue and bug tracking system called Trac [83] (version 0.9.6) as the basis for the benchmark. Trac was written in Python and it is an easily extendible and customizable plug-in oriented system.

5.1.1 Architecture

The core of the Trac system contains a documentation subsystem (Wiki), and modules for source browsing, timeline, roadmap, permission handling, issue tracking and so on. The goal of the Wiki subsystem is to make text editing easier and to ensure text content for a project. Timeline lists all Trac events that have occurred in chronological order and gives a brief description of each event. Permission handling is a customizable permission method where different permission types can be configured. Issue tracking is based on *tickets*, where a ticket stores information about an issue or a bug. Roadmap offers a view of the ticket system that helps planning and managing the future development of a project. Trac has a lot of other functionalities too, but we will not list them here. Figure 5.1 shows the architecture of the Trac system adapted to our needs.

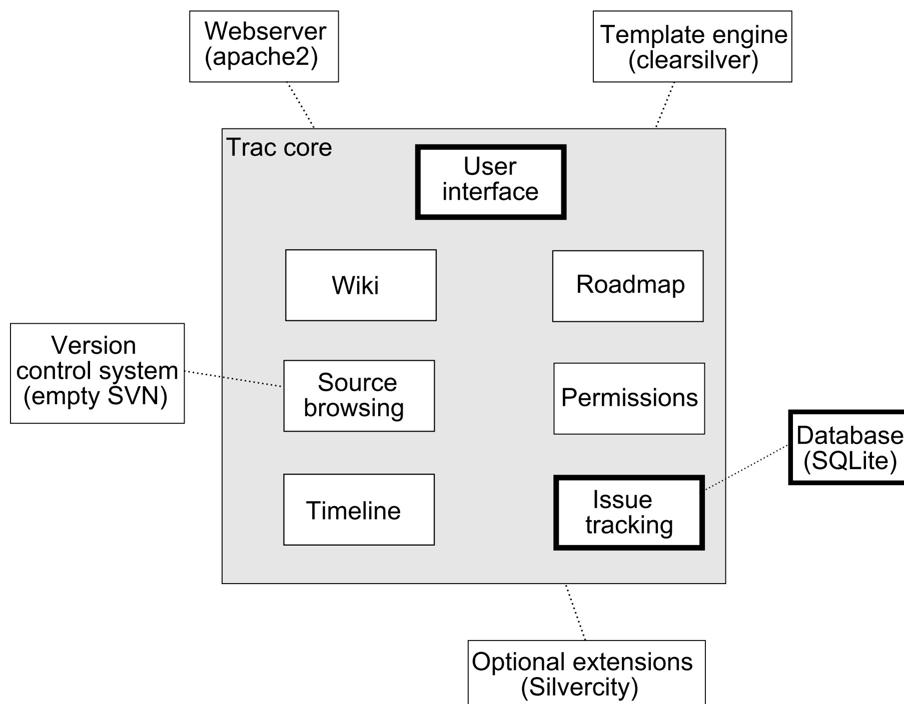

Figure 5.1: Architecture of Trac

Trac requires some external components, and other optional components can also be included. The required components are a version control system that hosts the source

code of the project, an SQL database that stores the tickets, a Web server where Trac is hosted and a template engine that is used to generate dynamic Web pages. We used an empty Subversion repository as a version control system of Trac, because we did not use this feature here. We used SQLite as the database component and Apache2 as the Web server. Because Web pages have to be dynamically generated, Trac employs a template engine. For this purpose we made use of the ClearSilver template engine. Furthermore, we integrated the SilverCity component for syntax highlighting, which currently supports highlighting C++ and Java source code.

Although the Trac system provides many useful services, we still had to customize and extend it to make a suitable benchmark from it. The two major extensions were the customization of the graphical user interface and the system's tickets. In the case of the former we had to inherit and implement some core classes of the Trac system. The customized graphical interface will be presented through usage scenarios in Section 5.2. In the case of the tickets, we had to extend them to be able to describe design pattern candidates (pattern name, information about its participants, evaluation data of the candidate). In Figure 5.1 the customized components are drawn with thick borders.

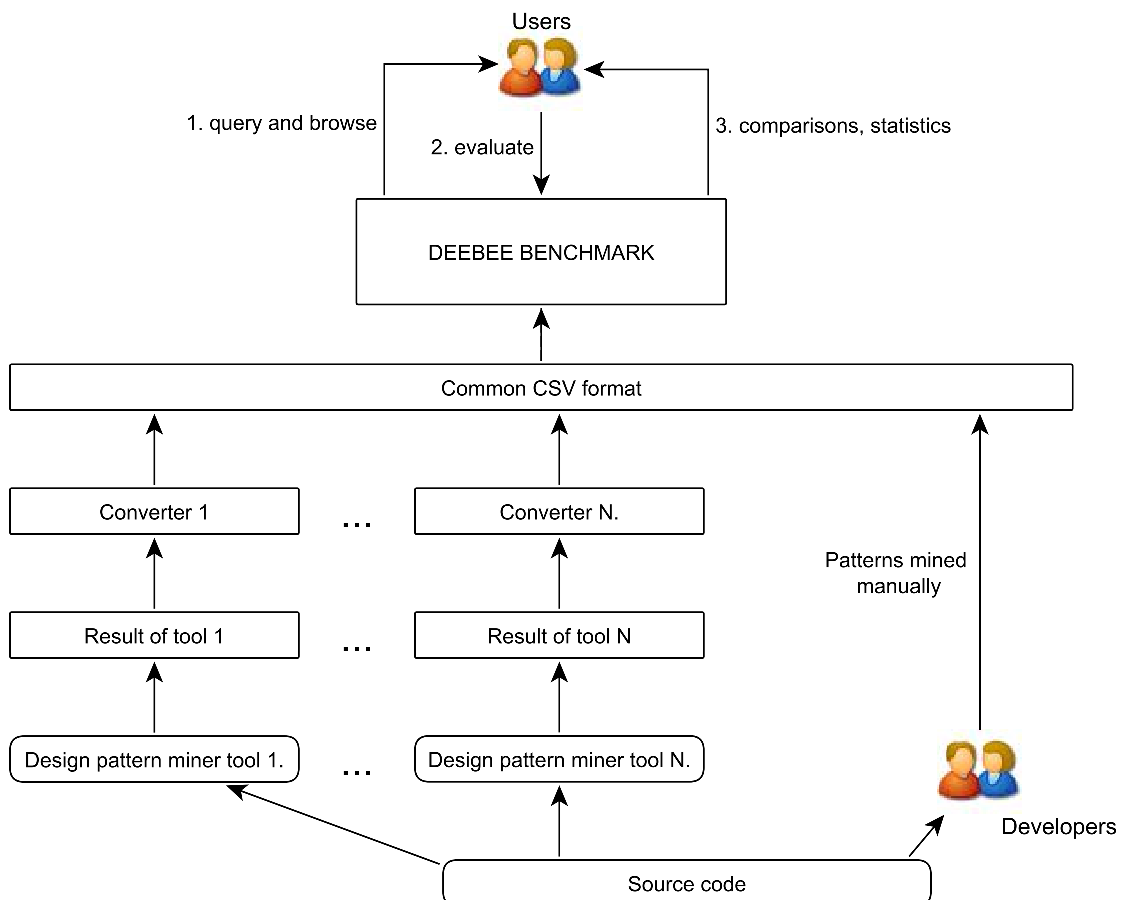

Figure 5.2: Overview of DEEBEE

Figure 5.2 shows an overview of the benchmark. First, design pattern miners 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). Developers manually discover design pattern candidates and then store their results in the DEEBEE CSV file format. In the next step the CSV file(s) are uploaded into the benchmark and then the candidates can be evaluated, compared and browsed via the online interface of DEEBEE.

5.1.2 Fundamental participants and siblings

There are some cases when design pattern miner tools report the same design pattern candidate, but they report them differently. As shown in Chapter 4, the results of the different design pattern miner tools may differ for several reasons. The main reasons for this are:

- *Different definitions of design patterns:* for example, when some tools leave out a participant from the definition of the searched design pattern, while the others do not.
- *Precision of pattern descriptions:* for instance, when some tools define a participant as abstract, while the others do not.
- *Differences in algorithms:* for example, when some tools report design patterns with some kind of grouping, while the others do not.

These points make a sound comparison of the results of different tools quite difficult. In the following 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* (mandatory roles) of design patterns. For example, in the case of the State pattern [39] (see Figure 2.3) the fundamental participant plays the (mandatory) role of the State class, while the other participant classes can be repeated.

In order to demonstrate siblings and fundamental participants, let us consider the following example. Figure 5.3 shows an example for the State design pattern taken from the book by Gamma et. al. [39]. The background for this example was:

> *"Consider a class TCPConnection that represents a network connection. A TCPConnection object can be in one of several different states: Established, Listening, Closed. When a TCPConnection object receives requests from other objects, it responds differently depending on its current state."* [39]

Suppose that three given design pattern miner tools (T1,T2 and T3) discover this pattern instance, but they report it differently. These tools report their results in the form of *(participant, sourceElement)* pairs, where *participant* denotes a certain participant of the currently searched design pattern, while *sourceElement* denotes the appropriate source code element discovered by the tool. T1 reports three design pattern candidates:

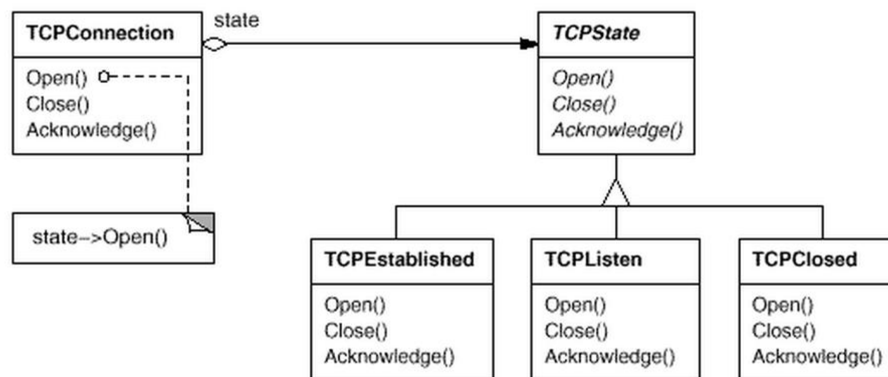

Figure 5.3: State example

- T1-1: (Context, TCPConnection), (State, TCPState), (ConcreteState, TCPEstablished)
- T1-2: (Context, TCPConnection), (State, TCPState), (ConcreteState, TCPListen)
- T1-3: (Context, TCPConnection), (State, TCPState), (ConcreteState, TCPClosed)

In contrast to T1, the other two tools (T2 and T3) report just one pattern instance. However, the results of T2 and T3 are different as well:

- T2-1: (Context, TCPConnection), (State, TCPState), (ConcreteState, TCPEstablished), (ConcreteState, TCPListen), (ConcreteState, TCPClosed)
- T3-1: (State, TCPState), (ConcreteState, TCPEstablished), (ConcreteState, TCPListen), (ConcreteState, TCPClosed)

The difference between T1 and the other two is caused by the fact that the latter two group the pattern instances, while the former does not. The results of T2 and T3 differ because T3 represents the State pattern without Context. However, for a valid comparison these results need to handled together in some way.

The key is to find some common point in these cases that is sufficient to relate these differently reported cases. In the case of each design pattern there are certain *fundamental participants* that typically occur in each case of design pattern candidates because the design pattern would be meaningless without that participant. For example, with the current example the State class (TCPState) occurs for each candidate (T1-1, T1-2, T1-3, T2-1 and T3-1). Based on the common fundamental participant, the differently reported but identical candidates can be related to each other and are referred to as *siblings*. In the following we will explain why it is necessary to evaluate and compare the tools by taking into account the sibling relation and *grouping* the candidates by their sibling relations.

Tools T1 and T2 also detect a false candidate that will be denoted by T1-4 and T2-2:

- T1-4: (Context, FalseContext), (State, FalseState), (ConcreteState, FalseConcreteState)
- T2-2: (State, FalseState), (ConcreteState, FalseConcreteState)

Hence several true candidates (T1-1, T1-2, T1-3, T2-1 and T3-1) were discovered by using the tools and some false candidates were also discovered (T1-4 and T2-2). Evaluating the tools *without grouping* their findings yields the following results:

- T1 originally discovers 4 candidates. The T1-1,T1-2 and T1-3 candidates are true, while only T4-4 is false. This means that the precision for T1 is 75%.
- T2 discovers one true (T2-1) and one false candidate (T2-2). This means that the precision for T2 is 50%.
- T3 discovers only one true (T3-1) candidate so its precision score is 100%.

From a precision viewpoint, the best tool is T3 (100%), the second one is T1 (75%) and the third one is T2 (50%). However, T1 is better than T2 because T1 reports a new candidate in the case of each ConcreteState participant. In a fair comparison we need to consider the sibling relations and *group* the findings of T1. Then the T1 precision score changes to 50% because T1-1, T1-2 and T1-3 are grouped into one candidate. In this way the precision scores of T1 and T2 become the same.

Furthermore, when comparing the original findings of tools T1, T2 and T3, the intersection of their results is empty, which is misleading. In contrast, if the grouping is performed on all candidates by their sibling relations then T1-1, T1-2, T1-3, T2-1 and T3-1 will be grouped into one common candidate and the intersection of the tools will be this one common candidate, as it should be. In summary, grouping candidates by their sibling relations has three use cases:

- The evaluation of a tool requires grouping its candidates, as in the case of T1: T1-1, T1-2 and T1-3 were grouped in the previous example.
- Comparing the tools requires grouping their candidates like that shown in the previous example. (T1-1, T1-2, T1-3, T2-1 and T3-1 were grouped).
- Developers who evaluate design pattern candidates first have to understand the current candidate context and connections in depth by examining the appropriate source code fragments. This is the most time-consuming part of the evaluation. Evaluating sibling candidates one after the other can save a lot of time because the context only has to be understood for the first candidate.

5.1.3 Upload file format.

The format of the DEEBEE's CSV file containing the design pattern candidates must be the following (see Figure 5.4). Each pattern candidate consists of several lines terminated by a blank line. The first line of each candidate is always the pattern name,

without any white space characters. If the name consists of several words, then they have to be concatenated and each new word should start with an upper case letter (CamelCase). The subsequent lines represent the participants of the candidate. These lines contain values separated by commas. The first value contains the role that may be *class*, *operation* and *attribute*. If the participant is fundamental, then a star has to be given before the role in the first value (see the *class State* in Figure 5.4). The second value contains the name of the participant as written the source code. Lastly, the third value contains the relative path of the participant in the source code together with starting and ending line information separated by colons.

```
State
class Context,TCPConnection,TCPConnection.h:1:6
operation Request,Open,TCPConnection.cpp:3:15
operation Request,Close,TCPConnection.cpp:15:25
operation Request,Acknowledge,TCPConnection.cpp:25:32
*class State,TCPState,TCPState.h:1:6
operation Handle,Open,TCPState.cpp:3:15
operation Handle,Close,TCPState.cpp:15:19
operation Handle,Acknowledge,TCPState.cpp:19:23
class ConcreteState,TCPListen,TCPListen.h:1:6
operation Handle,Open,TCPListen.cpp:3:12
operation Handle,Close,TCPListen.cpp:12:18
operation Handle,Acknowledge,TCPListen.cpp:18:23
```

Figure 5.4: CSV file for the T1-2 candidate of the previous example

The uploaded pattern candidates are checked to see if they exist in the database. If a known pattern candidate is found again by the new tool, and the two candidates are exactly the same, then the tool entry of the candidate is extended by the name of the new tool, and no new candidate will be created in the database. If only the fundamental participants are the same, then a new candidate will be created and the candidates will be related as siblings.

5.1.4 Benchmark contents

The benchmark contains 1,274 design pattern candidates from three C++ software systems (Mozilla [58] NotePad++ [60] and FormulaManager [81]), three Java software systems (JHotDraw [46], JRefactory [48] and JUnit [49]) and C++ reference implementations of design patterns. The uploaded design pattern candidates are recovered by three design pattern miner tools: Columbus (C++), Maisa (C++) and Design Pattern

Detection Tool (Java) [84] [27]. Table 5.1 shows the distribution of the candidates among tools and subject systems[1].

Subject system	Columbus 3.5	Maisa 0.5	DPD3	Human
Mozilla1.8a5	495	-	-	-
NotePad++3.9	13	27	-	11
ReferenceC++	15	12	-	24
JHotDraw	-	-	322	-
JRefactory	-	-	259	-
JUnit	-	-	7	-
FormulaManager	56	32	-	40

Table 5.1: Distribution of uploaded candidates

5.2 Usage scenarios

Before describing the usage scenarios, we will provide an overview of the benchmark's functionalities. The benchmark contains three main menu points, namely *evaluation*, *upload* and *register*. From the evaluation menu point, three important views can be accessed. These are the *statistics view*, *comparison view* and the *instance view*. From the upload menu point, the *new language*, *new software*, *new tool* and *new instances* functionalities can be accessed. Figure 5.5 shows a map of the views and functionalities of the benchmark (e.g. query view can be accessed from evaluation menu, while comparison view can be accessed from statistics view).

5.2.1 Browsing the database

In this scenario we will present different queries and views with which the content of the benchmark database can be readily explored. In each case we will first give a brief description of the view or functionality, and afterwards demonstrate it by supplying an example.

Query view. As the primary aim of our benchmark is to evaluate and classify the results of design pattern miner tools (the design pattern candidates), the first thing one should do is to decide how the candidates are to be listed, which is controlled by aspects. There are four aspects in the benchmark, namely *software*, *tool*, *pattern* and *language* which can be adjusted in any particular order (it is not necessary to set every aspect). For each aspect, different parameters can be set which influence the result. The number

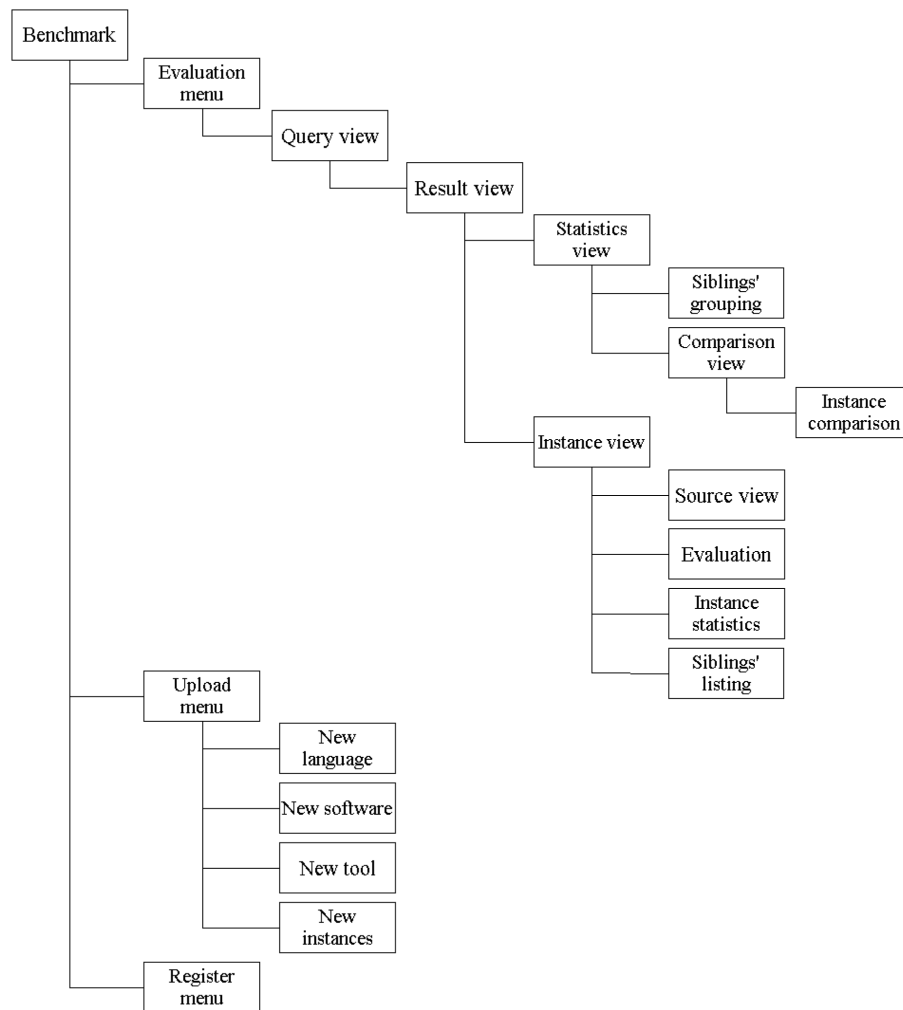

Figure 5.5: Functionalities of the benchmark

Please make your selection.

1. aspect: Language C++
2. aspect: Software NotePad++3.9
3. aspect: Tool All
4. aspect: Pattern All

Go to results view

Figure 5.6: Query view of DEEBEE

of possible parameter values will increase during the evolution of the benchmark because new patterns, languages, software packages and tools will be added.

[1]The tools found same candidates, so summarizing the values in the table will be greater than 1,274

Example: Figure 5.6 shows a screenshot of the query view. This example gives an ordering where the first aspect is the *language* with the parameter set to *C++*. The second aspect is *software* set to *NotePad++*. The following two aspects are *tool* and *pattern* with parameters set to *all*. By pressing the button *"Go to results view"*, the results view will be generated.

Results view. After running query, the results table is generated (see Figure 5.7). The columns (except the last one) correspond to the previously selected aspects. The last column contains the identifiers of the design pattern candidates which satisfy the criteria provided by the aspects. By clicking on one of the candidate identifiers, the *Instance view* will show up, while by clicking on the links in the other columns, the *Statistics view* will be accessed.

Language	Software	Tool	Pattern	Pattern instance ids
C++	NotePad++3.9	Human	Prototype	#1134
			Composite	#1131
			Proxy	#1129 #1130
			TemplateMethod	#1135
			Singleton	#1126 #1127
			Observer	#1128
			Iterator	#1125
			Facade	#1133
			Interpreter	#1132
		Columbus3.5	AdapterObject	#29 #30 #31 #32 #33 #34 #35 #36 #37
			State	#38
			TemplateMethod	#39 #40 #41
		Maisa0.5	AdapterObject	#6 #7 #8 #9 #10
			Prototype	#12 #13 #14 #15 #16 #17
			Proxy	#18 #19 #20 #21 #22 #23 #24 #25 #26 #27 #28
			Builder	#11
			AdapterClass	#2 #3 #4 #5

Figure 5.7: Results view of DEEBEE

Example: Based on the previously selected ordering aspects in the query view, we got a table where the top left link is C++ (see Figure 5.7). The second column contains a link to the NotePad++ software in accordance with the query. Because we queried all tools and patterns (third and fourth aspects), we got all the tools which currently participate in the benchmark and all pattern candidates found by these tools in NotePad++.

Instance view. The instance view provides the user interface for evaluating design pattern candidates (see Figure 5.8). The participants of the actual pattern candidate are shown in the top left corner with their concrete name in the source code. When the user clicks on a participant name, the source code is shown with the pattern candidate highlighted on the right hand side of the view. A design pattern candidate (instance) may be evaluated using two categories:

Instance view of #32

Back to previous view

Pattern Instance Information

Tool	Columbus3.5
Software	NotePad++3.9
Pattern	AdapterObject
Participants	

*class Target	StaticDialog
operation Request	run_dlgProc
class Adaptee	ColourPicker
operation SpecificRequest	destroy
class Adapter	WordStyleDlg
operation Request	run_dlgProc
attribute adaptee	_pFgColour

How complete is it? stat

- (•) The pattern instance is complete in every aspect.(100%)
- ( ) Some participants are missing from the pattern instance.(50%)
- ( ) Important participants are missing from the pattern instance.(0%)
- [ ] *Apply this answer to siblings too.*

submit

How correct is it? stat

- ( ) I am sure that it is a real pattern instance.(100%)
- ( ) I think that it is a real pattern instance.(66%)
- (•) I think that it is not a real pattern instance.(33%)
- ( ) I am sure that it is not a real pattern instance.(0%)
- [ ] *Apply this answer to siblings too.*

submit

Siblings:

#6 ; #7 ; #8 ; #9 ; #31 ; #33 ; #34 ; #35 ; #36 ;

submit comment

/PowerEditor/src/WinControls/StaticDialog/StaticDialog.h(41)

```
enum PosAlign{ALIGNPOS_LEFT, ALIGNPOS_RIGHT, ALIGNPOS_TOP, ALIGNPOS_BOTTOM};

struct DLGTEMPLATEEX {
      WORD   dlgVer;
      WORD   signature;
      DWORD  helpID;
      DWORD  exStyle;
      DWORD  style;
      WORD   cDlgItems;
      short  x;
      short  y;
      short  cx;
      short  cy;
      // The structure has more fields but are variable length
} ;

class StaticDialog : public Window
{
public :
        StaticDialog() : Window() {};
        ~StaticDialog(){
                if (isCreated())
                        destroy();
        };
        virtual void create(int dialogID, bool isRTL = false);

    virtual bool isCreated() const {
                return (_hSelf != NULL);
        };

        void goToCenter();
    void destroy() {
                ::SendMessage(_hParent, WM_MODELESSDIALOG, MODELESSDIALOGREMOVE, (WPARAM)_h

                ::DestroyWindow(_hSelf);
        };

protected :
        RECT _rc;
        static BOOL CALLBACK dlgProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
        virtual BOOL CALLBACK run_dlgProc(UINT message, WPARAM wParam, LPARAM lParam) = 0;

    void alignWith(HWND handle, HWND handle2Align, PosAlign pos, POINT & point);
        HGLOBAL makeRTLResource(int dialogID, DLGTEMPLATE **ppMyDlgTemplate);
};

#endif //STATIC_DIALOG_H
```

Figure 5.8: Instance view with highlighted source code in DEEBEE

- *Completeness* means how complete the evaluated pattern candidate is in a structural sense. More precisely, it means how many pattern participants can be found in the candidate. In this category, the possible answers are the following:
  - *The pattern instance is complete in every aspect.*
  - *Some participants are missing from the pattern instance.*
  - *Important participants are missing from the pattern instance.*
- *Correctness* means how correct the evaluated pattern candidate is in a behavioural sense. More precisely, it means to what degree the pattern candidate matches the original intent of the design pattern. In this category, the possible answers are the following:
  - *I am sure that it is a real pattern instance.*
  - *I think that it is a real pattern instance.*

– *I think that it is not a real pattern instance.*

– *I am sure that it is not a real pattern instance.*

The evaluation answers, Completeness and Correctness can be applied to the *siblings* of the pattern candidate as well (if there are any) by selecting the corresponding checkboxes. Most of the time sibling candidates are evaluated equally, hence the siblings of the pattern candidate are also listed in the instance view to help the user in the evaluation.

It can also be queried for statistics about the previous evaluations of the pattern candidate. The result will appear on the right hand side of the view (see Figure 5.9). User comments can be added to the pattern candidate at the bottom of the instance view. Registration needs to be performed to evaluate a pattern candidate and to add a comment.

Example: We shall continue with the previous example. After clicking on candidate #32 in the results view (see Figure 5.7), the instance view will show up (see Figure 5.8). If we select the participant called *StaticDialog*, then its syntax-highlighted source code will be automatically loaded into the right hand side of the view, which can be evaluated immediately inside the benchmark environment. Statistics about past evaluations are available for completeness and correctness by clicking on the *stat* links next to the corresponding questions. In the case of pattern candidate #32, both candidate statistics results are shown in Figure 5.9.

Completeness stat:

Username	Value
vadam	100%
iarpad	100%
flajos	100%

Aspect	Value
Mean	100.0%
Deviation	0.0%
Min	100.0%
Max	100.0%
Median	100.0%

Correctness stat:

Username	Value
vadam	33%
iarpad	66%
flajos	33%

Aspect	Value
Mean	44.0%
Deviation	19.05%
Min	33.0%
Max	66.0%
Median	33.0%

Figure 5.9: Instance view statistics in DEEBEE

Statistics view. The statistics view gives statistical data about the entity clicked in the results view (see Figure 5.7) together with all other entities which depend on it. The view contains two tables - one for *correctness* and one for *completeness*, respectively (see Figure 5.10).

Each line in the upper part of each table contains a statistic for one particular pattern candidate. A pattern candidate may be evaluated by some people, so the mean, deviation, minimum, maximum and median values are calculated for the candidate. The lower part of each table contains the same kind of basic statistics, but they are calculated from the statistics of the candidates (shown in the upper part of each table).

C++/NotePad++3.9/Maisa0.5/Proxy statistics

Correctness

	Mean	Deviation	Min	Max	Median
#18	16.5%	23.33%	0.0%	33.0%	16.5%
#19	66.0%	0.0%	66.0%	66.0%	66.0%
#23	16.5%	23.33%	0.0%	33.0%	16.5%
#24	16.5%	23.33%	0.0%	33.0%	16.5%
#25	16.5%	23.33%	0.0%	33.0%	16.5%
#28	16.5%	23.33%	0.0%	33.0%	16.5%
Mean	**24.75%**	**19.44%**	**11.0%**	**38.5%**	**24.75%**
Deviation	**20.21%**	**9.52%**	**26.94%**	**13.47%**	**20.21%**
Min	**16.5%**	**0.0%**	**0.0%**	**33.0%**	**16.5%**
Max	**66.0%**	**23.33%**	**66.0%**	**66.0%**	**66.0%**
Median	**16.5%**	**23.33%**	**0.0%**	**33.0%**	**16.5%**

Summary

Number of pattern instances:	6
Number of evaluated pattern instances:	6
Number of pattern instances above the threshold:	1
Precision:	16.67%
Total number of pattern instances:	8
Total number of evaluated pattern instances:	8
Total number of pattern instances above the threshold:	2
Recall:	50.0%

Completeness

	Mean	Deviation	Min	Max	Median
#18	100.0%	0.0%	100.0%	100.0%	100.0%
#19	100.0%	0.0%	100.0%	100.0%	100.0%
#23	100.0%	0.0%	100.0%	100.0%	100.0%
#24	100.0%	0.0%	100.0%	100.0%	100.0%
#25	100.0%	0.0%	100.0%	100.0%	100.0%
#28	100.0%	0.0%	100.0%	100.0%	100.0%
Mean	**100.0%**	**0.0%**	**100.0%**	**100.0%**	**100.0%**
Deviation	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.0%**
Min	**100.0%**	**0.0%**	**100.0%**	**100.0%**	**100.0%**
Max	**100.0%**	**0.0%**	**100.0%**	**100.0%**	**100.0%**
Median	**100.0%**	**0.0%**	**100.0%**	**100.0%**	**100.0%**

Figure 5.10: Statistics view of DEEBEE

Below, the correctness table *precision* and *recall* are given[2]. These values are based on a human evaluation of the candidates and on a threshold value, which by default is 50%. This means that a pattern candidate found by a tool is considered to be an instance (true positive) if the average score of the evaluation of its correctness is at least 50%. The benchmark can be customized to take into account sibling relations or not when generating statistics. The default option is to handle sibling candidates as only one common candidate. If the sibling relations are considered, then the least candidate identifier will denote the grouped candidates and it is marked in red instead of blue.

Example: Go from the instance view (Figure 5.8) to the results view (Figure 5.7) by clicking on the "Back to previous view" link. On the results view interface, click on the *C++/NotePad++3.9/Maisa0.5/Proxy* link in the table. This takes us to the statistics view, which is shown in Figure 5.10.

5.2.2 Evaluating and comparing tools

In this scenario we will show how design pattern miner tools can be evaluated and compared using our benchmark. The example gives the evaluation and comparison of Columbus 3.5 and Maisa 0.5 on reference implementations of the GOF patterns in C++.

First, the Evaluation menu has to be selected and the following query should be set (see Figure 5.6):

- 1. aspect: "Software" with parameter set to "ReferenceC++".
- 2. aspect: "Tool" with parameter "All".

[2]The definition of precision and of recall are both given in Section 2.2.

- 3. aspect: “None” with parameter “All”.
- 4. aspect: “None” with parameter “All”.

Pressing the *“Go to results view”* button will generate the results view (which is similar to Figure 5.7).

In the results view by clicking on the “ReferenceC++” link in the top left corner we got the statistics view (which is similar to Figure 5.10). Here we can see the statistics on the results of Columbus and Maisa. In the case of Columbus, the precision score is 100% and the recall score is 58.33%. In the case of Maisa these are 80% and 33.33%, respectively. Here we used the default threshold value of 50% (see Section 5.2.1).

The benchmark provides another interesting view, called *comparison view*, for comparing the capabilities of pattern miner tools. The page can be accessed from the statistics view by clicking on the *Switch to comparison view* link. The first table in Figure 5.11 shows the pattern candidates found by Maisa and by Columbus in the reference implementation of the GOF patterns (rows A and B). This table also shows the differences and the intersection of the candidates found by the tools which were compared (rows A-B, B-A and A&B). It can be seen in row A-B that Maisa found one pattern candidate (#1152, a Memento candidate), which was not found by Columbus. Row B-A shows 7 candidates found by Columbus and not found by Maisa. Row A&B shows those 7 candidates which are found by both tools.

The comparison view contains two other tables showing the results of a comparison of Maisa and Columbus with the results of human inspection of the code, respectively. The instances found by careful human inspection represent the desirable results of a tool. Therefore, the difference between e.g. Human and Columbus represents the *false negatives* for Columbus. (This value is used for calculating the recall values.) This way the false negatives of a tool can be quickly and efficiently discovered using the benchmark.

5.2.3 Adding a new tool

In this scenario we will show how the results of a new design pattern miner tool can be added to the benchmark. Uploading the results of a tool requires the *the tool name*, *name of the mined software*, the software *programming language*, *source location*, and information about the *design pattern candidates* found in the comma separated value (CSV) file format (see Figure 5.12 and Section 5.1.3).

To upload design pattern candidates, public access to the source code of the mined software has to be given. This is important because the source code is used in the instance view to show and evaluate pattern participants. There are two possible ways to provide access to the source code: via *http* and *svn*.

Do you want to group siblings?

Yes

Threshold for true instances.

50 % Set

No threshold (every pattern instance)

A = ReferenceC++/Maisa0.5
B = ReferenceC++/Columbus3.5

A	#1152	#1154	#1155	#1160	#1138	#1139	#1141	#1173		
B	#1154	#1155	#1157	#1158	#1159	#1160	#1138	#1139	#1140	#1141
	#1145	#1147	#1150	#1173						
A - B	#1152									
B - A	#1157	#1158	#1159	#1140	#1145	#1147	#1150			
A & B	#1154	#1155	#1160	#1138	#1139	#1141	#1173			

A = ReferenceC++/Maisa0.5
B = ReferenceC++/Human

A	#1152	#1154	#1155	#1160	#1138	#1139	#1141	#1173		
B	#1152	#1153	#1154	#1155	#1156	#1157	#1158	#1159	#1160	#1173
	#1138	#1139	#1140	#1141	#1142	#1143	#1144	#1145	#1146	#1147
	#1148	#1149	#1150	#1151						
A - B										
B - A	#1153	#1156	#1157	#1158	#1159	#1140	#1142	#1143	#1144	#1145
	#1146	#1147	#1148	#1149	#1150	#1151				
A & B	#1152	#1154	#1155	#1160	#1138	#1139	#1141	#1173		

A = ReferenceC++/Columbus3.5
B = ReferenceC++/Human

A	#1154	#1155	#1157	#1158	#1159	#1160	#1138	#1139	#1140	#1141
	#1145	#1147	#1150	#1173						
B	#1152	#1153	#1154	#1155	#1156	#1157	#1158	#1159	#1160	#1173
	#1138	#1139	#1140	#1141	#1142	#1143	#1144	#1145	#1146	#1147
	#1148	#1149	#1150	#1151						
A - B										
B - A	#1152	#1153	#1156	#1142	#1143	#1144	#1146	#1148	#1149	#1151
A & B	#1154	#1155	#1157	#1158	#1159	#1160	#1138	#1139	#1140	#1141
	#1145	#1147	#1150	#1173						

Figure 5.11: Comparison view of DEEBEE

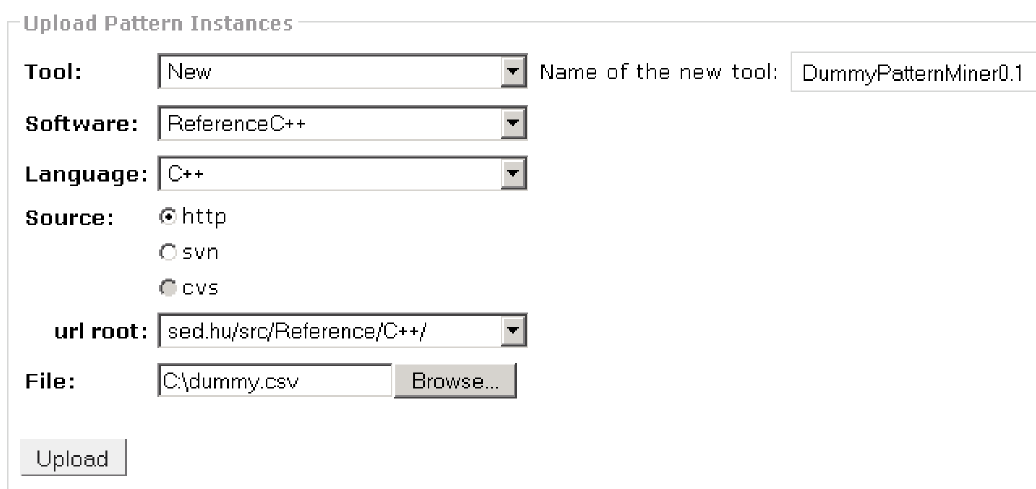

Figure 5.12: Adding the results of a new tool to DEEBEE

Example: In the example we will add a new tool called Dummy Pattern Miner v0.1 to the benchmark. The mined software will be the reference implementation of the GOF patterns in C++. Figure 5.12 lists the completed forms.

5.3 Experiments performed

Here Columbus and Maisa were evaluated and compared using the benchmark. The tools were evaluated on C++ reference implementations of design patterns, on NotePad++, and on a program called FormulaManager implemented by us to have a test case where the use of design patterns is well defined and documented. The reference implementations test the structural matching ability of the tools, while in the case of NotePad++ and FormulaManager the tools were examined in a real-life context where other factors are also considered, like the goal of the pattern. Columbus provides the common framework for the experiment (see Figure 5.13).

Reference implementations. To compare different tools, reference implementations of design patterns based on book by Gamma et. al. [39] were created by us. With these reference implementations the basic capabilities of C++ pattern miner tools can be evaluated and compared.

Design pattern mining strongly depends on the analysis of the source code. Hence the source files in the reference implementations do not contain any difficult programming structures like templates and they do not include any standard headers to avoid parsing problems.

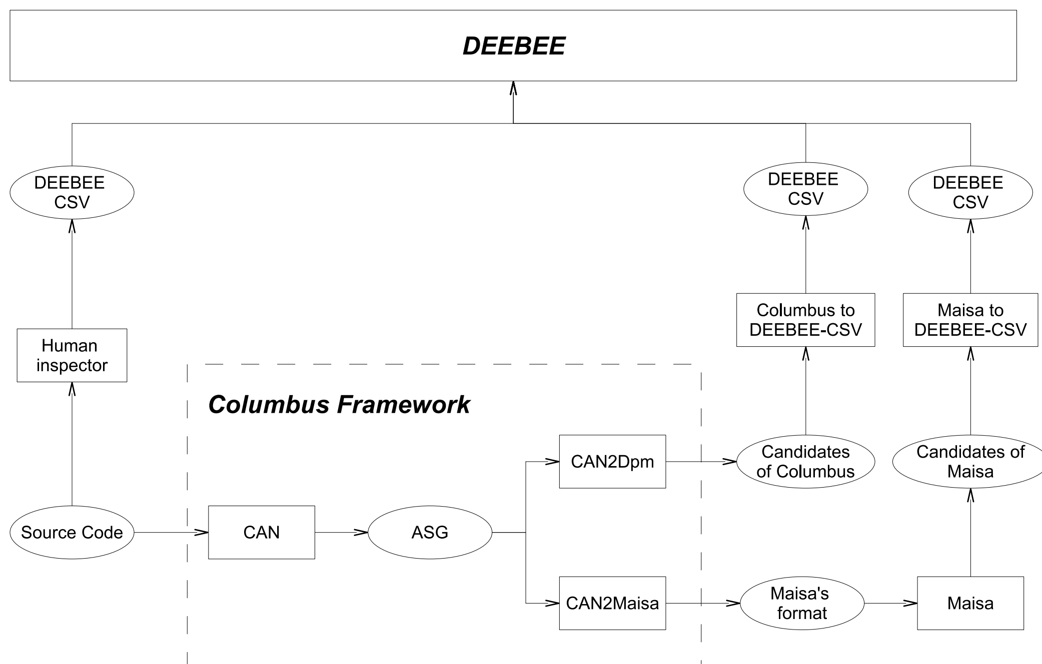

Figure 5.13: Framework for design pattern mining

Let us take a concrete reference implementation, the *Adapter Object*[3]. The reference implementation of Adapter Object is shown in Figure 5.14. In reference implementations each participant class starts with the "Example_" prefix; similarly, attributes start with the "example_" prefix.

NotePad++ is a free source code editor which supports several programming languages, running in Microsoft Windows environments. It is based on the Scintilla editor component (a very powerful editor component), written in C++ with pure win32 API and STL. Calculated by the metric component of Columbus, NotePad++ contains 95 classes, 1,371 methods, 776 attributes and 27,033 useful lines of code.

When evaluating design pattern miner tools on real software systems, it is necessary to consider not only candidates mined by the tools, but instances recognized by us as well. If instances mined by programmers are skipped in an evaluation then the recall of the tools cannot be calculated. Therefore instances from NotePad++ recovered manually were also added to the benchmark and were examined during the evaluation of Columbus and Maisa.

FormulaManager. Since the reference implementations contain disjoint implementations of the design patterns in an unreal context, we developed a program called

[3]We introduced this pattern earlier in Section 3.2.1

```
class Example_Target {
    public:
        virtual void function() = 0;
};

class Example_Adaptee {
    public:
        virtual void other_function();
};

class Example_AdapterObject : public Example_Target {
    public:
        virtual void function();
    private:
        Example_Adaptee* m_adaptee;
};

void Example_Adaptee::other_function() {
}

void Example_AdapterObject::function() {
    m_adaptee->other_function();
}
```

Figure 5.14: Reference implementation of Adapter Object

FormulaManager where each design pattern occurs in a real context at least once. The goal of FormulaManager is to manage formulas, namely execute operations on formulas (evaluation, prefix form, postfix form), export the results to different kinds of representations (HTML, XML, CSV), display the results in different views (list, table), handle different numbering systems and validate a formula.

Tables 5.2, 5.3 and 5.4 present the results obtained from using the tools, where the upper part shows the number of candidates found. In the lower part of the tables a summary is shown as well, where precision, recall, the correctness mean and the completeness mean are provided by the benchmark's statistics view (see Section 5.2.1). A dash in the tables shows that the tool did not search for the given pattern.

Tool	Columbus	Maisa	Human
Abstract Factory	2	2	1
Adapter Class	1	1	1
Adapter Object	1	1	1
Bridge	1	-	1
Builder	1	1	1
Chain Of Resp.	0	-	1
Command	-	-	1
Composite	-	-	1
Decorator	1	-	1
Facade	-	-	1
Factory Method	1	0	1
Flyweight	-	0	1
Interpreter	-	-	1
Iterator	1	0	1
Mediator	-	0	1
Memento	-	1	1
Observer	-	0	1
Prototype	1	1	1
Proxy	1	1	1
Singleton	0	0	1
State	1	-	1
Strategy	1	-	1
Template Method	1	-	1
Visitor	1	4	1
Precision	100%	80.00%	100%
Recall	58.33%	33.33%	100%
Correctness mean	100%	80.00%	100%
Completeness mean	96.43%	85.00%	100%

Table 5.2: Number of design pattern candidates found for Reference Implementations

5.3.1 Reference implementations

Now we will give a summary of our experiments on evaluating the tools on reference implementations. The number of pattern candidates found by the tools in reference implementations are listed in Table 5.2. Pattern instances found by programmers (the column labelled *Human*) in the reference implementations are true candidates.

Some design patterns are more difficult to discover than others. This is due to their unclear structure even if the aim of such a design pattern is obvious. The purpose of the unclear specification is to allow flexible implementations of the design pattern. For example, in the case of Mediator, only an abstract incomplete structural specification is given; e.g. no methods are specified in the participant classes. Most of the tools consider just the structural information of the patterns so they encounter problems in this case. A promising solution is presented by Wendehals [89] for improving design pattern discovery by exploiting the dynamic information available.

Because the two evaluated tools use only structural information, they do not discover most of the patterns with unclear specification, e.g. Command, Facade and Interpreter, as can be seen in Table 5.2. Although Maisa seeks to discover some patterns that have an unclear specification like Mediator, Memento, Observer and Flyweight, it works only in the case of Memento (candidate #1152). Because Maisa failed to find these patterns, its recall score is only 33.33%.

If several pattern candidates were discovered in the reference implementations for a given pattern, then either *false positives were found* or *the pattern instance was recognized several times*. For instance, Maisa discovered four Visitor pattern candidates. We examined these candidates in the benchmark and realized that two of them were false positives, while the other two instances were actually the same instance with a common abstract, but with different concrete participants. The two true instances were related as siblings (see Section 5.1) to each other and to the single pattern instance found by Columbus and Human. This way, these instances appear as one common instance in the benchmark.

Instance view of #1169

Back to previous view

Pattern Instance Information

Tool	Maisa0.5
Software	ReferenceC++
Pattern	Visitor
Participants	
class ConcreteElement	Example_ConcreteElement2
operation accept	example_accept
operation operation	example_operation2
class ConcreteVisitor	Example_ConcreteVisitor1
operation visitElement	VisitConcreteElement2
*class Element	Example_ConcreteElement2
operation accept	example_accept
*class Visitor	Example_Visitor
operation visitElement	VisitConcreteElement2

How complete is it? stat

- (selected) The pattern instance is complete in every aspect.(100%)
- Some participants are missing from the pattern instance.(50%)
- Important participants are missing from the pattern instance.(0%)
- *Apply this answer to siblings too.*

submit

How correct is it? stat

- I am sure that it is a real pattern instance.(100%)
- I think that it is a real pattern instance.(66%)
- I think that it is not a real pattern instance.(33%)
- (selected) I am sure that it is not a real pattern instance.(0%)
- *Apply this answer to siblings too.*

submit

example_Visitor.h(40)

```
class Example_Visitor;
class Example_ConcreteVisitor1;
class Example_ConcreteVisitor2;

class Example_ObjectStructure;
class Example_Element;
class Example_ConcreteElement1;
class Example_ConcreteElement2;

class Example_Visitor{
public:
        virtual void VisitConcreteElement1(Example_ConcreteElement1*) = 0;
        virtual void VisitConcreteElement2(Example_ConcreteElement2*) = 0;

};
class Example_ConcreteVisitor1 : public Example_Visitor{
        virtual void VisitConcreteElement1(Example_ConcreteElement1*);
        virtual void VisitConcreteElement2(Example_ConcreteElement2*);
};
class Example_ConcreteVisitor2 : public Example_Visitor{
        virtual void VisitConcreteElement1(Example_ConcreteElement1*);
        virtual void VisitConcreteElement2(Example_ConcreteElement2*);
};

class Example_ObjectStructure{};

class Example_Element{
protected:
        Example_Visitor *_visitor;
public:
        virtual void example_accept(Example_Visitor*) = 0;
};

class Example_ConcreteElement1: public Example_Element{
public:
        virtual void example_accept(Example_Visitor*);
        virtual void example_operation1();
};

class Example_ConcreteElement2: public Example_Element{
public:
        virtual void example_accept(Example_Visitor*);
        virtual void example_operation2();
};
```

Figure 5.15: Visitor candidate mined by Maisa from reference implementations

Figure 5.15 shows the instance view of one of the two false Visitor candidates mined by

Maisa (candidate #1169). The source of Example_ConcreteElement2 class is loaded into the right hand side of the view because it was selected previously from the participants. It is a false candidate because Example_ConcreteElement2 acts as a class Element participant, as can be seen in the participants' area (see Figure 5.15). Maisa discovered another false candidate because it made the same mistake (#1171).

In the case of Abstract Factory, both Maisa and Columbus discovered two pattern candidates. These pattern candidates are true, the only problem being that neither tool can group concrete elements of the Abstract Factory pattern, so they discovered the same pattern instance twice with different concrete participants. It caused the mean of Columbus' completeness to be 96.43%. Maisa had a similar problem in the case of the Visitor pattern so its mean of completeness is 85%.

Comparison. The candidates found by Columbus, Maisa and humans were compared by the comparison view module of the benchmark (see Figure 5.16). Let us examine the first table in Figure 5.16. This table shows a comparison of all candidates found by Maisa and Columbus (no threshold was set as it was in case of Figure 5.11). The difference between Maisa and Columbus (row A-B) contains three pattern candidates. The previously mentioned false positive Visitor candidates (#1169 and #1171) were only found by Maisa, so these appear in the difference. The other difference is a true positive pattern candidate of Memento (#1152), which was found by Maisa but not by Columbus. Because Columbus found several design patterns that Maisa did not, a lot of instances appear in the difference between Columbus and Maisa (row B-A). But the tools found some patterns in common, with the same results that appear in the intersection (row A&B).

5.3.2 NotePad++

After examining the tools on the reference implementations, we performed another test, but this time on the real project called NotePad++. The results obtained from using the tools can be seen in Table 5.3, which will be elaborated on below.

Columbus only found a couple of types of patterns (Adapter Object, State and Template Method), but with better precision than those (Adapter Class, Adapter Object, Builder, Prototype and Proxy) found by Maisa. It should also be remarked that software developers tend to find those pattern instances which are difficult to find by pattern miner tools because of their unclear specifications. Therefore, we think that instances found by humans are a good supplement to the use of automatic tools even if software developers do not find all the pattern instances in a particular project.

In Table 5.3 we see that the precision of Human on NotePad++ is not 100%. This is due to the fact that the results of Human in the benchmark consist of several people

Do you want to group siblings?

Yes

Threshold for true instances.

0 % Set

☑ *No threshold (every pattern instance)*

A = ReferenceC++/Maisa0.5
B = ReferenceC++/Columbus3.5

A	#1152	#1154	#1155	#1171	#1160	#1169	#1138	#1139	#1141	#1173
B	#1154	#1155	#1157	#1158	#1159	#1160	#1138	#1139	#1140	#1141
	#1145	#1147	#1150	#1173						
A - B	#1152	#1169	#1171							
B - A	#1157	#1158	#1159	#1140	#1145	#1147	#1150			
A & B	#1154	#1155	#1160	#1138	#1139	#1141	#1173			

A = ReferenceC++/Maisa0.5
B = ReferenceC++/Human

A	#1152	#1154	#1155	#1171	#1160	#1169	#1138	#1139	#1141	#1173
B	#1152	#1153	#1154	#1155	#1156	#1157	#1158	#1159	#1160	#1173
	#1138	#1139	#1140	#1141	#1142	#1143	#1144	#1145	#1146	#1147
	#1148	#1149	#1150	#1151						
A - B	#1169	#1171								
B - A	#1153	#1156	#1157	#1158	#1159	#1140	#1142	#1143	#1144	#1145
	#1146	#1147	#1148	#1149	#1150	#1151				
A & B	#1152	#1154	#1155	#1160	#1138	#1139	#1141	#1173		

A = ReferenceC++/Columbus3.5
B = ReferenceC++/Human

A	#1154	#1155	#1157	#1158	#1159	#1160	#1138	#1139	#1140	#1141
	#1145	#1147	#1150	#1173						
B	#1152	#1153	#1154	#1155	#1156	#1157	#1158	#1159	#1160	#1173
	#1138	#1139	#1140	#1141	#1142	#1143	#1144	#1145	#1146	#1147
	#1148	#1149	#1150	#1151						
A - B										
B - A	#1152	#1153	#1156	#1142	#1143	#1144	#1146	#1148	#1149	#1151
A & B	#1154	#1155	#1157	#1158	#1159	#1160	#1138	#1139	#1140	#1141
	#1145	#1147	#1150	#1173						

Figure 5.16: Comparison of candidates in reference implementations

Tool	Columbus	Maisa	Human
Abstract Factory	0	0	0
Adapter Class	0	4	0
Adapter Object	9	4	0
Bridge	0	-	0
Builder	0	1	0
Chain Of Resp.	0	-	0
Command	-	-	0
Composite	-	-	1
Decorator	0	-	0
Facade	-	-	0
Factory Method	0	0	0
Flyweight	-	0	0
Interpreter	-	-	1
Iterator	0	0	1
Mediator	-	0	0
Memento	-	0	0
Observer	-	0	1
Prototype	0	6	1
Proxy	0	11	1
Singleton	0	0	2
State	1	-	0
Strategy	0	-	0
Template Method	3	-	1
Visitor	0	0	0
Precision	62.50%	16.67%	90.91%
Recall	29.41%	11.76%	58.82%
Correctness mean	69.42%	20.63%	77.05%
Completeness mean	100%	92.36%	90.91%

Table 5.3: Number of design pattern candidates found for NotePad++

who may disagree on whether a pattern candidate fulfils the original intent of the design pattern. It is one of the great benefits of the benchmark that it lets people vote on whether the uploaded pattern candidates are pattern instances or not. Even though human analysis resulted in the highest recall (58.82%), it is still far from 100%, which indicates that the current amount of human analysis effort put into the benchmark is still not enough to find all the pattern instances in larger projects.

5.3.3 FormulaManager

Next, we will summarize our experiments on evaluating the tools on FormulaManager. We note that FormulaManager differs from the simple reference implementations because the design patterns are used in a real-life context and they are interrelated. The number of pattern candidates found by the tools in FormulaManager is listed in the second, third

Tool	Columbus	Maisa	Human
Abstract Factory	4	0	2
Adapter Class	2	1	1
Adapter Object	4	0	1
Bridge	6	-	2
Builder	1	0	1
Chain Of Resp.	1	-	1
Command	-	-	1
Composite	-	-	1
Decorator	1	-	1
Facade	-	-	1
Factory Method	2	3	4
Flyweight	-	1	1
Interpreter	-	-	1
Iterator	1	0	1
Mediator	-	0	1
Memento	-	0	1
Observer	-	0	1
Prototype	3	2	2
Proxy	2	0	1
Singleton	3	0	3
State	9	-	1
Strategy	2	-	2
Template Method	1	-	1
Visitor	2	3	1
Precision	52.27%	80%	100%
Recall	71.88%	25%	100%
Correctness mean	55.23%	85.95%	100%
Completeness mean	85.98%	48.02%	100%

Table 5.4: Number of design pattern candidates found for FormulaManager

and fourth columns of Table 5.4. Let us examine and compare the scores of Columbus and Maisa on FormulaManager. It is clear that Maisa has a good precision score (80%), but its recall score is poor (25%). The poor recall is due to the fact that Maisa failed to find the instances of several design patterns (e.g. Builder, Mediator, Proxy) and it did not try to discover every kind of pattern (e.g. Command, Decorator). In contrast, Columbus has a fair recall score (71.88%) and precision score (52.27%). Columbus found several false positives in the case of State, Bridge and Adapter Object, which resulted in its lower precision score compared to Maisa. In the case of the other patterns, Columbus has good precision scores.

5.4 Evaluation of the benchmark

Sim et al. [74] defined seven requirements for a benchmark. In this section we will cite the requirements, and afterwards give an evaluation of the benchmark based on them.

Accessibility. *"The benchmark needs to be easy to obtain and easy to use. The test materials and results need to be publicly available..."* The benchmark is easy to obtain, it is online and accessible from the home page of the Institute of Informatics at the University of Szeged [26].

Affordability. *"The cost of using the benchmark must be commensurate with the benefits..."* The only investment required is to extend the tool to be measured in the benchmark to produce a CSV file in a suitable format (see Section 5.2.3) for uploading its results. If it is measured on a subject system that has already been inspected by professional software developers, then the results (e.g. precision and recall) will be immediately available without any extra effort required.

Clarity. *"The benchmark specification should be clear, self-contained, and as short as possible..."* In the case of design pattern mining, it is clear that the tool has to find instances of patterns in the source code. The most important inconsistency that we experienced is that many tools tend to find the same pattern candidate several times if there is e.g. a participant in the pattern description which is typically implemented in several ways (e.g. the ConcreteStrategy participant in the Strategy design pattern). Our benchmark includes the siblings mechanism to automatically deal with this issue.

Relevance. *"The task set out in the benchmark must be representative of ones that the system is reasonably expected to handle in a natural (meaning not artificial) setting and the performance measure used must be pertinent to the comparisons being made..."* The benchmark contains reference implementations of the GOF design patterns [39] that test the basic capabilities of the tools. To test the tools on real-life cases, we also uploaded pattern candidates found in real software by three well-known tools (Maisa, Columbus and Design Pattern detection Tool). There are also two software systems in the benchmark that have already been inspected by experienced software developers concerning design pattern usage (NotePad++ and FormulaManager).

Solvability. *"It should be possible to complete the task domain sample and to produce a good solution..."* The benchmark contains subject systems of varying size and complexity, which makes it easier or harder to mine pattern instances from them. It contains reference implementations of design patterns expected to be found by all competing tools and real-life software systems to test their actual capabilities.

Portability. *"The benchmark should be specified at a high enough level of abstraction to ensure that it is portable to different tools or techniques and that it does not bias one*

technology in favour of others..." The benchmark is general; it is language, software, tool and pattern independent. It already contains pattern instances found in subject systems running on different platforms and written in different programming languages.

Scalability. *"The benchmark tasks should scale to work with tools or techniques at different levels of maturity..."* The benchmark contains reference implementations of design patterns that can be utilized to test research prototypes and it also contains real-life software systems to test mature tools like commercial products.

5.5 Summary

In this chapter we presented a newly developed benchmark and performed experiments on it for evaluating and comparing the precision and recall of design pattern miner tools. The benchmark is general from the viewpoint of the mined software systems, programming languages, uploaded design pattern candidates, and design pattern miner tools.

With the help of our benchmark the accuracy of two design pattern miner tools (Columbus and Maisa) were evaluated on reference implementations of design patterns and on two software systems, NotePad++ and FormulaManager. In this part design pattern instances used in NotePad++ were also discovered by hand, so both precision and recall scores can be calculated via the benchmark. Furthermore, we developed a software system called FormulaManager to test the tools on a program where each design pattern is implemented in a real-life context.

Sometimes false candidates were found by the tools because they only examined the structure of the code, while programmers also take the code's behaviour into account. This is why instances mined by humans can be used to provide additional results on top of those obtained by the tools.

With the help of our benchmark it was demonstrated that it is indeed possible to evaluate design pattern miner tools readily and effectively, which will hopefully lead to better quality tools in the future. This benchmark is *freely accessible* from the home page of the Institute of Informatics at the University of Szeged:

http://www.inf.u-szeged.hu/designpatterns/

"Most of the fundamental ideas of science are essentially simple, and may, as a rule, be expressed in a language comprehensible to everyone."
Albert Einstein

Chapter 6

Common format for design pattern miner tools

The previous chapters and other studies (e.g. fusing the results of tools [54]) highlighted several limitations of the current outputs of the design pattern detector (DPD) tools in form and content: some output formats (1) do not report either their own identity or the name and version of the program that they analyzed; (2) do not report all roles relevant to a given motif; (3) do not identify reported roles unambiguously; (4) do not identify detected motif candidates unambiguously; (5) do not report their conceptual schema of the identified motif; (6) do not justify their results; and (7) use ad hoc (generally textual) output formats. Point 1 makes it difficult to reproduce the DPD tool results; point 2 makes it hard to combine results from different tools; points 3 and 4 make results ambiguous; point 5 makes a comparison of results difficult; point 6 leads to problems when understanding and verifying the results; and, point 7 hinders the automated use of the results by other tools.

In the previous chapter, we introduced DEEBEE and described what it does. The CSV uploading format of DEEBEE was designed to represent information that is necessary only for evaluating and comparing the results of different tools. In this way there is no guarantee that the CSV format of DEEBEE will satisfy any kind of tool that may work on the result of a design pattern miner like a visualization tool or a fusion tool.

We propose to address these limitations by introducing a common exchange format for DPD tools, called DPDX, based on a well-defined and extendible metamodel. This format should aid the comparison [95], fusion [53], visualization [29], and validation [97] of the outputs of different DPD tools.

Consequently, the contributions of this chapter are twofold. First, we provide a good foundation for DPD. Second, we propose a common exchange format for DPD tools that fosters their synergetic use and supports the automated processing of their results.

6.1 Background

6.1.1 Motivation

A common exchange format for DPD tools would be beneficial to achieve a synergy of many different tools. Our vision is illustrated in Figure 6.1, where a federation of tools based on the common exchange format interact to produce new values. This federation and the common format is also an invitation to program comprehension and maintenance, and for re-engineering research communities to contribute individual tools, including tools unforeseen in the vision given in Figure 6.1.

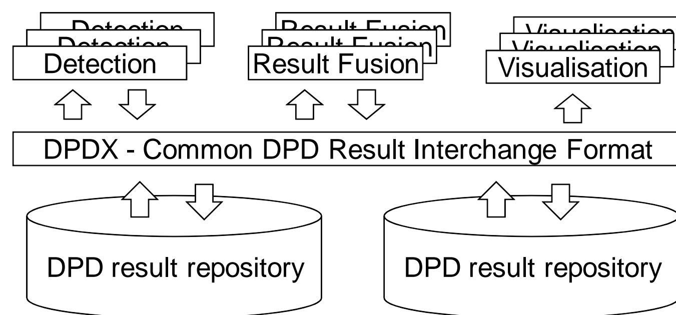

Figure 6.1: A federation of design pattern detection, visualization and assessment tools cooperating via the common exchange format.

For example, visualizations of DPD outputs could be built entirely using the common exchange format, instead of being implemented separately for each DPD tool. Next, it should be possible to automate the process of collecting, comparing, and evaluating the outputs of different tools, which is currently a manual, error-prone, and time-consuming task. Then public repositories of instances of design motifs (P-MARt [61], DEEBEE [97]) should benefit greatly from a common exchange format. These repositories are important in DPD research as a reference for assessing the accuracy of tools [63]. Moreover, a common exchange format would also help in achieving an automated round-trip in DPD tools (see Albin-Amiot et al. [2]), including pattern detection, collection, fusion, visualization, validation, storage, and generation.

6.1.2 Requirements

The common exchange format must fulfil the following core requirements 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 of results and scores expressing the confidence of a tool in its diagnostics to help experts and tool users digest and apply the reported results.
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.

In addition to the previous core requirements, the following two optional requirements are also desirable:

1. **Language-independence.** 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).
2. **Standard-compliance.** The specification should be consistent with existing standards so that it can be easily adapted, maintained, and evolved.

6.1.3 State of the art

We evaluated the output formats of several existing DPD tools (SPQR [79], DP-Miner [28], Fujaba [89, 91], Maisa [34, 57], SSA [84], Columbus [8], PINOT [70], Ptidej [41]) and of two DPD result repositories (DEEBEE [96],[97] and p-MARt [61]) from the viewpoint of the requirements listed above. The conclusions presented below are based not just on a thorough literature review, but also on intensive practical evaluations [53], [96], [100] of all tools except SPQR, which is not publicly available.

Each of the reviewed output formats for describing design pattern candidates contains some elements that are worth using in a general exchange format. In particular, each fulfils the *Language Independence* requirement, i.e. they contain no language specific information. Despite this, there is no format available that would fulfil all of our requirements.

Tool	DP-Miner	Maisa	SSA	SPQR	Columbus	Pinot	Ptidej	Fujaba	DEEBEE	P-Mart
Language-independence	√	√	√	√	√	√	√	√	√	-
Completeness	-	√	-	√	√	√	-	√	√	-
Standard compliance	CSV	-	XML	XML	-	-	-	XML	CSV	XML
Identification of role players	-	-	Nested classes	-	Outer classes	Outer classes	Outer classes	Classes Signatures	Classes methods fields	Classes
Identification of candidates	-	-	√	√	√	-	√	-	Unique IDs	-
Justification	-	-	-	-	-	-	Scores explanations	Scores	-	-
Comparability	-	-	-	-	-	-	-	-	-	-
Reproducibility	-	-	-	-	-	-	-	-	Tool and repository info	Repository info
Specification	-	-	-	-	-	-	-	-	-	Role types

Table 6.1: Tools and requirements satisfied by their output formats

Table 6.1 shows that four formats (of DP-Miner, SSA, PTidej and P-Mart) do not fulfil *Completeness* by not reporting all the relevant roles. Half of the tools analyzed by us use ad hoc formats, thus they fail to fulfill *Standard Compliance*. The formats of DP-Miner and DEEBEE exchange are based on a quasi standard called CSV, but unfortunately one that fails unambiguity since even classes are not identified uniquely (only by their name). However, it is worth noting that the other standard compliant output formats structure information using XML syntax, which supports nesting and therefore recommends itself as a good basis for a general exchange format.

All formats (except the output format of SSA) either do not support *Identification of Role Players* at all or just for a limited set of program elements, mostly outer classes. Fujaba is the only one that supports classes and method / field signatures. No format supports the unambiguous identification of elements at finer grained levels (individual statements). We noted that line numbers are not a satisfactory identification scheme.

Obviously, all the reviewed output formats are mainly intended to satisfy a human expert. They assume much implicit knowledge about programming languages and design patterns that a software engineer typically has, but which an automated tool does not. Typically, none of the tools provide explicit schemata of the searched design motifs, which would help other tools to understand their conceptual model of a pattern, and information about analysis methods employed. Therefore the *Comparability* requirement is not fulfilled by any tool.

DP-Miner does not fulfil *Identification of Candidates* since it can repeat the same candidate with the same role assignments. *Identification of Candidates* is not fulfilled by Maisa, PINOT and Fujaba because when a candidate has several method/fields playing the same role, these tools report multiple candidates (one for each method/field).

SSA does not report multiple candidates in such cases. By reporting only mandatory class roles, a candidate is identified unambiguously, hence *Identification of Candidates* is fulfilled.

Ptidej, DEEBEE, Columbus and SPQR fulfil *Identification of Candidates* since they merge different candidates that have the same mandatory role assignments. *Reproducibility* is fulfilled partly by P-Mart (only repository names and versions are included) and DEEBEE. We should add that P-Mart reports role kinds for class roles (Class, Abstract Class, etc.). Therefore we could claim that *Specification* is partly fulfilled (only by P-Mart). Last, but not least, only two tools (Ptidej and Fujaba) report confidence scores and only one tool (Ptidej) provides explanations. Ptidej, which provides explanations about violated and fulfilled constraints, implicitly hints at constraint satisfaction as the analysis technique employed.

The output formats of the above tools are shown in the Appendix (see Section B.1), while a detailed evaluation and comparison of the formats were published in our technical report [100].

6.2 DPDX concepts

Now we shall develop the concepts on which our proposed exchange format, DPDX, is based. We will show how DPDX addresses each of the requirements stated in Section 6.1.2, overcoming the limitations of existing output formats identified above.

6.2.1 Specification

The common exchange format will be specified by a set of extendible metamodels that capture the structural properties of the relevant concepts, such as candidates, roles and their relations. Metamodels that reflect the decisions explained in this section are presented in Section 6.3. They significantly extend previous similar proposals, like the PADL metamodel of Albin-Amiot et al. [3]. The possible kinds of program constituents and the related abstract syntax tree are not first-class elements of the metamodel, but are captured by a set of predefined values for certain attributes in the metamodel. This ensures easy extendibility since only the set of values must be extended to capture new relations or language constructs, while the metamodel and the related exchange format remain stable. The set of defined terms can be viewed as a simple ontology. Ontologies have already been used in the domain of design pattern detection. For example, Kampffmeyer et al. [51] showed that an ontology can be used to model the intents of design patterns. Their proposed ontology is useful for automatically relating design patterns to one another.

6.2.2 Reproducibility

A DPD result file must contain the diagnostics of a particular DPD tool for a particular program. To allow reproducibility of the results, it must include the name and version of the *tool employed* and the name, version, and the URI of the *program being analyzed*. Names and versions may be arbitrary strings. The URI(s) must reference the root directory(ies) of the program being analyzed. The URI field is optional, since the program being analyzed might not be publicly accessible. The other fields are mandatory.

6.2.3 Justification

The justification of diagnostics consists of confidence scores, reported as real values between 0 and 1, and textual explanations. Justification information can be added at each level of granularity: i.e. for a complete candidate, individual role assignments and individual relation assignments.

6.2.4 Completeness

To identify a candidate unambiguously, each program constituent that may play a mandatory role must be reported (*Identification of Candidates* requirement). Therefore, DPDX allows the reporting of each of the following constituents: *nested and top-level types* (interfaces, concrete and abstract classes); *fields* and *methods*; *any statements* (including field accesses and method invocations). Reporting role mappings at all possible granularity levels improves the presentation of the results and aids their verification by experts and use by other tools. Reporting roles used by statements different from invocations and field accesses is important because they are essential for disambiguating certain motifs.

6.2.5 Identification of role players

The main part of a DPD result file consists of role mappings, i.e. assignments of program constituents to the roles that they play in a motif. Given a particular program version and program constituent description, it should be possible to identify the constituent precisely and unambiguously in the program. In addition, it would be beneficial if the identification scheme were *stable*, i.e. if it were not affected by changes in the source code that are mere formatting issues or the reordering of elements whose order has no semantic meaning. For instance, after inserting a blank line or changing the order of declarations, each program element should still have the same identifier as before. This is necessary to compare DPD results across different program versions, when analyzing the evolution of design pattern implementations over time.

Identifying named elements According to *Completeness*, we must unambiguously identify program elements down to the granularity level of individual statements.

Stable identification is easy for type and field declarations, which are typically labelled. Chaining names from outer to inner scopes is sufficient for identifying declarations of classes and fields. For instance, in the example presented in Figure 6.2, `myApp.A` identifies class A and `myApp.B.b` identifies field b of class B.

```
package myApp;
class A {public void f(int a, int b){...}}
class B {
  int b;
  public void b(B b) {...}
  public void b(A a) {
    int c, d;

    if (...) a.f(c,d) else a.f(d,c);
  }
}
```

Figure 6.2: Named and unnamed elements example

Because in many object-oriented languages methods can be overloaded, unique identification requires including the types of method arguments in the identifier of a method.

Identifying unnamed elements Alas, nested naming is unsuitable for fine-grained elements (statements and expressions), which may occur several times in the same scope, e.g. in the same method body or field initializer expression. Cases like the two invocations of method f() within the body of method B.b(myApp.A) in Figure 6.2 can neither be disambiguated by additionally reporting the static type of invocation arguments (which is the same in both cases) nor by adding line number information (which anyway fails the stability requirement).

However, each element can be identified uniquely by a *path* in an abstract syntax tree (AST) representation of the respective program. This path consists of names for the child branches of each AST element and positions within statement sequences. We call this the *model-based identification scheme* since it assumes a standardized model of an abstract syntax tree and standardized names for its parts. For instance, the if statement in the example presented in Figure 6.2 can be identified by *ifPath* = myApp.B.b(myApp.A).body.2. This illustrates how child elements of an already identified element are identified either by their unique, standardized name within the enclosing element (e.g. body as the name of the block representing a method body) or by their

unique position inside the enclosing element (e.g. 2 as the position of the if-statement within the block). Accordingly, we can denote the invocation of f() in the first alternative by *ifPath*.then.1.call, distinguishing it from that in the second alternative, denoted *ifPath*.else.1.call.

Serving all needs. To satisfy the diverging needs of fusion tools, visualization tools and humans, precise hierarchical identification information is complemented with information about source code positions, where available. Source code positions contain a file path in Unix syntax (relative to the base directory indicated by the URI of the program being analyzed), a start position and an end position in the file, each indicated by a line and column number.

In addition, field accesses and method invocations may be complemented by information about the accessed field or called method. For instance, the invocation of f() in the *then* part could also be reported as "*ifpath*.then.1.call=myApp.A.f(int,int)". Since model-based identification is unambiguous the additional information "=myApp.A.f(int,int)" is just an optional courtesy to programmers and tools who use the DPD results. It lets them know which element is referenced by the field access or method invocation without needing to analyze the code of the source program. The class 'ReferencingStatement' in the metamodel (Section 6.3.2) reflects the option to provide additional referencing information like this.

DPD tools are required to support at least the hierarchical naming of types, fields, methods and argument types in method signatures. The source code position and the model-based identification of statements is optional, since tools based on byte code analysis will not always be able to provide it.

6.2.6 Language independence

The standardized model of an abstract syntax tree that underlies the above program element identification approach is reflected by the program element identifier metamodel described in Section 6.3.2 and a set of standardized element names (see Appendix, Section B.2) cover the abstract syntax of a wide range of strongly typed imperative and object-oriented languages with a name-based type system (e.g. Beta, C, C++, Eiffel and Java) and dynamically typed languages (e.g. Smalltalk). The metamodel abstracts from details that are not relevant for unique identification. Types are subsumed as named elements.

6.2.7 Identification of candidates

Several of the tools we reviewed (like PINOT and DP-Miner) report multiple candidates for the same instance of a motif, whose given roles are played by different program

constituents. For example, PINOT outputs a separate Decorator candidate for each forwarding method if multiple methods play the 'Operation' role. Reporting "related" candidates repeatedly

- can confuse developers and automated tools that might use the results and it would lead to
- erroneous precision and recall and
- false diagnostics that could be otherwise avoided.

Avoiding the multiple reporting of "related" candidates requires first of all a well-defined notion of identity for candidates. Most tools do not explicitly define such a notion. Some define the conceptual identity of a candidate to be the set of values that it assigns to mandatory roles (Columbus, Ptidej and DEEBEE). However, this definition is insufficient, since it implies that two Decorator candidates that only differ in the player of the (mandatory) "Operation" role will be treated as different. However, a decorator instance may have many methods that play the "Operation" role and all the players must be reported as being part of the same instance (or candidate).

In this context the contribution of this section is a precise definition of candidates and candidate identity and a clarification of its implications for DPD tools, the exchange format and DPD result fusion.

A *design pattern schema* is a set of named roles and named relationships between these roles. *A role* has a name, a set of associated properties, an indication of the kind of program element that may legally play that role (like a class or method), a set of contained roles and a specification of the role cardinality, which determines how many elements that play the role may occur within the enclosing entity. Mandatory roles have a cardinality greater than zero. *A relationship* has a name and cardinalities specifying how many program elements that play a particular role can be linked to either end of the relationship.

A *role mapping* maps roles and relations of the schema to elements of a program so that the target program elements are of the required kind, have the required properties and relationships and fulfil the cardinality constraints stated in the schema. The essential task of DPD tools is suggesting role mappings. The set of all role mappings identified by a DPD tool for a particular schema and program being analyzed defines a graph with nodes being the program elements playing roles and arcs being the relations between these elements. Each relation between elements reflects a relation between the roles that the elements play. We call this graph the *projection graph*, since it represents the projection of the schema on the program being analyzed. A *candidate* is the set of nodes in a connected component of the projection graph. Each proper subset of a candidate is called a candidate fragment or simply a *fragment*.

The identifiers of *any* program element that is part of a candidate uniquely identifies that candidate since the same element cannot occur in any other (complete) candidate. However, having possibly different identifiers for the same candidate is unsatisfactory, since it makes it hard to compare candidates based on their identifiers. Therefore, we require that every pattern schema specifies exactly one of its mandatory roles as the identifying role. The identifier of the element that plays that role in a particular candidate will identify that candidate.

These notions are illustrated in Figure 6.3. The left-hand side shows a graph that represents the core structure of the Decorator schema (only the roles and relationships are shown without their attributes; for a detailed representation, see Section 6.3.1). The right-hand-side shows a projection graph induced in some hypothetical program by a possible set of role and relation mappings. It has two connected components, corresponding to two candidates. If we assume that "Component" is the identifying role, then the two candidates are uniquely identified by the classes A and C. The different colours in the A candidate represent possible fragments. The multiple candidates erroneously reported by PINOT and DP-Miner for one instance correspond to such fragments.

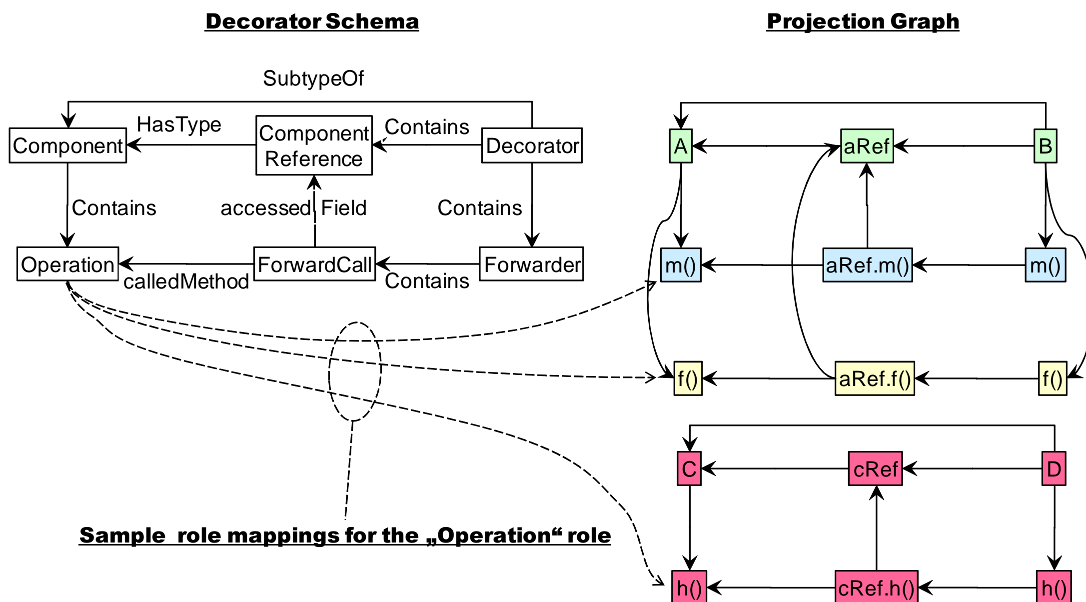

Figure 6.3: Illustration of candidates

The unambiguous candidate identification requirement is fulfilled if a tool reports (complete) candidates only, not fragments. This requires that the exchange format provide the means of expressing a mapping of a particular role to multiple players within the same candidate, like the methods m() and f() playing the "Operation" role in the upper candidate shown in Figure 6.3. Since XML is well suited to represent hierarchical nesting and also fulfils our standard compliance requirement, that DPDX be based on XML.

6.2.8 Comparability

DPDX supports comparability by specifying a precise metamodel of schemata, enabling tools to report their schemata. In addition, it provides the means to specify analysis methods employed and specifies a common vocabulary of analysis methods.

6.3 DPDX meta-models

This section presents the three meta-models that 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. These models reflect the decisions outlined in the previous section. Figure 6.4 shows how these models are related.

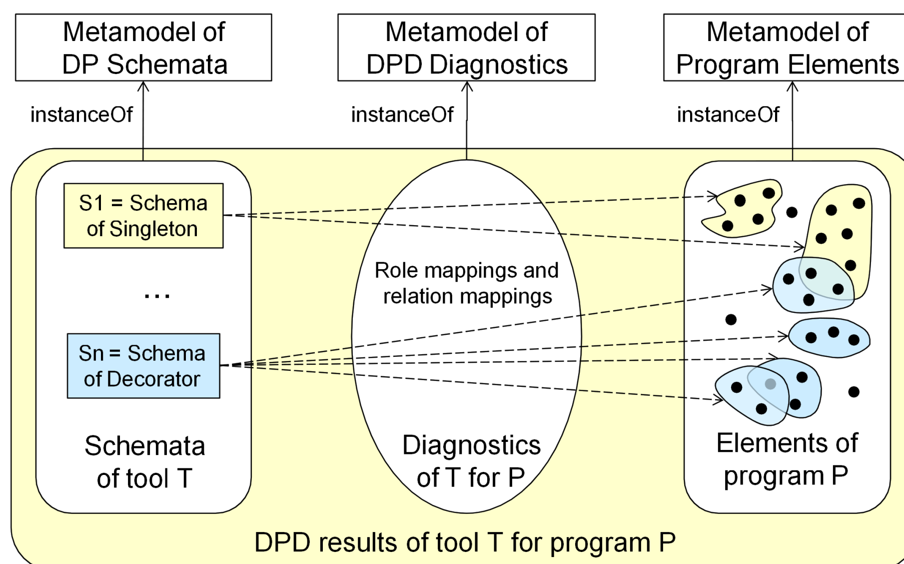

Figure 6.4: Relation between schemata, diagnostics and instances

Here the results are instances of the result metamodel. Their main part is the mapping of roles and relations to program elements. Candidates are targets of mappings like this (see Section 6.2.7). 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 in Figure 6.4.

6.3.1 Schema metamodel

The metamodel of design pattern schemata is illustrated in Figure 6.5. An instance of the metamodel that represents the schema of the 'Decorator' motif is shown in Figure

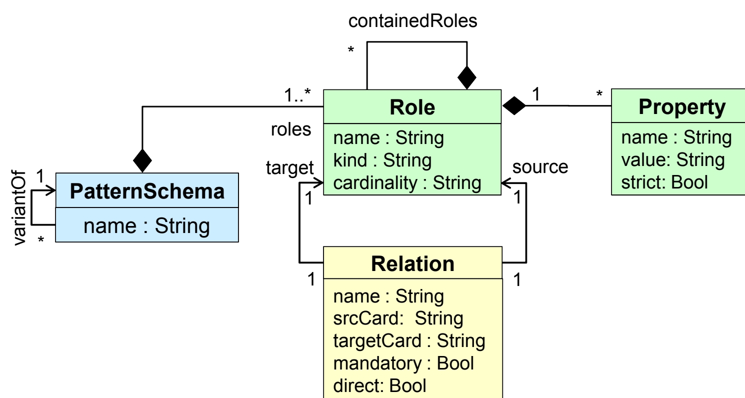

Figure 6.5: Metamodel of design pattern schemata

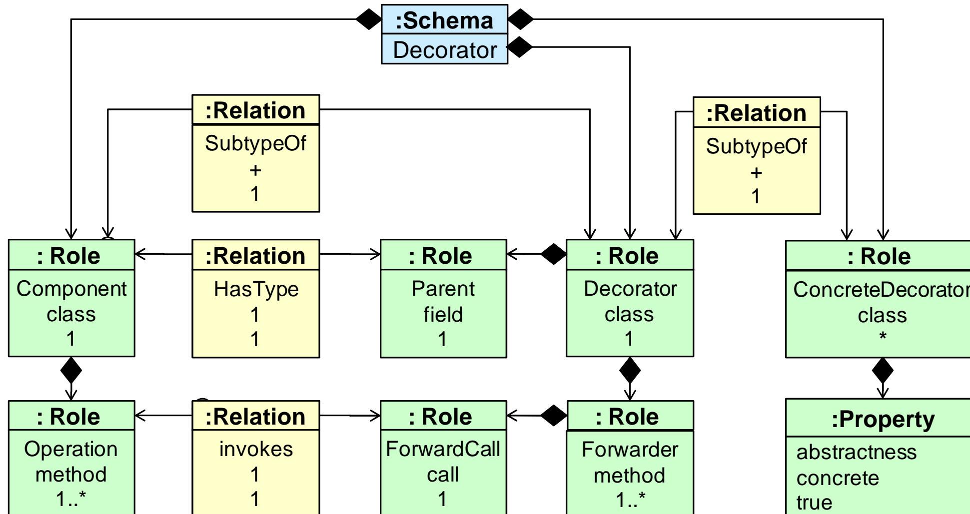

Figure 6.6: Sample of design pattern schemata

6.6. Here, the values of the attributes “mandatory” and “direct” in relation instances, and the aggregation of relation instances within the schema instance have been omitted for conciseness.

The metamodel reflects the definition of schemata in Section 6.2.7 and supplements it with the definition of properties as triples consisting of a name, a value and a boolean that indicates whether the property must be met exactly or might be relaxed. In the first case it represents a core characteristic (e.g. the 'ConcreteDecorator' role must be played by a class whose ‘abstractness’ property has the value *concrete* – see Figure 6.6). Otherwise, it is ignored if it not fulfilled, but it increases the confidence in the diagnostic if it fulfilled (e.g. the ‘Decorator’ is typically abstract, but not always). The metamodel also adds the option to formally state that a schema is a variant of another one, e.g. a ‘Push Observer’ is a variant of the ‘Observer’ motif.

Note that the representation can accommodate arbitrary languages and the evolution of existing languages without any change in the metamodel because language level concepts

(e.g. classes, methods, statements) are not first class entities of the metamodel, but just values of the 'kind' field of the Role class. In order to enable different tools to understand each other, it is sufficient to agree on a common vocabulary; that is, a set of 'kind' values with a fixed meaning. For instance, the 'kind' class generally represents an object type and the distinction between interfaces, abstract classes and concrete classes is represented by the property 'abstractness' with predefined values *interface*, *abstract* and *concrete*. A suggested common vocabulary is presented in the Appendix (see Section B.2).

6.3.2 Program element metamodel

The program element metamodel is illustrated in Figure 6.7. The identification scheme elaborated on in Section 6.2.5 distinguishes

- named elements (fields, classes, interfaces and primitive or built-in types),
- typed elements (method signatures),
- indexed elements (statements in a block) and
- blocks.

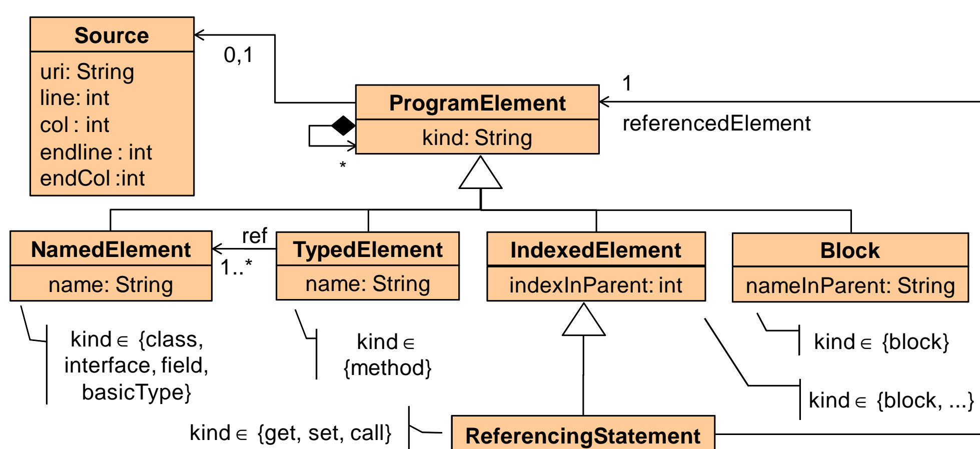

Figure 6.7: Metamodel of program element identifiers and optional source locations

Each of these elements can be nested inside other elements. This is general enough to accommodate even exotic languages. Although blocks and named elements look similar (both contain just a name), there is a significant distinction. The names of named elements stem from the program being analyzed, whereas those of blocks have a fixed vocabulary (see Section B.2). Each block is named after its role in the program element in which it occurs (e.g. ifCondition, then, else, whileCondition, whileBody). The 'kind' field corresponds to the one in the schema metamodel and it can have the same values. Indexed elements whose kind is get, set or call can be optionally treated as

referencing statements, allowing us to add information concerning the referenced element (see Section 6.2.5).

Figure 6.8 illustrates the object representation of the invocation a.f(d,c) from the code example given in Section 6.2.5. In the textual notation used there it is denoted by 'myApp.B.b(myApp.A).body.2.else.1.call'.

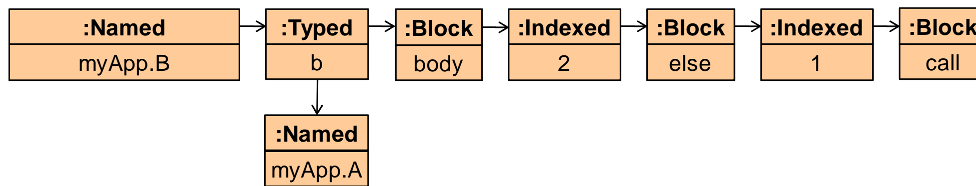

Figure 6.8: Representation of the invocation a.f(d,c) from the code example in Figure 6.2

6.3.3 Result metamodel

Figure 6.9 shows the metamodel of DPD results. A *DPD result* contains a set of diagnostics produced by a tool for a given program. Each *diagnostic* contains a set of role and relation assignments and a reference to the pattern schema whose roles and relations are mapped. Each *role assignment* references a mapped role and a mapped program element that plays the mapped role. A *relation assignment* references a mapped relation, a program element that serves as a relation source and a program element that serves as a relation target. Optional justifications can be added to diagnostics and each of their role and relation assignments. Schemata, roles, relations and program elements are defined according to Figure 6.5 and Figure 6.7.

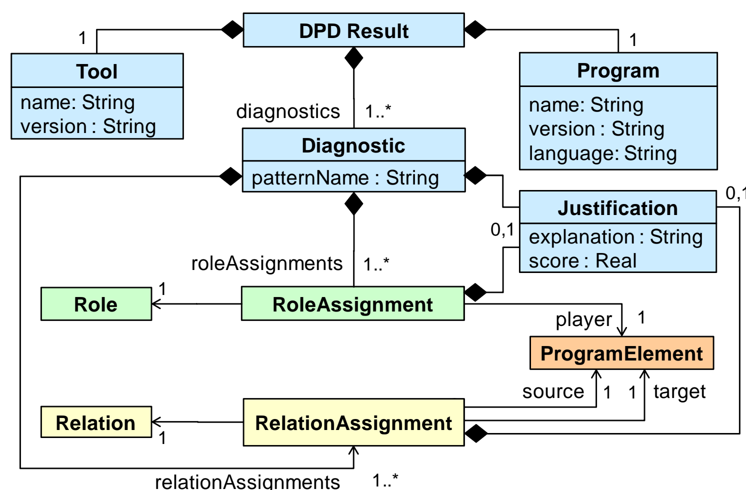

Figure 6.9: Metamodel of the design pattern detection results

The result metamodel is demonstrated in the common example of a Decorator instance taken from Java IO, illustrated in Figure 6.10. Mandatory roles are shown with filled boxes, while optional roles are shown with empty boxes.

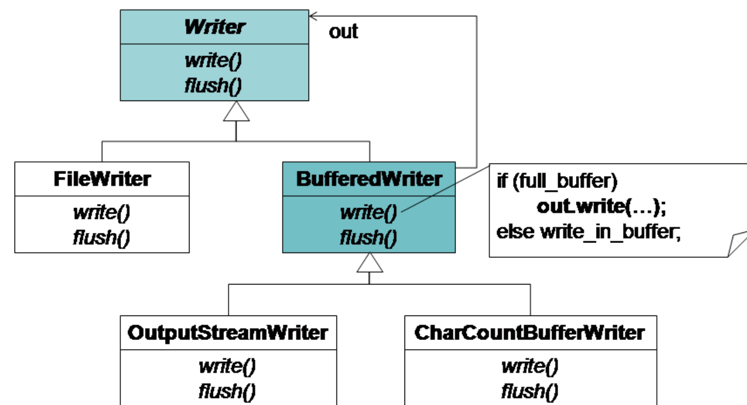

Figure 6.10: A decorator instance from the java.io package of the JDK with subclasses implemented by us (OutputStreamWriter and CharCountBufferWriter)

Figure 6.11 shows a few role and relation assignments for an instance of the Decorator pattern, consisting of the program elements java.io.Writer, java.io.BufferedWriter, java.io.Writer.write() and java.io.BufferedWriter.write().

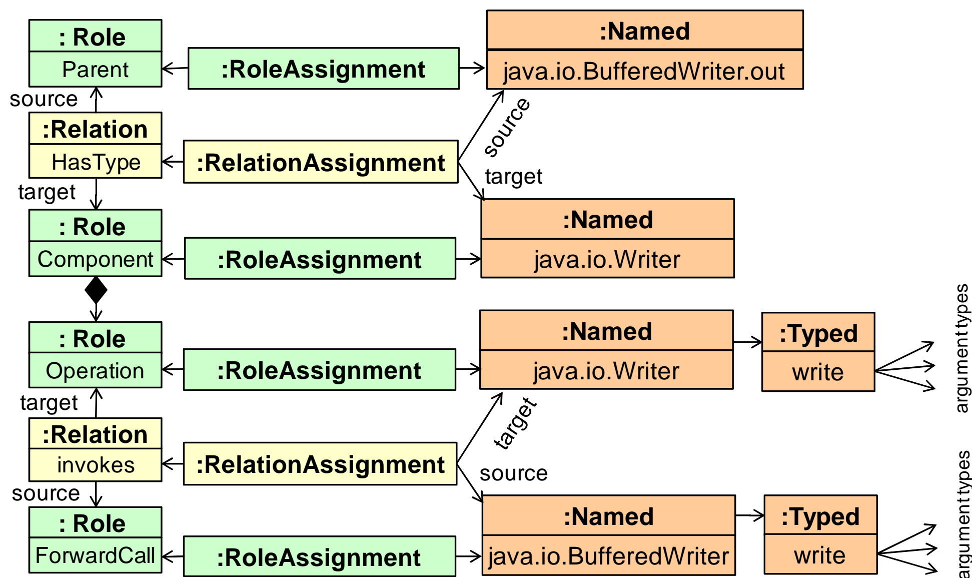

Figure 6.11: Sample of the design pattern detection results

6.4 DPDX implementation

The DPDX meta-models are a common framework of reference for developing and implementing the textual output of DPD tools and the parsing / interpretation of this output by users of DPD results (developers, the fusion tool, benchmarks and visualization tools). 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. Thus, XSLT can be used to transform results into a readable format or the proprietary format of individual tools. Furthermore, the rules of the format can be easily defined by XSD. In the appendix, figures B.10, B.11, B.12 and B.13 present the implementation of DPDX via the previous example of Decorator. Below, we will describe the implementation details through this example.

6.4.1 Implementation details

The implementation of DPDX consists of the realization of the three meta-models introduced in Section 6.3. The implementation of the Schema meta-model (see Section 6.3.1) allows the tools to report the schema of the patterns they search for; the Program Element meta-model (see Section 6.3.2) implementation is for identifying the program elements of the source code playing some role in the pattern instance; and the Result Meta-model (see Section 6.3.3) implementation describes the detected pattern candidates themselves.

To keep the implementation simple, we have adhered as much as possible to the following general principles for mapping meta-models to XML:

- classes of the meta-models are mapped to XML tags,
- attributes of the meta-model elements are mapped to attributes of the XML elements,
- aggregation between the elements of the meta-models is represented by the parent-children nesting technique of XML,
- 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. The referencing attribute is labelled as the meta-model association that it implements.
- an association with target cardinality greater than 1 is represented by a group element included with individual referencing elements. For instance, the TypedElement' nodes contain a 'ref' element which collects further 'ref' elements to refer to the types of the parameters of the 'TypedElement' (Figure B.11, line 17-21).

To avoid any ambiguities, we require that intentionally missing values be made explicit by special reserved values (enclosed in % signs) instead of simply providing empty attributes. In particular,

- the 'variantOf' attribute of a 'PatternSchema' element must have the value %NONE% if the respective schema is not a variant of another schema (Figure B.10, line 1);

- the 'player' attribute of a 'Role' element must have the value %MISSING% if no program element that plays this role could be found (see Figure B.13, line 15).

We chose to slightly deviate from these general principles when we felt it would make the implementation clearer and easier to understand:

- To avoid cluttering the format with redundant information, the 'kind' attribute of program elements is not set if it is implied by the program element type (for example, the 'Block' node in Figure B.11, line 23).
- The result metamodel defined an aggregation between 'Justification' - 'RoleAssignment' and 'Justification' - 'RelationAssignment' elements. However, in the implementation we represent 'Justification' elements separately and not as the children of the assignment nodes. 'Justification' tags can refer to a 'RoleAssignment' or to a 'RelationAssignment' with the 'for' attribute (Figure B.13, line 26).

6.4.2 Integration and visualization

The implementation of the three metamodels could have been placed into three different XML files. However, we chose to fuse them, since it eases processing and visualization. This means that the 'ProgramElements' and the 'PatternSchema' tags have been inserted into DPDX files as the children of the 'DPDResult' tag.

In order to support human readability of the format, an XSLT transformation file is also provided. It transforms DPDX files into nicely formatted HTML tables, whereby the source code of the pattern candidates can be loaded immediately. The HTML representation of the example DPDX presented in this section is available online [31]. Furthermore, the exact rules of the XML format are defined by an XSD schema file, which is also available online [31].

6.5 Summary

Design pattern detection is a significant part of the reverse engineering process that can aid program comprehension, and to this end several design pattern detection tools have been developed. However, each tool reports design pattern candidates in its own format, prohibiting a comparison, validation, fusion and visualization of their results. Apart from this limitation, each pattern identification approach employs different terms to describe concepts that underlie the pattern detection process, further inhibiting their synergetic use.

In this chapter we proposed DPDX, a common exchange format for design pattern detection tools. The proposed format is based on a well-defined and extendible metamodel that addresses a number of limitations of current tools. The XML-based format

employed is adaptable for existing and future tools, providing the basis for improving accuracy and recall scores when their findings are combined. Moreover, we strove to clarify central notions in the design pattern detection process by providing a common format for researchers and tools.

Part III

Evaluation of reverse engineering tools

"All our knowledge has its origins in our perceptions."
Leonardo da Vinci

Chapter 7

Validation of reverse engineering tools

This part introduces the further development of 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). 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 include design pattern detectors, duplicated code detectors and coding rule violation checkers. BEFRIEND largely differs from its predecessor (DEEBEE) in five aspects:

- it allows the uploading and evaluating of results related to different domains (domains like duplicated code detectors and design pattern miners)
- it allows the adding and deleting of the evaluating aspects of the results in an arbitrary way, while DEEBEE has fixed evaluation aspects
- it improves and extends the user interface
- it generalizes the definition of 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 (see Section 5.1.2)
- it allows the uploading of files in different formats by introducing a plug-in oriented architecture

7.1 Background

As we saw in Section 5.1.2, it may happen that several candidates can be grouped, which can help speed up their evaluation, improve the statistics and facilitate a comparison of

the tools. For example, if two clone detector tools found 500 clone pairs (most clone detector tools find clone pairs), then by grouping them, the number of clone pairs can be reduced to a fraction of the original candidate number. In another case, if one of the clone detectors finds groups of candidates (e.g. 30), and the other one finds clone pairs (e.g. 400), the reason why the latter tool finds more candidates could be that its output is defined differently. Because of this, it may be the case 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. There may be several reasons for this, such as:

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

7.1.1 Sibling relation

In order to group candidates, their relation needs to be defined. If two candidates are related to each other, then they will be denoted as *siblings*. Basically, three things determine the existence of the sibling relation between two candidates. These are

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

By using these three things, candidates can be connected. In the following we will examine these three cases in detail.

Source code positions Sibling relations are mostly determined by the matching of the source code positions. We examined the literature and found that the *contained*, the *overlap* and the *ok* metrics defined by Bellon et. al. [10] are a good starting point for a general sibling relation. We adapted the *contained* and *ok* metrics with a little modification via a *contain* function. The main reason for the modification was that the *contained* and *ok* metrics work on clone pairs, but we found that some tools report candidates in groups (e.g. clone groups or grouped design pattern candidates).

Let P and $P^{'}$ be participants with the same roles[1]. The *contain* and the *overlap* functions are defined in the following way:

$$contain(P, P^{'}) = \max(\frac{|P \cap P^{'}|}{|P|}, \frac{|P \cap P^{'}|}{|P^{'}|})$$
$$overlap(P, P^{'}) = \frac{|P \cap P^{'}|}{|P \cup P^{'}|}$$
$$\text{where } P.role = P^{'}.role$$

In the case of the *contain* and the *overlap* functions, the set operations are applied to the source code lines of P and $P^{'}$. Both $contain$ and $overlap$ functions take values between 0 and 1. Now, we will describe the *contain* and *overlap* functions through the example presented in Figure 7.1.

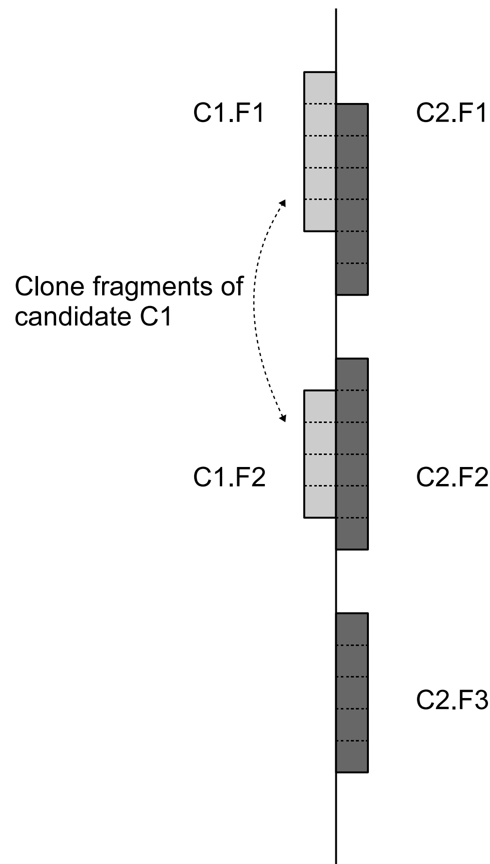

Figure 7.1: Example code for contain and overlap functions

The figure is based on an application of the *contained*, *good* and *ok* metrics published by Bellon et. al. [10]. In this figure the vertical line in the middle represents the whole

[1]If the roles are not the same then, for example, a ConcreteStrategy participant will be incorrectly connected to a Context participant.

source code linearly. The source code begins at the top of the vertical line and ends at the bottom of the vertical line. The code fragments of the participants are represented by the filled rectangles. Here, there are two clone candidates: *C1* on the left hand side and *C2* on the right hand side. *C1* has two cloned fragments (participant), *C1.F1* and *C1.F2*, while *C2* has three cloned fragments, *C2.F1*, *C2.F2* and *C2.F3*. Subsequent source code lines are separated by dashed lines inside the clone fragments, e.g. *C1.F1* contains 5 lines of code. For example, in the case of *C1.F1* and *C2.F1*, the functions are calculated in the following way:

$$\begin{aligned} contain(C1.F1, C2.F1) &= \max\left(\frac{|C1.F1 \cap C2.F1|}{|C1.F1|}, \frac{|C1.F1 \cap C2.F1|}{|C2.F1|}\right) \\ &= \max\left(\frac{4}{5}, \frac{4}{6}\right) = \frac{4}{5} = 0.8 \end{aligned}$$

$$\begin{aligned} overlap(C1.F1, C2.F1) &= \frac{|C1.F1 \cap C2.F1|}{|C1.F1 \cup C2.F1|} \\ &= \frac{4}{7} = 0.57 \end{aligned}$$

With $match_p$, either the contain or the overlap function is represented (exclusive or):

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

The $match_{pb}$ function describes whether P and P' have a $match_p$ above a predefined *bound*:

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

With the use of the $match_{pb}$ function, a $match_i$ function can be defined between the candidates. The $match_i$ function denotes how many times $match_{pb}$ returns one for each participant of the C and C' candidates:

$$match_i(C, C', bound) = \sum_{P \in C} \sum_{P' \in C'} match_{pb}(P, P', bound)$$

Let us see how $match_i$ is calculated in the case of the previous example. The first step is to calculate the selected function of $match_p$ (either overlap or contain) for each possible combination of participants. Table 7.1 shows these values for C1 and C2. Afterwards, $match_{pb}$ is calculated based on a predefined bound, which is 0.7 in this example. In Table 7.1 we denoted elements above this bound in bold. It means that $match_{pb}$ is one in these cases, which is used in the calculation of $match_i$. In this way, we get the following:

- In the case of $match_p = contain$, summarizing the $match_{pb}$ values (in Table 7.1, bold values showing that $match_{pb}$ is one) we find that $match_i = 2$
- In the case of $match_p = overlap$, each $match_{pb}$ value is 0, so $match_i = 0$

Function	Contain			Overlap		
P/P'	C2.F1	C2.F2	C2.F3	C2.F1	C2.F2	C2.F3
C1.F1	**0.8**	0	0	0.57	0	0
C1.F2	0	**1**	0	0	0.67	0

Table 7.1: Contain and overlap values for the previous example

Minimal number of matching participants. Sometimes it is not enough to match just one participant between two candidates. For example, in the case of design patterns, the Abstract Factory pattern has two fundamental participants (mandatory roles), called Abstract Factory and Abstract Product, for matching. For this reason, it is important to determine the minimal number of common participants of two potential siblings. Let us denote the minimal number by *m*. The *sibling relation* between candidates C and C' with parameters *m* and *bound* is defined in the following way:

$$sibling(C, C', bound, m) = \begin{cases} true & \text{if } match_i(C, C', bound) >= m \\ false & \text{otherwise} \end{cases}$$

The candidates can be grouped based on the sibling relation. A *group* contains candidates that have a sibling relation in pairs. By using groups, the evaluation of the tools is more straightforward, and the statistical results are better than without it (see Section 5.1.2 for a detailed discussion). In BEFRIEND, users can customize the sibling relations by arbitrarily choosing between the *contain* and the *overlap* functions, giving the *bound* and *m* parameters, and optionally selecting the *roles* for matching.

Continuing the previous example, let $m = 1$, $match_p = contain$ and $bound = 0.7$. Besides these parameters, C1 and C2 are placed into the same group.

Domain dependent name matching In certain domains, the roles are not so important (e.g. code duplications have only one role called *duplicated code*). However, if a domain contains several roles, some roles may be more important than the rest (as in the case of the design pattern domain). For example, in the case of a Strategy design pattern, the Strategy participant determines a candidate and hence, the sibling relation should be based on this participant. With such domains, the $match_{pb}$ function has to be modified.

Let $roles$ be a set that denotes the roles that are the basis of the sibling relations among the candidates. The $match_{pb}$ is redefined as $match_{pb'}$ in these cases:

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

In the case of domains where roles are important (e.g. design pattern miners), the $match_{pb}$ function has to be replaced with $match_{pb'}$ in the formula of $match_i$.

Future extensions With the previous definitions, it is possible that a group contains two or more candidates which are not siblings. Figure 7.2 shows an example of this problem. The previous example has been extended with one more candidate C3. In this way, grouping candidates based on the original sibling relation means that C1 and C3 are placed in the same group. As we do not allow such cases, the groups have to be transitive. Currently, in these cases the common candidates appear in each group. In the current example it means two groups: (C1, C2) and (C2, C3).

Overall, it may bias the statistics and the comparison because some candidates may be repeated across the groups (like C2). Therefore, this problem should be dealt with in the future. A trivial solution for this problem is to apply different grouping methods per use case:

- In the case of manual evaluation, it is easier to evaluate related candidates together so a compliant grouping is enough. E.g. in the current example, it means only one group: (C1, C2, C3).
- Comparing candidates requires transitive, but not distinct groups. E.g. in the current example it means two groups: (C1, C2) and (C2, C3).
- Calculating statistics requires transitive and distinct groups, E.g. in the example, it means three groups: (C1), (C2) and (C3).

At present, the benchmark supports the first strategy for each use case. However, it could be easily extended with the two other approaches.

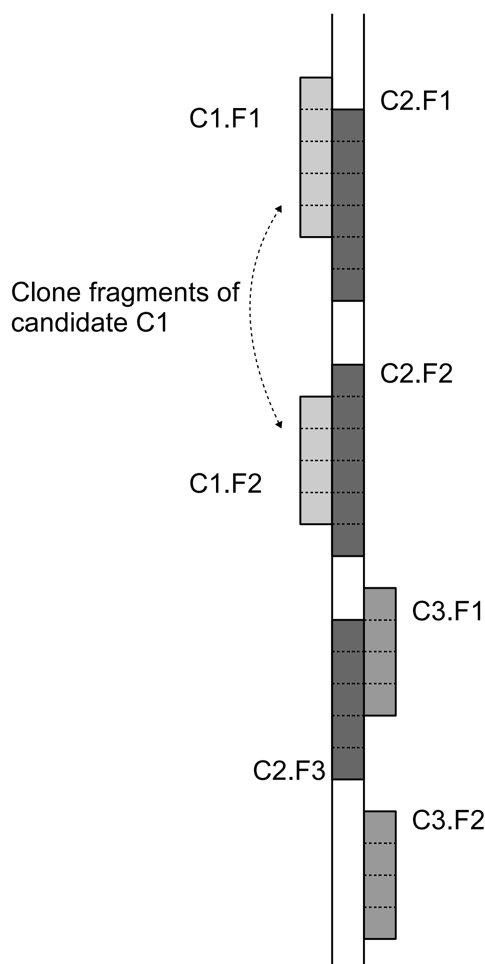

Figure 7.2: Problem of the non-transitivity feature of the sibling relation

7.2 Use scenarios

Now we will show how the system works with the help of some scenarios graphically illustrated below. In each case, first we give a general description of its functionality, then we demonstrate its use with a concrete example. The examples are continuous, they are built on each other, and they show how the benchmark is used. The first example begins with an empty benchmark without any kind of domain or evaluation criteria.

7.2.1 Setting up the database

In this scenario, we show how the user can create a new domain and evaluation criterion, how he can upload data in the system, and how he can set the sibling algorithm parameters. The functions that help one make the necessary settings can be found in the *Settings* and *Upload* menus.

Creating a new domain Now a new domain has to be created based on the data to be uploaded. For the creation of a new domain, the *Domain settings* panel must be used.

Figure 7.3: Creating a new domain

Example: We will upload the results of a duplicated code detecting tool. First, with the help of the *Domain settings* panel, we create a new domain which is called *Duplicated Code*. As a result, the actual domain is set to the newly created Duplicated Code (see Figure 7.3). 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. The functionality of BEFRIEND depends on and is performed on the active domain (e.g. querying candidates, evaluating candidates, statistics and comparison).

Creating new evaluation criteria In order to be able to evaluate the uploaded data, appropriate evaluation criteria are required. The user can create an arbitrary number of criteria per domains. 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. Similar to the domain case, we can create a new criterion under the *Settings* menu. When creating a new criterion, the following data should be entered:

- The title of the evaluation criterion (Title).
- The question related to the criterion (Question).
- The possible answers to the question (Choice). For each answer a percentage ratio should be added 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.
- Should the *precision* and *recall* values be calculated? (Y/N)

Example: We will create an evaluation criterion called *Correctness* for the previously created *Duplicated Code* domain. Enter a value of 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 and clicking on the *Submit* button, the criterion appears in the

How correct is it?

I am sure that it is a real duplicated code class.(100%)
I think that it is a real duplicated code class.(66%)
I think that it is not a real duplicated code class.(33%)
I am sure that it is not a real duplicated code class.(0%)
Compute precision and recall values for this criteria. Yes
Delete Modify

Figure 7.4: Correctness criteria

setting surface (see Figure 7.4). The *Correctness* criterion is used to decide to what extent a code clone group comprises cloned code fragments. For the criterion, we also have to say whether the precision and recall values will be calculated or not.

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 given candidate 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 from a refactoring viewpoint.

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

- 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 would be gained by placing the code fragments into a function. It may happen that something is easily replaceable, but it is not worth the trouble since no benefit is gained by doing so.

Uploading data into the benchmark When the user has created the appropriate domain and the required evaluation criteria, he has to upload the candidates produced

Figure 7.5: Uploading data into BEFRIEND

by the tools to be evaluated. The upload page can be found in the *Upload* menu. The format of the files to be uploaded is completely arbitrary, the only condition being that the evaluated tool should provide a BEFRIEND plug-in to perform the uploading. The benchmark provides a plug-in mechanism whereby 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 items are needed: the name of the tool and the name and programming language of the software being analyzed. If an item was not present earlier, we can add the new data to the list by selecting the *New* item option. The uploaded data refer to a certain source code, so it is important to provide some kind of access to these sources. Currently, the benchmark supports two access methods - *http* and *svn*.

Example: Continuing 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 Figure 7.5). 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 allow candidates with similar properties to be handled together during an evaluation (see Section 7.1.1). Settings can be applied in the *Settings* menu, on the *Siblings settings* panel (see Figure 7.6). The user can choose the *Contain* or the *Overlap* function. In the *Path matching bound* field, the bound of the $match_{pb}$ relation should be between 0 and 100 (the $match_p$ relation is projected into the interval 0 to 100). In the *Min. number of matching participants* field, the number of participants whose matching we need for two sibling candidates should be given (the m parameter

of the *sibling* relation). Settings set by the user have to be saved in named configurations. Thus, if we wish to use an already saved setting in the future, we can load it by selecting it from the *Load shortcut* menu. In this case the system will not rerun the sibling grouping algorithm, but just reload the previously stored configuration.

Figure 7.6: Sibling settings

Example: In the last part of the scenario, we set the sibling parameters (see Figure 7.6). For linking, we use the *Contain* function and the matching bound is set at 90%. The reason for choosing this bound is that this way the candidates 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 candidates. We save this configuration as *Moderate*.

7.2.2 Data evaluation

The main task of the benchmark is to perform a visualization and evaluation of the uploaded data. In this section, we summarize what these functions do.

Query view The uploaded candidates can be accessed through the *Evaluation* menu. First, the *Query view* appears (see Figure 7.7), which helps the user define the aspects, on the basis of which the uploaded candidates are listed. There are four possible aspects: *Tool*, *Software*, *Language* and the currently active domain (Design Pattern, Duplicated Code or Rule Violation). The possible values of the aspects depend on the data of the already uploaded candidates. For example, the Design Pattern aspect contains values like State or Adapter, while Tool contains values like Columbus when the active domain is Design Pattern.

Please make your selection.

1. aspect: Software | JUnit4.1

2. aspect: Tool | All

3. aspect: None | All

4. aspect: None | All

☑ Connect siblings

Go to results view

Figure 7.7: Query view

With more settings, even narrower candidate sets can be defined. If the user would like to skip an aspect when filtering, this 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 order of the sequence of aspect selection is arbitrary. At the bottom of the query view, there is a *Connect siblings* check box. If this box is marked, the candidates appear in groups, not individually.

Example: Select the *Evaluation* menu. The *Query view* page appears (see Figure 7.7). In the figure, 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 also activated the *Connecting siblings* check box. By clicking on the *Go to results view* button, we will get to the view of candidates satisfying the given aspects (see Figure 7.8).

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																

Figure 7.8: Results view

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 candidate 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*. Blue identifiers represent a single candidate (or instance), while red identifiers represent grouped candidates (or instances).

Example: In the first column of the table, JUnit4.1 can be seen based on the aspects set in the Query view (see Figure 7.8). 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 candidates 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 candidate identifiers.

Group instance view and Individual instance view The Instance view is used to display candidates and evaluate whether they are instances or not. BEFRIEND supports two kinds of instance views, namely *Group instance view* and *Individual instance view*. While the *Group instance view* can display multiple (grouped) candidates simultaneously, the *Individual instance view* displays the participants of only one candidate in each case. Apart from this difference, the two views are essentially the same.

In the *Results view*, the *Instance view* can be accessed by clicking on a candidate identifier. If the user has set the grouping of siblings in the *Query view* (see Figure 7.7), the system automatically applies the *Group instance view*. Otherwise, it applies the *Individual instance view*.

Duplicated Code Instance Information

Software JUnit4.1
Duplicated Code CloneInstance

Participants	#31	#91	#256	#259
clone	✓	✓	✓	✓
clone	✓	✓	✓	×
clone	×	×	×	✓

Show criteria...

/JUnit4.1/org/junit/tests/ForwardCompatibilityPrintingTest.java(68)

```
	public static class ATest {
		@Test public void error() {
			Assert.fail();
		}
	}

	public void testErrorAdapted() {
		ByteArrayOutputStream output= new ByteArrayOutputStream();
		TestRunner runner= new TestRunner(new TestResultPrinter(
				new PrintStream(output)));

		String expected= expected(new String[] { ".E", "Time: 0",
				"Errors here", "", "FAILURES!!!",
				"Tests run: 1,  Failures: 0,  Errors: 1", "" });
		ResultPrinter printer= new TestResultPrinter(new PrintStream(output)) {
			@Override
			public void printErrors(TestResult result) {
				getWriter().println("Errors here");
			}
		};
		runner.setPrinter(printer);
		runner.doRun(new JUnit4TestAdapter(ATest.class));
		assertEquals(expected, output.toString());
	}

	private String expected(String[] lines) {
		OutputStream expected= new ByteArrayOutputStream();
		PrintStream expectedWriter= new PrintStream(expected);
		for (int i= 0; i < lines.length; i++)
			expectedWriter.println(lines[i]);
		return expected.toString();
	}
}
```

/JUnit4.1/junit/tests/runner/TextFeedbackTest.java(86)

```
			}
		};
		runner.setPrinter(printer);
		TestSuite suite = new TestSuite();
		suite.addTest(new TestCase() { @Override
		public void runTest() {throw new AssertionFailedError();}});
		runner.doRun(suite);
		assertEquals(expected, output.toString());
	}

	public void testError() {
		String expected= expected(new String[]{".E", "Time: 0", "Errors here",

		ResultPrinter printer= new TestResultPrinter(new PrintStream(output))

			@Override
			public void printErrors(TestResult result) {
				getWriter().println("Errors here");
			}
		};
		runner.setPrinter(printer);
		TestSuite suite = new TestSuite();
		suite.addTest(new TestCase() { @Override
		public void runTest() throws Exception {throw new Exception();}})

		runner.doRun(suite);
		assertEquals(expected, output.toString());
	}

	private String expected(String[] lines) {
		OutputStream expected= new ByteArrayOutputStream();
		PrintStream expectedWriter= new PrintStream(expected);
```

Figure 7.9: Group instance view

Group instance view: The *Participants* table comprises the participants of the candidates; the different participants of all the siblings appear. The table has as many

columns as the number of candidates in the group. Each column corresponds to a given candidate whose identifiers are indicated at the top of the column. In the intersection of a row of participants and a column of candidates, either a green $\surd$ or a red $\times$ symbol appears. The green $\surd$ means that the candidate in the given column comprises the participant in the row. The $\times$ symbol means that this particular candidate of the group does not have 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. Whether the source code should be displayed on the left or right hand side of the surface under the table can be selected from the menu. This is very useful because, for instance, the participants of the duplicated code candidates lie next to each other.

The evaluation criteria and, along with these, the voting user interface can be accessed by clicking on the 'Show criteria' link under the table of participants (see Figure 7.9). Statistics about previous votes of the candidates can be accessed by clicking on the *stat* link above the evaluation column. 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 candidates. A candidate can be commented on by clicking on the candidate identifier appearing above the Participants table. This window even contains information that reveals which tool found the given candidate.

Example: Click on candidate #31 in the *Results view* created in the previous example (see Figure 7.8). The *Group instance view* appears (see Figure 7.9). Right-click on any of the green $\surd$ symbols, and select *Open to left* in the appearing pop-up menu. Right-click on another green $\surd$ symbol, and select *Open to right* in the menu. This way, the source code of two clone participants will be displayed next to each other. After examining the participants of all the four candidates belonging to this group, the evaluation criteria can be displayed by clicking on the 'Show criteria' link. Here, the candidates can be evaluated.

Statistics view This view can be reached from the *Query view* by clicking on a link other than a candidate identifier. Here, relying on the structure seen in the *Results view*, the user gets some statistics according to the evaluation criteria applied, based on the previous user votes (see Figure 7.10). One table that comprises the vote statistics for all of the candidates concerned is associated with each evaluation criterion. In the first column of the table, the candidate identifiers appear. The identifier of the grouped candidates is red, while that of the others is blue. Besides the identifiers, five columns can be found (mean, deviation, minimum, maximum and median) representing statistics about user votes for the corresponding candidate[2]. If we have decided in the *Settings*

[2] On the top of the statistics view the user can set the strategy of how to aggregate the votes of a user in the case of grouped candidates (red identifiers): based on the average, maximum, minimum, deviation or median of the group candidate ratio. In the case of groups, these values (for each user) were used to calculate the statistics shown in Figure 7.10.

menu that the precision and recall scores 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%

Figure 7.10: Bauhaus correctness statistics

Example: After having evaluated all the candidates found by the tools, we examine the statistic we get for the votes of each tool. Let us go back from the *Group instance view* (see Figure 7.9) to the *Results view* (see Figure 7.8) by clicking on the *Back to previous view* link. Here, we get the statistical 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 candidate votes can clearly be seen (see Figure 7.10). Furthermore, it can also be seen that some candidates are grouped, these being marked in red by the system. Underneath the table, information is provided like precision and recall scores. The *Number of instances above the threshold* and *Total number of instances above the threshold* are based on a setting at the top of the statistics view. It is called the *Threshold for calculating precision and recall*, which by default is 50%. This value determines which candidates can be accepted as instances (as true positives). The first three items of information (starting with 'Number of') refer to the results of the current tool, while the three items of information below Precision (starting with 'Total number') refer to the results of all the tools examined.

On the basis of this data, precision and recall scores are automatically calculated. In the case of Bauhaus the first is 62.79%, while the second is 84.38% .

Comparison view The system comprises yet another interesting view called the Comparison view. In order to activate this view we have to start from a statistics view that

comprises several tools. The comparison view compares the instances found by each tool. For each possible coupling, it defines the instance sets found by two tools, and the difference and intersection of these sets. This view allows us 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									

Figure 7.11: 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 Figure 7.11). A comparison of tools is carried out in pairs, and here we also have the opportunity to link siblings in the same way as in the other views (here the grouped instances are marked in red as well).

7.3 Experimental results

Now we will summarize the results of the experiments performed using BEFRIEND. We should mention here 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 demonstration, five *duplicated code* finder tools were assessed on two different open source projects, called *JUnit* and *NotePad++*. The tools used in the experiments were *Bauhaus* (*clones* and *ccdiml*), *CCFinderX*, *Columbus*, *PMD* and *Simian*.

Over 700 duplicated code candidates were evaluated by two developers. For the evaluation, three evaluation criteria were used, namely *Correctness*, *Procedure abstraction* and *Gain* (see Section 7.2.1). The results of using the tools on the two open source projects are shown in Table 7.2 and Table 7.3.

The *precision* and *recall* scores shown in the tables were calculated based on the Correctness criteria. In the rows of *Procedure abstraction* and *Gain*, the average values obtained from the votes given for the candidates are shown according to the criteria applied. We should add that a *threshold* value is needed to calculate the precision and recall scores.

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%

Table 7.2: Results on NotePad++

The candidates above this threshold are treated as instances (see Section 7.2.2). We used the default threshold value of 50%, but the threshold can be adjusted arbitrarily. For example, in the case of three voters, if two of them give 66% to a candidate, while the third one gives 33%, the average of the three votes is 55%. In such cases, it is reasonable to accept the candidate as an instance since two of the three voters accepted it, while only one rejected it.

Criteria	Bauhaus clones	CCFinder	Colum-bus	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 7.3: Results on JUnit

CCFinderX is missing from Table 7.2 because it produced a huge number of hits with the parameters we used (it found over 1000 duplicated code candidates containing at least 10 lines). We were only able to evaluate the four other tools on NotePad++.

We learned some useful things during the evaluation. In the case of JUnit, PMD and Simian both produced a very similar result to Columbus's, but our experience is that in general, the *token* based detectors (Bauhaus-clones, CCFinderX, PMD, Simian) produce a 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 candidates of at least 10 lines, which are also syntactically coherent, the token-based detectors mark the clones that are in many cases shorter than 10 lines and 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). For example, a clone candidate can mark the last line of a method and the first line of the subsequent method. Based on this, the evaluation could be extended with a new criterion. The new criterion would apply to the accuracy of the marking of a clone candidate. It would also define what portion of the marked code is in the instance.

Based on the *Gain* and *Procedure abstraction* values of certain tools, we can say that the ASG-based detectors find fewer but mostly *more valuable* and *easily refactorable* clone

instances. In contrast, the token-based tools find more clone instances that produce a *more complete* result.

We should mention that we also imported the design pattern candidates from DEEBEE into BEFRIEND. In addition to these, the system contains coding rule violation candidates found by PMD and CheckStyle. In this way, BEFRIEND contains the results of three significantly different families (domains) of reverse engineering tools, namely duplicated code detectors, design pattern miners and coding rule violation checkers.

7.4 Summary

We developed BEFRIEND from our benchmark for evaluating design pattern miner tools called DEEBEE. During the development of BEFRIEND, we tried to generalize it in every way possible, e.g. an arbitrary number of domains can be created, domain evaluation aspects and the settings of candidate siblings can be customized. To upload the results of different tools, the benchmark provides a plug-in mechanism. We applied BEFRIEND to three reverse engineering domains, namely design pattern mining tools, code clone mining tools, and coding rule violation checking tools. The evaluation results stored in DEEBEE were migrated to BEFRIEND, and in the code clones domain we applied the benchmark using more examples. The sibling mechanism of DEEBEE was also improved to support other domains as well.

The work done here is the first step towards creating a generally applicable benchmark that can help one to evaluate and compare many kinds of reverse engineering tools. In the future, it would be good to get the opinions and advice of reverse engineering tool developers about our benchmark to achieve and satisfy all their needs. BEFRIEND is freely available to the public via the link **http://www.inf.u-szeged.hu/befriend/**

*"In all human affairs there are efforts, and there are results,
and the strength of the effort is the measure of the result."*
James Allen

Chapter 8

Conclusions

The maintainability of (legacy) software systems usually becomes increasingly difficult after several releases of the system. It may be caused by the fact that important parts of the software development process were left out because of strict deadlines. These phases typically include the specification of the system, designing the system architecture and testing the deliverables. However, legacy systems provide important, valuable and frequently irreplaceable functionalities, hence the system has to be recovered. Rewriting the system from scratch is costly, time consuming, and there is a big chance of failure. In contrast, reengineering the system in small steps is relatively cheap and safe. The success of reengineering largely depends on the reverse engineering phase. Reverse engineering provides high level information about the source code, e.g. in the form of design patterns, coding rule violations and duplicated code fragments. The problem is that these tools sometimes produce false positives, which might lead to the overall failure of the whole reengineering project. In this thesis we provided techniques, tools and experiments for improving, evaluating and comparing the performance and correctness of reverse engineering tools.

First, we proposed a machine learning–based approach to improve the results of our design pattern miner tool. We extended the architecture of the Columbus framework and integrated machine learning tools. Experiments were also performed on a real–life software system to demonstrate the feasibility of our approach.

Next, we evaluated and compared three design pattern miner tools called Columbus, Maisa and CrocoPat. These tools were evaluated in terms of speed, memory consumption and the differences between the candidates. In the case of CrocoPat, the necessary Columbus plugin was also developed to generate the input for the tool. Furthermore, the corresponding pattern definitions in its own pattern description language (RML) were defined by us.

In the next part, an online benchmark (DEEBEE) for evaluating and comparing design pattern miner tools was presented. In this study, based on the results of Chapter 4, the

grouping of similar candidates was introduced to provide reliable statistics and comparisons. Experiments were also performed to demonstrate the capabilities of the benchmark. Then the benchmark was evaluated based on the requirements defined by Sim et al. [74].

We made the CSV format of DEEBEE as simple as possible, but it became insufficiently expressive. Furthermore, we found that formats of design pattern miner tools had certain deficiencies or problems. Based on this motivation, a common exchange format was proposed in Chapter 6. First, we defined requirements for the format, and then, taking into account the requirements, we elaborated on the metamodels of the format. The format was implemented in XML, based on the entities and relations defined in the metamodel.

In Chapter 7 we introduced BEFRIEND, a benchmark for reverse engineering tools. BEFRIEND is based on DEEBEE and is a fully generalized version of it. First, it supports the evaluation and comparison of different kinds of reverse engineering tools via domains. BEFRIEND supports arbitrary evaluation criteria for each tool domain. It also handles and groups similar candidates based on a generalized sibling relation. The benchmark user interface was also applied on real-life examples.

There are several ways our work might be extended in the future. For example, noting the first thesis point, the proposed machine learning based improvement method could be adapted to other kinds of reverse engineering tools (e.g. bad code smell [36] miners). As for the benchmarks presented(DEEBEE and BEFRIEND), other case studies and new reverse engineering domains could be introduced like that done in the area of change impact analysis tools [85]. In addition, benchmarks could support performance measurement information as well. The DPDX format might be extended in different ways. For instance, it could be compared with other approaches (e.g. GXL) or it could be evaluated and improved through the collaboration of design pattern miner tool creators.

Appendix A

Related Work

A.1 Design pattern mining

One of the first studies in this topic is the one describing the BACKDOOR (Backwards Architecting Concerned with Knowledge Discovery of OO Relationships) [71] inductive method. The method provides guidelines for analyzing existing object-oriented software systems and reverse architecting design patterns. The crucial part of the method is the analysis of design pattern candidates, namely to select the characteristics that properly identify them. This is the most human-intensive part. The purpose of this part is to compare the potential patterns found against the set of reference patterns. The basis of the comparison is not only their structure but also their semantics. The pattern candidates were classified according to their correspondence with the reference design patterns. One aspect of the classification was the implementation and another was the purpose. The measures of a classification are complete match and partial match, thus they have a four level classification of possible patterns. The technique permits the creation of a pattern knowledge base whose main purpose is to help development organizations create their own custom pattern library.

Kaczor et al. [50] proposed a bit-vector algorithm for design pattern identification. The algorithm initialization step converts the design pattern motif and the analyzed program model into strings. To model the design patterns and the analyzed program, six possible relations can be used between elements: association, aggregation, composition, instantiation, inheritance and dummy. The authors gave an efficient Iterative Bit-vector Algorithm to match the string representation of the design patterns and the analyzed program. They compared their implementation with explanation-based constraint programming and metric-enhanced constraint programming approaches.

Costagliola et al. [22] based their approach on a visual language parsing technique. The design pattern recognition was reduced to recognizing sub-sentences in a class diagram,

where each sub-sentence corresponds to a design pattern specified by an XPG grammar. Their process consist of two phases: the input source code is translated into a class diagram represented in SVG format; then DPRE (Design Pattern Recovery Environment) recovers design patterns using an efficient LR-based parsing approach.

Tonella and Antoniol [82] presented an interesting approach to recognize design patterns. They did not use a library of design patterns as others did but, instead, discovered recurrent patterns directly from the source code. They employed concept analysis [72] to recognize groups of classes sharing common relations. The reason for adapting this approach was that a design pattern could be considered as a formal concept. They used inductive context construction which then helped them to find the best concept.

Albin-Amiot *et al.* [1] introduced a Pattern Description Language (PDL) that was suitable for detecting design patterns from source code and also for generating source code. Their system was also able to detect some distorted versions of design patterns and could repair them automatically with the aid of a source-to-source transformation engine.

Keller *et al.* [52] argued that design patterns form the basis of many of the key elements of large-scale software systems, so to comprehend these systems they needed to recover and understand design patterns. They emphasized not only the design's structure, but its rationale too. They utilized the SPOOL environment which provided tools for analyzing existing source code and recovering design components like design patterns. They implemented query mechanisms that could recognize the structural descriptions of patterns in the source code models. The SPOOL environment gave some visual information about the query's results (the design patterns found) and information about the design pattern's class diagram to discover the pattern's structure and documentation about its intent and motivation. They used this environment with three industrial systems and searched for three design patterns (Template Method, Factory Method and Bridge). They checked the intent of the design patterns found and noticed that the discovered design patterns' intents did not necessarily correspond to the original design patterns' intents.

Asencio *et al.* [6] used the Imagix [45] tool to parse the source code. The tool built a database of program entities and relationships. They introduced a recognizer specification language where they made declarative specifications – logical conditions – to describe design pattern structures. Afterwards, their tool called Osprey automatically generated Python source code from these declarative specifications, which searched patterns in the database generated by Imagix. They tested their system on several software systems and obtained promising results, but also found false positives. They classified the causes of these false hits. The first class of problems were the front end analyzer errors. The second class was the pattern ambiguity. The cause of these false hits were the structural similarities among design patterns, like those of the Decorator and Proxy patterns. The third class of problems were the partial patterns. Actually, there were many situations where the full pattern was not present in the code.

Campo *et al.* [18] utilized design pattern recognition for framework comprehension. They concluded that some design patterns could be distinguished only by their dynamic behaviour, because their structures were the same (e. g. Composite vs. Decorator and State vs. Strategy).

A.2 Improvement of design pattern mining

The accuracy of design pattern discovery has been improved by some researchers using dynamic analysis which, apart from the structural information, also considers runtime information of the system in question [89, 90]. First, the author performed a static analysis and labeled each pattern candidate with a fuzzy value that represented the correctness of the candidate. Afterwards, false candidates were ruled out by dynamic analysis and by using these fuzzy values.

Guéhéneuc et al. [44] introduced a method for reducing the search space for design patterns. First, they analyzed several programs manually and searched for source classes that act together as design patterns and set up a repository from them. Afterwards, they parsed these programs with tools to obtain models of their source code, and computed metrics from these models (like size, cohesion and coupling). In the next step they ran their rule learner algorithm that returned a set of rules characterizing the design pattern participants by the metric values. This way, they obtained rule sets (called a fingerprint) for the participant classes of the design patterns. Based on these fingerprints, unknown classes were characterized. They then integrated this fingerprint technique into their constraint-based tool suite to reduce the search space. Their work is in some sense similar to ours, but we made use of machine learning after the structural matching phase (taking into consideration the whole design pattern and not just individual classes) to filter out false candidates.

Antoniol *et al.* introduced pattern recognition by using metrics [4]. They analyzed source code and class diagrams and created AOL (Abstract Object Language) specifications containing information about classes, their members and relations. The design pattern descriptions were also stored in AOL format. Next, they created an AST (Abstract Syntax Tree) from the AOL specification. Afterwards, they computed metrics for each candidate class from the AST and created a set for each participant class in the searched design pattern containing only those candidate classes which met the participant classes' metric conditions (these conditions were set up *manually*). This way, they significantly reduced the search space. Next, they checked the required structural relations among the candidate classes in these sets. They were also able to verify the consistency between the code and the design. They then tested their system on public and industrial systems and got good results.

As we mentioned previously, a lot of effort has gone into improving the design pattern mining process. The approaches focus on two aspects of the design pattern mining process, namely

- improving the results in the last step of the process via a dynamic analysis
- reducing the search space by incorporating some human knowledge into the process

A.3 Evaluation of reverse engineering tools

Sim et al. [74] 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, namely accessibility, affordability, clarity, relevance, solvability, portability and scalability. Sim gives a more detailed description and examples in her PhD thesis [73].

Nowadays, more and more papers deal with the evaluation of reverse engineering tools. 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 for a given task.

Evaluation of design pattern mining tools

Petterson et al. [63] summarized problems during an 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 the 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, hence solutions taken from natural language parsers were 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 that of a 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 in a real-life software system a single group is not able to determine a complete control set. They stated that a *community effort is required* to make control sets for a given set of applications.

Guéhéneuc et al. [43] introduced a comparative framework for design recovery tools. The purpose of the authors' framework was not to rank the tools, Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java [65]) and LiCoR (Library for Code Reasoning [55]), but to compare them on the basis of their qualitative aspects. This framework contained eight aspects called context, intent, users, input, technique, output, implementation and tool. There is a major need for such a framework since making a comparison among the several design recovery tools is very difficult due to the fact that they have quite different characteristics in terms of representation, output format and implementation techniques. This framework provides an opportunity for comparing not only similar systems, but also systems which are different. Our study is different from this one because we used a *benchmark* to evaluate design pattern instances and to compare design pattern miner tools based on their results.

Dong et al. [30] introduced a systematic review of design pattern mining techniques. The authors presented a comparative study on different aspects of several approaches of design pattern mining. These aspects include the type of the algorithm applied (structural, behavioural, etc.) and supported programming languages (C++, Java, etc.), system representation (AST, matrix, etc.). Furthermore, the authors found that different approaches yield different results, and they collected the possible reasons for this, like missing roles, role types, missing relationships, delegation implementations and merging roles. In this sense, this work is similar to that presented in Chapter 4, but it was published later.

Arcelli et al. [5] proposed three categories for design pattern mining tools, considering the information which was used during the detection process. These categories are the "entire" representation of design patterns, the minimal set of key structures that a design pattern consists of, and the sub-components of design patterns. They dealt with the last category, and two tools for this called FUJABA and SPQR were introduced and compared. The basis of the comparison was how a tool decomposes a design pattern into smaller pieces. The conclusion was that the decomposition methods of the two examined systems are very similar, and after they argued the benefits of sub-patterns.

P-MARt, an XML-based repository of micro-architectures similar to design patterns, was introduced by Guéhéneuc [40]. In each session, he asked students to analyse a new system and detect design patterns manually. Furthermore, he collected micro-architectures from his colleagues as well. The micro-architectures are stored in a XML file.

Evaluation of other reverse engineering tools

Bellon et al. [10] performed 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 were previously

unknown, so both the strengths and the weaknesses of the tools were discovered. Their benchmark provides a standard procedure for every new clone detector. Bellon also published a publicly available benchmark [11].

Rysselberghe et al. [69] compared different clone searching techniques (string, token and parse tree-based). For the sake of comparison, they developed reference implementations of the different techniques instead of using existing tools. During their evaluation, they asked certain questions, some of which correspond to the criteria introduced in BEFRIEND. One such question was 'How accurate are the results?' The different techniques were tested on five small-and medium-sized projects by using the evaluating questions. In another article, they compared the reference implementations of the clone searching techniques based on refactoring aspects [68]. These aspects were called suitable, relevance, confidence and focus.

Burd et al. [17] evaluated five clone searching tools on the university project called GraphTool. They also examined the problem of how to link the instances. For this reason, they used a simple overlap method. In the evaluation part, they applied the well-known precision and recall scores approach. In addition, they presented different statistics and comparison mechanisms, some parts of which are even now supported by BEFRIEND (e.g. intersection and difference of the results).

Wagner et al. [88] compared three Java bug searching tools on one university and five industrial projects. A 5-level severity scale, which can be integrated into BEFRIEND, served as the basis for comparison. Based on this 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 least serious one. The tools were compared not only with each other, but with reviews and tests as well. We should mention here both the reviews and the tests can be loaded into BEFRIEND by writing the appropriate plug-ins. In two other articles [86, 87] they also examined the performance of bug searching tools.

Ayewah et al. [7] evaluated the FindBugs tool on three large scale projects called 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. [67] evaluated and compared five tools called FindBugs, JLint, PMD, Bandera and ESC/Java. The evaluation was carried out on five 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 and the correlation among the hits of the tools was weak. This led them to conclude that rule violation searching tools definitely require an evaluation and comparison tool like BEFRIEND.

Sim et al. [75] developed the CppETS benchmark to evaluate C++ extractors. The benchmark used two categories, namely accuracy and robustness. It consisted of 25 test cases, 14 in the accuracy and 11 in the robustness category. The test cases were

typically less than 100 lines of code long, and none contained more than 1000 lines. They measured the performance by asking questions in a text file in all test cases. These questions have to be answered by the persons using the extractor. They evaluated four extractors called Ccia, cppx, Rigi and TkSee/SN.

Sim et al. [76, 77] described a live demonstration of program comprehension tools. Five software development teams were involved in this demonstration, namely Lemma, PBD, Rigi, TkSee and Visual Age C++. UNIX Tools such as grep and emacs were also demonstrated because of their popularity. The tools had to solve two reverse engineering and three maintenance tasks on the xfig drawing package. Afterwards, they collected the results of the tools and closed with a discussion of observations. They mentioned that *'the biggest difficulty for some teams was parsing the source code.'* This is a serious problem in design pattern mining too, so it is vital that the extracted facts about the source code are correct.

We already have some experience in developing a benchmark. The official code size benchmark for the GCC compiler called CSiBE [12, 23] is the result of our previous work (see http://gcc.gnu.org/benchmarks/). The primary purpose of CSiBE is to monitor the size of the code generated by GCC. In addition, the compilation time and the code performance measurements are also provided.

Appendix B

DPDX

B.1 Output formats of DPD tools

To emphasize the similarities and differences all formats are presented using the example of the 'Decorator' instance taken from Java IO, illustrated in Figure 6.10 (see Section 6.3.3). This is also done for tools that cannot analyze Java programs (like Columbus) or do not try to detect instances of the 'Decorator' pattern (like Fujaba). In such cases, we showed what the format would look like if the tool supported the 'Decorator' pattern mining from Java source code. In addition, we also reviewed the result representation format used by two DPD result repositories, namely P-MARt [61] and DEEBEE [97].

```
1<pattern name="Decorator">
2 <role name="Component">"Writer"</role>
3 <role name="Decorator">"BufferedWriter"</role>
4 <role name="ConcreteComponent">"OutputStreamWriter"</role>
5 <role name="ConcreteDecorator">"CharCountBufferedWriter"</role>
6 <role name="operation">"close"</role>
7</pattern>
```

Figure B.1: Output format of SPQR

INSTANCE	Component	ConcreteComponent	Decorator	ConcreteDecorator
Decorator[0]	Writer	OutputStreamWriter	BufferedWriter	CharCountBufferedWriter
Decorator[0]	Writer	OutputStreamWriter	BufferedWriter	BlackListBufferedWriter

Table B.1: DP-Miner result

```
1<StructuralAnnotation name="Decorator"
2     fuzzyBelief="56.6666666666664">
3 <BoundObject key="Component" name="java.io.Writer"/>
4 <BoundObject key="Decorator" name="java.io.BufferedWriter"/>
5 <BoundObject key="operation" name="write(char[],int,int)"/>
6</StructuralAnnotation>
```

Figure B.2: Output format of Fujaba

```
Solution 0
Component = Writer
Component.Operation() = Writer.close()
Decorator = BufferedWriter
Decorator.Operation() = BufferedWriter.close()
Decorator.component = BufferedWriter.out
ConcreteComponent = OutputStreamWriter
ConcreteComponent.Operation() = OutputStreamWriter.close()
ConcreteDecorator = CharCounterBufferedWriter
ConcreteDecorator.Operation() =
CharCounterBufferedWriter.close()
ConcreteDecorator.AddedBehavior() =
CharCounterBufferedWriter.extendStream()
```

Figure B.3: Output format of Maisa

```
1<pattern name="Decorator">
2 <instance>
3  <role name="Component" class="java.io.Writer"/>
4  <role name="Decorator" class="java.io.BufferedWriter"/>
5 </instance>
6</pattern>
```

Figure B.4: Output format of SSA

```
Source class(es) for pattern class Component:
/src/Writer.java(50): pattern class Component =
source class Writer
/src/Writer.java(323): pattern operation Operation =
source operation close

Source class(es) for pattern class ConcreteComponent:
...
Source class(es) for pattern class Decorator:
...
Source class(es) for pattern class ConcreteDecorator:
```

Figure B.5: Output format of Columbus

```
Decorator Pattern.
Writer is the Decoratee class.
BufferedWriter is the Decorator class.
Concrete Decorator classes:
CharCounterBufferedWriter BlackListBufferedWriter
flush() is the decorate operation
Files Location:
java/io/Writer.java, java/io/BufferedWriter.java
```

Figure B.6: Output format of PINOT

```
Decorator : 100%
component as java.io.Writer ,
concretecomponent-1 as java.io.FileWriter ,
concretecomponent-2 as java.io.ObjectWriter ,
decorator as java.io.BufferedWriter ,
concretedecorator-1 as java.io.CharCounterBufferedWriter ,
concretedecorator-2 as java.io.BlackListBufferedWriter
```

Figure B.7: Output format of Ptidej

```
Decorator
class AbstractClass,Writer,Writer.java:1:6
operation flush,Writer,Writer.java:3:15
class ConcreteClass,BufferedWriter,BufferedWriter.java:8:12
operation flush,BufferedWriter,BufferedWriter.java:9:9
```

Figure B.8: Input format of DEEBEE

```
01<microArchitectures>
02 <microArchitecture number="76">
03 <roles>
04    <components>
05      <component roleKind="AbstractClass">
06        <entity>java.io.Writer</entity>
07      </component>
08    </components>
09    <concreteComponents>
10      <concreteComponent roleKind="Class">
11        <entity>java.io.StringWriter</entity>
12      </concreteComponent>
13    </concreteComponents>
14    <decorators>
15      <decorator roleKind="AbstractClass">
16        <entity>java.io.BufferedWriter</entity>
17      </decorator>
18    </decorators>
19    <concreteDecorators>
20      <concreteDecorator roleKind="Class">
21        <entity>./CharCountBufferedWriter</entity>
22      </concreteDecorator>
23      <concreteDecorator roleKind="Class">
24        <entity>./BlackListBufferedWriter</entity>
25      </concreteDecorator>
26    </concreteDecorators>
27  </roles>
28 </microArchitecture>
29</microArchitectures>
```

Figure B.9: Format of PMART

B.2 DPDX attribute values

In order to make our DPDX format really useful, the tools (and people) using the format must agree on a standard terminology for keywords applied in DPDX. This is important because XML attributes are just strings, but we need a special meaning for some strings (e.g. the relation name cannot be an arbitrary string but a discrete value from a known relation types set). So it is essential that all users of the format use the same keyword for describing the same concept. Otherwise the primary aim of the common exchange format, namely to make the design pattern candidates comparable, would be lost. For this very reason we would suggest that all researchers interested in using (or even developing) DPDX should use our publicly available WIKI page [92] as a primary source of the existing DPDX attribute values. Each researcher is welcome to extend the available keywords if there is no suitable one available. This page prevents us from using different keywords with the same meaning; moreover the introduction of new keywords would be based on a common consensus.

As an example, we collected a reference list of possible keywords used as attribute values in DPDX, which is presented in tables B.2, B.3 and B.4.

Tag name	Attribute name	Possible attribute values
'Role'	'kind'	Class, Method, Field, Call
'Property'	'name'	abstractness, visibility staticness
'Relation'	'relationName'	subtypeOf, hasType, invokes

Table B.2: Attribute values of DPDX

Each property tag with a different name attribute has a different set of possible *kind* values. These values are listed in Table B.3.

Property name	Property attribute	Possible attribute values
abstractness	'kind'	abstract, interface, concrete
visibility	'kind'	public, protected, private
staticness	'kind'	static, non-static

Table B.3: Property values

Table B.4, which summarizes the possible values for program element *kinds* and *names*, is organized somewhat differently. The reason for this is that there are semantic dependencies between program elements.

The first column of the table shows the possible values for the 'kind' attribute of a program element (note that program elements are 'NamedElement', 'TypedElement', 'IndexedElement', 'Block' and 'ReferencingStatement'). The second column shows the possible program element names within this kind of element. The third column contains a brief description of which element is denoted by the second column.

The listed values are language independent, although the keywords used in the tables follow the Java terminology. We prefer this solution instead of creating a whole new terminology. We chose Java for the base language because of its current popularity. Despite the equivalence in terminology, our keywords have a more general meaning. They can be interpreted for various different programming languages. For example, the property value *abstract* (which is a Java keyword) can be used in DPDX to describe C++ design pattern candidates as well. We have different interpretations of the keywords for different languages; e.g. the *abstract* property of a Java class means the class is defined using the abstract Java keyword, but in the case of a C++ class, it means that the class has at least one pure virtual method (but not all of them are pure virtual as this would make the class an *interface*).

Program element *kind*	**Possible child node** *name*	**Denoted element**
class, field, method	body (implicit)	field initializer, class initializer or body of method
invocation	target	the expression denoting the object on which the method is invoked
	call	the invocation expression itself
get	target	the expression denoting the object whose field is accessed. The value "this" denotes the object for which the current method is executing
	field	the expression denoting the accessed field
set	target	the expression denoting the object whose field is accessed. The value "this" denotes the object for which the current method is executing
	field	the expression denoting the accessed field
	value	the expression denoting the new value of the field
conditional	if	condition
	then	first alternative
	else	second alternative
switch	switch	switch expression
	cases	a block of blocks, each representing a different case. Individual cases are identified by their index within this block
whileloop, dowhileloop, dountilloop	loopcondition	condition
	loopbody	loop body
forloop	forinit	
	forincrement	
	forcondition	
	forbody	

Table B.4: Program element hierarchy

B.3 DPDX implementation examples

Figures B.10, B.11, B.12 and B.13 show the implementation of DPDX.

```
01<PatternSchema id ="PS1" name="Decorator" variantOf="%NONE%">
02 <Roles>
03  <Role id="R1" name="Component" kind="Class" cardinality="1">
04   <Property name="abstractness" value="abstract" strict="false"/>
05   <Role id="R2" name="Operation" kind="Method" cardinality="+"/>
06  </Role>
07  <Role id="R3" name="Decorator" kind="Class" cardinality="1">
08   <Role id="R4" name="Forwarder" kind="Method" cardinality="+">
09    <Role id="R5" name="ForwardCall" kind="Call" cardinality="1"/>
10   </Role>
11   <Role id="R6" name="Parent" kind="Field" cardinality="1"/>
12  </Role>
13  <Role id="R7" name="ConcreteDecorator" kind="Class" cardinality="*">
14   <Property name="abstractness" value="concrete" strict="true"/>
15  </Role>
16 </Roles>
17
18 <Relations>
19  <Relation id="RE1" name="subTypeOf" source="R3" srcCard="1"
20       target="R1" targetCard="1" mandatory="true" direct="false"/>
21  <Relation id="RE2" name="subTypeOf" source="R7" srcCard="1"
22       target="R3" targetCard="1" mandatory="false" direct="false"/>
23  <Relation id="RE3" name="invokes" source="R5" srcCard="1"
24       target="R2" targetCard="1" mandatory="true" direct="true"/>
25  <Relation id="RE4" name="hasType" source="R6" srcCard="1"
26       target="R1" targetCard="1" mandatory="true" direct="false"/>
27 </Relations>
28</PatternSchema>
```

Figure B.10: Implementation of the schema metamodel

```
01<ProgramElements>
02 <NamedElement id ="PE1" name="java.io.Writer" kind="class"
03       source="P1">
04  <TypedElement id="PE2" name="write" kind="method" source="P2">
05   <ref>
06    <ref namedElement="PE13"/>
07    <ref namedElement="PE14"/>
08    <ref namedElement="PE14"/>
09   </ref>
10  </TypedElement>
11  <TypedElement id="PE3" name="flush" kind="method" source="P3"/>
12 </NamedElement>
13
14 <NamedElement id ="PE4" name="java.io.BufferedWriter" kind="class"
15       source="P4">
16  <TypedElement id="PE5" name="write" kind="method" source="P5">
17   <ref>
18    <ref namedElement="PE13"/>
19    <ref namedElement="PE14"/>
20    <ref namedElement="PE14"/>
21   </ref>
22   <IndexedElement id="PE6" indexInParent="1" kind="block">
23    <Block nameInParent="synchronized">
24     <IndexedElement id="PE7" indexInParent="3" kind="conditional">
25      <Block nameInParent="then">
26       <ReferencingStatement id="PE8" indexInParent="2"
27            kind="call" referencedElement="PE2" source="P6"/>
28      </Block>
29     </IndexedElement>
30    </Block>
31   </IndexedElement>
32  </TypedElement>
33  <TypedElement id="PE9" name="flush" kind="method" source="P7">
34   <IndexedElement id="PE10" indexInParent="1" kind="block">
35    <Block nameInParent="synchronized">
36     <ReferencingStatement id="PE11" indexInParent="2"
37          kind="call" referencedElement="PE3" source="P8"/>
38    </Block>
39   </IndexedElement>
40  </TypedElement>
41  <NamedElement id="PE12" name="out" kind="field" source="P9"/>
42 </NamedElement>
43
```

Figure B.11: Implementation of the program metamodel - first part

```
44 <NamedElement id="PE13" kind="basicType" name="char[]"/>
45 <NamedElement id="PE14" kind="basicType" name="int"/>
46
47 <Sources>
48   <Source id="P1" URI="/java/io/Writer.java" line="33" col="1"
49         endLine="308" endCol="1"/>
50   <Source id="P2" URI="/java/io/Writer.java" line="128" col="5"
51         endLine="128" endCol="81"/>
52   <Source id="P3" URI="/java/io/Writer.java" line="293" col="5"
53         endLine="293" endCol="52"/>
54   <Source id="P4" URI="/java/io/BufferedWriter.java" line="47"
55         col="1" endLine="253" endCol="1"/>
56   <Source id="P5" URI="/java/io/BufferedWriter.java" line="154"
57         col="5" endLine="183" endCol="5"/>
58   <Source id="P6" URI="/java/io/BufferedWriter.java" line="169"
59         col="3" endLine="169" endCol="28"/>
60   <Source id="P7" URI="/java/io/BufferedWriter.java" line="232"
61         col="5" endLine="237" endCol="5"/>
62   <Source id="P8" URI="/java/io/BufferedWriter.java" line="235"
63         col="6" endLine="235" endCol="17"/>
64   <Source id="P9" URI="/java/io/BufferedWriter.java" line="49"
65         col="5" endLine="49" endCol="23"/>
66 </Sources>
67</ProgramElements>
```

Figure B.12: Implementation of the program metamodel - second part

```
01<DPDResult >
02 <Tool name="NotNamed" version="1.0"/>
03 <Program name="JDK" version="1.6" language="Java"/>
04 <Diagnostic id="PI1" patternName="Decorator" patternSchema="PS1">
05   <RoleAssignments>
06     <RoleAssignment id="RA1" role="R1" player="PE1"/>
07     <RoleAssignment id="RA2" role="R2" player="PE2"/>
08     <RoleAssignment id="RA3" role="R2" player="PE3"/>
09     <RoleAssignment id="RA4" role="R3" player="PE4"/>
10     <RoleAssignment id="RA5" role="R4" player="PE5"/>
11     <RoleAssignment id="RA6" role="R4" player="PE9"/>
12     <RoleAssignment id="RA7" role="R6" player="PE12"/>
13     <RoleAssignment id="RA8" role="R5" player="PE8"/>
14     <RoleAssignment id="RA9" role="R5" player="PE11"/>
15     <RoleAssignment id="RA10" role="R7" player="%MISSING%"/>
16   </RoleAssignments>
17
18   <RelationAssignments>
19     <RelationAssignment relation="RE1" source="PE4" target="PE1"/>
20     <RelationAssignment relation="RE3" source="PE8" target="PE2"/>
21     <RelationAssignment relation="RE3" source="PE11" target="PE3"/>
22     <RelationAssignment relation="RE4" source="PE12" target="PE1"/>
23   </RelationAssignments>
24
25   <Justifications>
26     <Justification for="RA5" score="80%" explanation=""/>
27     <Justification for="RA8" score="80%" explanation="conditional
28          forward"/>
29     <Justification for="RA10" score="" explanation="missing subclass"/>
30     <Justification for="PI1" score="95%" explanation=""/>
31   </Justifications>
32 </Diagnostic>
33</DPDResult>
```

Figure B.13: Implementation of the result metamodel

Appendix C

Summary

C.1 Summary in English

The main contributions of this work are summarized as follows. First, we employ machine learning methods to further refine the results of our design pattern miner tool. We perform experiments to measure the performance (speed and memory consumption) of design pattern miner tools. Furthermore, we develop DEEBEE, a benchmark for evaluating and comparing design pattern miner tools, and perform some experiments by using it. We also introduce an XML-based output format (DPDX) for design pattern miner tools. Last but not least, we develop BEFRIEND, a benchmark that can be used for evaluating and comparing reverse engineering tools.

A method for improving design pattern mining

The motivation of this work is to remove false pattern candidates given by the basic matching algorithm of our design pattern miner tool [13] by applying machine learning methods. Our approach is to analyze the candidates returned by the basic matching algorithm, taking into consideration several aspects of the candidate code fragment and its neighbourhood, such as whether a candidate class has a parent or not. The information corresponding to these aspects is referred to as *predictors*, whose values can be used in a machine learning system. We employ a conventional learning approach; that is, we first manually tag the candidates as true or false. Afterwards, the values of the predictors on the candidates are calculated. Then we load these into two learning systems, namely a decision tree-based and a neural network system. These in turn provide models that incorporate the acquired knowledge. We test the models with the cross-validation method. We performed our experiments on *StarWriter* [80] as the subject system for pattern mining and we searched for the *Adapter Object* and the *Strategy* design patterns. In our experiments we achieved learning accuracy scores of 67–95%

and with the model we were able to filter out 51 of the 59 false candidates of the Adapter Object design pattern (out of a total of 84 candidates) and 33 of the 35 false candidates of the Strategy pattern (out of a total of 42 candidates).

Evaluating the speed and memory consumption of design pattern miner tools

The aim of this study is to compare three design pattern miner tools, namely Columbus, Maisa and CrocoPat. The study is based on the same input, hence ensuring a fair comparison, because any parsing errors that occured affected all three tools alike. The tools are compared in three aspects: differences between the candidates, speed and memory consumptions. We do not analyze whether a design pattern candidate is true or false; we examine these tools only from the viewpoint of structural candidates and differences. We experimentally test our preliminary assumptions about the causes of the differences between the results of the tools. During the experiments we identified the following common causes of the differences: different definitions of design patterns, precision of pattern descriptions and differences in algorithms.

Based on the experiments, it is clear which tool should be used in certain circumstances in terms of speed or memory consumption. Columbus is the fastest in the case of complex patterns or in the case of small-or medium-sized systems. CrocoPat is the fastest in the case of simpler patterns or in the case of large sized systems, while Maisa can used if just a small amount of memory is available.

Validating design pattern miner tools with DEEBEE

The motivation for this thesis derives from the above two studies. During these two studies we faced the problem of evaluating and comparing the results of design pattern miner tools. Different result formats have to be processed and the appropriate source code fragments have to be located and the results of a comparison need to be recorded. We performed these tasks manually in the case of the above two studies, which were quite dull and time-consuming. We also observed that there was a desire in conferences and publications for a means of evaluating patterns easily and effectively [63]. Hence, we developed a publicly available benchmark, 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. Our benchmark is also able to relate the same but differently reported pattern candidates (siblings), thus it eases the evaluation process and improves the correctness of evaluation and comparison results.

With the help of our benchmark the accuracy of two design pattern miner tools (Columbus and Maisa) are evaluated on reference implementations of design patterns and on

two software systems, NotePad++ and FormulaManager. Design pattern instances used in NotePad++ were also discovered by hand, so both precision and recall scores are calculated by DEEBEE. We developed FormulaManager to test the tools on a program where each design pattern is implemented in a real-life context.

DPDX: a common format for design pattern miner tools

The above studies and other studies [54] revealed many limitations of the current output formats of the design pattern miner tools. Some tools do not report either their own identity or the name and version of the program that they analyzed. Some output format do not contain all roles relevant to a given motif or do not identify reported roles unambiguously. Some tools do not identify detected motif candidates unambiguously or do not report their conceptual schema of the identified motif. Other tools do not justify their results or use ad hoc (generally textual) output formats.

In this thesis we present DPDX, a common exchange format for design pattern detection tools. The proposed format is based on a well-defined and extendible metamodel that addresses the limitations described above. The extendible metamodel consists of three parts, namely the schema metamodel, program element metamodel and result metamodel. The proposed metamodel is implemented in an XML-based language (DPDX) that can be easily adapted by existing and future tools, providing a means for improving accuracy and recall scores when evaluating, comparing and combining their findings.

Validation of reverse engineering tools with BEFRIEND

This thesis introduces **BEFRIEND** (BEnchmark For Reverse engInEering tools workiNg on source coDe), the further development of DEEBEE. The results of reverse engineering tools recognizing the arbitrary characteristics of source code can be subjectively evaluated and compared with BEFRIEND. Such tools include design pattern miners, duplicated code detectors and coding rule violation checkers. BEFRIEND differs from DEEBEE in five aspects. First, it allows the uploading and evaluating of results related to different domains (like duplicated code detectors and design pattern miners). Second, it allows the editing of the evaluating aspects of the results, while DEEBEE has fixed evaluation aspects. Third, it improves and extends the user interface. Fourth, it generalizes sibling relationships to tackle the problems of other domains, not just design pattern mining, and last but not least, it allows the uploading of files in different formats by introducing a plug-in oriented architecture.

We applied BEFRIEND to three reverse engineering domains, namely design pattern mining tools, duplicated code detector tools, and coding rule violation checking tools. The evaluation results stored in DEEBEE were migrated to BEFRIEND, and in the duplicated code domain we applied the benchmark using additional examples, namely five *duplicated code* finder tools were assessed on two different open source projects.

C.2. Summary in Hungarian

Jelen munka fő hozzájárulásai a következőképpen foglalhatóak össze. Először gépi tanuló algoritmusokat alkalmaztunk a tervezési minta kereső eszközünk eredményeinek pontosításához. Majd kísérleteket végeztünk el tervezési minta kereső eszközök performanciájának (sebesség és memóriaigény) mérésére vonatkozóan. Továbbá kifejlesztettünk egy benchmarkot (DEEBEE), tervezési minta kereső eszközök kiértékelésére és összehasonlítására, valamint ezt felhasználva végeztünk néhány kísérletet. A tervezési minta kereső eszközök számára bevezettünk egy XML alapú formátumot (DPDX) is. Végül, de nem utolsó sorban kifejlesztettük a BEFRIEND benchmarkot, amely visszatervező eszközök kiértékelésére és összehasonlítására használható fel.

Egy módszer tervezési minta keresés javítására

Ezt a munkát a tervezési minta keresőnk alap algoritmusa által megadott hamis mintajelöltek eltávolítása motiválta, melyhez gépi tanuló algoritmusokat alkalmaztunk. Az eljárásunk lényege az alap algoritmus által visszaadott jelöltek analizálása, figyelembe véve a jelölt kódrészletnek és környezetének számos tulajdonságát, például azt, hogy egy jelölt osztálynak van-e szülője vagy sem. Ezen információkat hívjuk prediktoroknak, melyek értékeit a gépi tanuló rendszer használja fel. Szokványos tanulási megközelítést alkalmaztunk: először kézzel igaznak vagy hamisnak osztályoztuk a jelölteket, ezután pedig a jelöltekhez tartozó prediktor értékeket határoztuk meg. A soron következő lépésben ezeket betöltöttük két tanulórendszerbe, ahol az egyik egy döntési fa, míg a másik egy neurális háló volt. A tanuló algoritmusok modelleket adtak a megszerzett tudásról, összefüggésekről. A modelleket keresztvalidáció módszerrel teszteltük. Kísérleteinket a *StarWriter* [80] rendszeren végeztük el, ahol az *Adapter Object* és a *Strategy* mintákat kerestük. A kísérletek során 67–95% tanulási pontosságot értünk el, és az előállt modellekkel képesek voltunk kiszűrni 51 hamis Adapter Object jelöltet az 59 hamis jelölt közül (összesen 84 jelölt volt), valamint 33 hamis jelöltet a 35 hamis Strategy jelölt közül (összesen 42 jelölt volt).

Tervezési minta kereső eszközök sebességének és memóriaigényének kiértékelése

Ennek a tanulmányak az volt a célja, hogy három tervezési minta kereső eszközt összehasonlítsunk, a Columbust, a Maisat és a CrocoPat-et. A tanulmány közös bemeneten alapszik, így biztosítva az összehasonlítás korrektségét, mivel bármilyen elemzési hiba egyformán befolyásolta mindhárom eszközt. Az eszközök három szempont szerint kerültek összehasonlításra: a jelöltek közötti különbségek, sebesség és memóriaigény. Azt nem vizsgáltuk, hogy egy tervezési minta jelölt helyes vagy sem; az eszközöket csak a jelöltek szerkezeti eltéréseinek fényében hasonlítottuk össze. Kísérletek során teszteltük

előzetes feltevéseinket az eszközök eredményei közötti különbségek okairól. A kísérletek során a különbségek következő gyakori okait azonosítottuk: tervezési minták eltérő definíciója, minta leírások pontossága és az algoritmusok különbözősége.

A kísérletekre alapozva egyértelművé vált, hogy mely eszköz milyen körülmények mellett alkalmazható, figyelembe véve a sebességet vagy memóriaigényt. Komplex minták vagy kis és közepes rendszerek esetén a Columbus volt a leggyorsabb. A CrocoPat egyszerűbb minták vagy a nagy méretű rendszerek esetén volt a leggyorsabb, míg a Maisat akkor célszerű használni ha csak kevés memória áll rendelkezésre.

Tervezési minta kereső eszközök validációja DEEBEE-vel

Ezt a tézispontot a két előző tanulmány motiválta. Az előző két tanulmány során szembesültünk a tervezési minta kereső eszközök eredményeinek kiértékelése és összehasonlítása során felmerülő problémákkal. Különböző eredmény formátumokat kellett feldolgozni, meg kellett keresni a megfelelő forráskódrészletet és az összehasonlítás eredményét el kellett tárolni. Ezeket a feladatokat a két korábbi tanulmány során kézzel végeztük el, ami igencsak időigényes volt. Konferenciákon és publikációkban [63] is megfigyeltünk egy olyan eszköz iránti igényt, amely lehetővé teszi a minták egyszerű és hatékony kiértékelését. Ezek miatt kifejlesztettünk egy publikusan elérhető benchmarkot, a DEEBEE-t (DEsign pattern Evaluation BEnchmark Environment), amely tervezési minták kiértékelését és összehasonlítását támogatja. A benchmark általános: nyelv-, szoftver-, eszköz- és mintafüggetlen. A benchmarkkal az eszközök pontossága(precision) és teljessége(recall) bárki által validálhatóvá válik. A benchmark arra is képes, hogy ugyanazon de eltérő módon közölt mintajelölteket (testvérek) összekapcsoljon, így egyszerűsítve a kiértékelési folyamatot és javítva a kiértékelési és összehasonlítási eredmények helyességét.

A benchmark segítségével két tervezési minta kereső eszköz (Columbus és Maisa) pontosságát értékeltük ki, a tervezési minták referencia implementációján és két szoftverrendszeren, a NotePad++-on és a FormulaManageren. A NotePad++ rendszerből további tervezési minta példányokat is felderítettünk kézzel, így mind a pontosság (precision) és teljesség (recall) statisztikákat biztosítja a DEEBEE. A FormulaManagert is mi fejlesztettük ki azért, hogy az eszközöket egy olyan programon is teszteljük, ahol minden egyes minta valós környezetben van implementálva.

DPDX: tervezési minta kereső eszközök közös formátuma

A fenti tanulmányok és egyéb munkák [54] rámutattak a tervezési minta kereső eszközök jelenlegi kimeneti formátumainak számos hiányosságára és korlátaira. Néhány eszköz nem közli saját identitását vagy az elemzett program nevét és verzióját. Más kimeneti formátumok nem tartalmazzák egy minta lényeges szereplőit vagy nem egyértelműen azonosítják a szereplőket. Bizonyos eszközök nem egyértelműen azonosítják a jelölteket

vagy az azonosított mintához tartozó sémájukat nem adják meg. Néhány eszköz nem indokolja az eredményeit vagy ad-hoc formátumot (pl. szöveges) használ.

Ebben a tézispontban bemutattuk a DPDX-et, a tervezési minta kereső eszközök közös csereformátumát. A javasolt formátum egy jól definiált és bővíthető metamodellen alapszik amely a fent említett hiányosságokat, határokat célozza meg. A bővíthető metamodell három részből áll, a séma metamodellből, a programelem metamodellből és az eredmény metamodellből. A metamodell egy XML alapú nyelven van implementálva, ami így könnyen adaptálható a létező és a jövőben kifejlesztésre kerülő eszközök által, ezzel biztosítva egy eszközt amely növeli az eredmények kiértékelésének, összehasonlításának és kombinálásának pontosságát.

Visszatervező eszközök validációja BEFRIEND-el

Ez a tézispont bevezeti a BEFRIEND-et (BEnchmark For Reverse engInEering tools worKiNg on source coDe), amely a DEEBEE továbbfejlesztett változata. A BEFRIEND-el a forráskód tetszőleges tulajdonságait felismerő visszatervező eszközök eredményei értékelhetőek ki és hasonlíthatóak össze egymással. Ilyen eszközök a tervezési minta keresők, duplikált kód azonosítók és a szabálysértés ellenőrzők. A BEFRIEND öt szempontban jelentősen eltér az elődjétől (a DEEBEE-től). Először, lehetővé teszi különböző területekhez (pl. duplikált kód detektorokhoz és tervezési minta keresőkhöz) kapcsolódó eredmények feltöltését és kiértékelését. Másodszor, biztosítja az eredmények kiértékelési szempontjainak szerkesztését, míg a DEEBEE-nek rögzített kiértékelési szempontjai vannak. Harmadszor, javítja és bővíti a felhasználói felületet. Negyedszer, általánosítja a csoportosító mechanizmust (testvér kapcsolatok) az egyéb területek (pl. duplikált kód keresők) problémáinak kezelése miatt, és végül de nem utolsó sorban, lehetővé teszi különböző formátumok feltöltését egy plug-in architektúra bevezetésével.

A BEFRIEND-et három visszatervező területen alkalmaztuk, tervezési minta keresők, kód másolat keresők és kódolási szabálysértés azonosító eszközök esetén. A DEEBEE-ben tárolt kiértékelési eredményeket pedig migráltuk a BEFRIEND-be, és a kód másolat területen ki is próbáltuk a benchmarkot: öt duplikált kód kereső eszközt értékeltünk ki két különböző nyílt forráskódú rendszeren.

Bibliography

[1] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In *16th International Conference on Automated Software Engineering (ASE'01)*, pages 166–173. IEEE Computer Society, November 2001.

[2] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Design patterns: A round-trip. In *Proceedings of 11th ECOOP Workshop for PhD students in Object-Oriented Systems*, June 2001.

[3] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: application to pattern detection and code synthesis. In *Proceedings of First ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[4] 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.

[5] 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.

[6] Angel Asencio, Sam Cardman, David Harris, and Ellen Laderman. Relating Expectations to Automatically Recovered Design Patterns. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 87–96. IEEE Computer Society, 2002.

[7] 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.

[8] 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.

[9] The Bauhaus Homepage.
http://www.bauhaus-stuttgart.de.

[10] 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.

[11] Bellon benchmark. http://www.bauhaus-stuttgart.de/clones/.

[12] Árpád Beszédes, Rudolf Ferenc, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. CSiBE Benchmark: One Year Perspective and Plans. In *Proceedings of the 2004 GCC Developers' Summit*, pages 7–15, June 2004.

[13] 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.

[14] Dirk Beyer and Claus Lewerentz. CrocoPat: Efficient pattern analysis in object-oriented programs. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003)*, pages 294–295. IEEE Computer Society, 2003.

[15] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient Relational Calculation for Software Analysis. In *Transactions on Software Engineering (TSE'05)*, pages 137–149. IEEE Computer Society, February 2005.

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

[17] 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.

[18] Marcelo Campo, Claudia Marcos, and Alvaro Ortigosa. Framework comprehension and design patterns: A reverse engineering approach. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, June 1997.

[19] The CCFinder Homepage.
http://www.ccfinder.net/.

[20] Checkstyle homepage . http://checkstyle.sourceforge.net/.

[21] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In *Journal: IEEE Software*, Volume 7, pages 13–17, January 1990.

[22] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design Pattern Recovery by Visual Language Parsing. In *Proceedings of the 9th Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 102–111. IEEE Computer Society, March 2005.

[23] CSIBE Homepage.
`http://www.csibe.org`.

[24] DC++ Project.
`http://sourceforge.net/projects/dcplusplus`.

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

[26] Design Pattern Benchmark Homepage.
`http://www.inf.u-szeged.hu/designpatterns/`.

[27] The Design Pattern Detection tool Homepage.
`http://java.uom.gr/~nikos/pattern-detection.html`.

[28] Jing Dong, Dushyant S. Lad, and Yajing Zhao. DP-Miner: Design Pattern Discovery Using Matrix. In *ECBS'07*, pages 371–380, Washington, USA, 2007. IEEE Computer Society.

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

[30] 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.

[31] DPDX Homepage.
`https://sewiki.iai.uni-bonn.de/dpdx/`.

[32] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, October 2002.

[33] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 58–70. University of Szeged, June 2001.

[34] 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.

[35] FindBugs homepage . http://findbugs.sourceforge.net/.

[36] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Pub Co, 1999.

[37] The FUJABA Homepage. http://www.cs.uni-paderborn.de/cs/fujaba/.

[38] FxCop homepage .
http://msdn.microsoft.com/en-us/library/bb429476.aspx.

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

[40] 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 .

[41] Yann-Gaël Guéhéneuc. A reverse engineering tool for precise class diagrams. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research (CASCON'04)*, pages 28–41. IBM Press, 2004.

[42] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design patterns identification. In *Proceedings of IJCAI Workshop on Modelling and Solving Problems with Constraints*, pages 57–64, August 2001.

[43] 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.

[44] 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.

[45] The Imagix Homepage.
http://www.imagix.com.

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

[47] IBM Jikes Project.
http://jikes.sourceforge.net/.

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

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

[50] Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Efficient Identification of Design Patterns with Bit-vector Algorithm. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 175–184. IEEE Computer Society, 2006.

[51] Holger Kampffmeyer and Steffen Zschaler. Finding the Pattern You Need: The Design Pattern Intent Ontology. In *MoDELS*, Volume 4735 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2007.

[52] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-Based Reverse-Engineering of Design Components. In *The 21st International Conference on Software Engineering (ICSE'99)*, pages 226–235. IEEE Computer Society, 1999.

[53] 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.

[54] Günter Kniesel and Alexander Binun. Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection. Technical report IAI-TR-2009-01, ISSN 0944-8535, CS Department III, Uni.Bonn, Germany, January 2009.

[55] The Licor Homepage.
`http://prog.vub.ac.be/research/DMP/soul/soul2.html`.

[56] Alan K. Mackworth. The logic of constraint satisfaction. *Artif. Intell.*, 58(1-3):3–20, 1992.

[57] Maisa Homepage.
`http://www.cs.helsinki.fi/group/maisa/`.

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

[59] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research (CASCON '93)*, 1993.

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

[61] P-MARt Homepage.
`www.ptidej.net/downloads/pmart/`.

[62] 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.

[63] 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.

[64] PMD homepage . `http://pmd.sourceforge.net/`.

[65] The Ptidej Homepage. `http://www.ptidej.net`.

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

[67] 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.

[68] Filip Van Rysselberghe and Serge Demeyer. Evaluating Clone Detection Techniques. In *Proceedings of the International Workshop on Evolution of Large Scale Industrial Software Applications, 2003.*, 2003.

[69] 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.

[70] Nija Shi and Ronald A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 123–134, Washington, USA, 2006. IEEE Computer Society.

[71] Forrest Shull, Walcelio L. Melo, and Victor R. Basili. An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems. Technical report, 1996.

[72] Michael Siff and Thomas W. Reps. Identifying modules via concept analysis. In *Proceedings of the 13th IEEE International Conference on Software Maintance (ICSM'97)*, pages 170–179. IEEE Computer Society, October 1997.

[73] Susan Elliot Sim. *A Theory of Benchmarking with Applications to Software Reverse Engineering*. PhD thesis, University of Toronto, 2003.

[74] Susan Elliot Sim, Steve Easterbrook, and Richard C. Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *Proceedings of the Twenty-fifth International Conference on Software Engineering (ICSE'03)*, pages 74–83. IEEE Computer Society, May 2003.

[75] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On Using a Benchmark to Evaluate C++ Extractors. In *Proceedings of the Tenth International Workshop on Program Comprehension (IWPC'02)*, pages 114–123. IEEE Computer Society, Jun 2002.

[76] Susan Elliott Sim and Margaret-Anne D. Storey. A Structured Demonstration of Program Comprehension Tools. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 184–193. IEEE Computer Society, Nov 2000.

[77] Susan Elliott Sim, Margaret-Anne D. Storey, and Andreas Winter. A Structured Demonstration of Five Program Comprehension Tools: Lessons Learnt. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 210–212. IEEE Computer Society, Nov 2000.

[78] The Simian Homepage.
`http://www.redhillconsulting.com.au/products/simian/`.

[79] Jason Smith and David Stotts. SPQR: Flexible Automated Design Pattern Extraction From Source Code. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'03)*. IEEE Computer Society, 2003.

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

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

[82] Paolo Tonella and Giuliano Antoniol. Object Oriented Design Pattern Inference. In *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, pages 230–238. IEEE Computer Society, 1999.

[83] The Trac Homepage.
`http://trac.edgewall.org/`.

[84] 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.

[85] Gabriella Tóth, Péter Hegedűs, Judit Jász, Árpád Beszédes, and Tibor Gyimóthy. Comparison of Different Impact Analysis Methods and Programmer's Opinion - an Empirical Study. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010)*, pages 109–118, September 2010.

[86] Stefan Wagner. A literature survey of the quality economics of defect-detection techniques. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 194–203. ACM, 2006.

[87] Stefan Wagner, Florian Deissenboeck, Michael Aichner, Johann Wimmer, and Markus Schwalb. An Evaluation of Two Bug Pattern Tools for Java. In *Proceedings of the 1st IEEE International Conference on Software Testing, Verification and Validation (ICST 2008)*, pages 1–8. ACM, 2008.

[88] 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.

[89] 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.

[90] 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.

[91] Lothar Wendehals. *Struktur- und Verhaltensbasierte Entwurfsmustererkennung*. PhD thesis, Universität Paderborn, Institut für Informatik, September 2007.

[92] WIKI page of DPDX.
`https://sewiki.iai.uni-bonn.de/research/dpd/`.

[93] WinMerge Project.
`http://sourceforge.net/projects/winmerge`.

Corresponding publications of thesis topics

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] 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.