Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Dottorato di Ricerca in Informatica — XXIV Ciclo

Data mining techniques for design pattern detection

SUPERVISOR: Prof.ssa Francesca Arcelli Fontana

TUTOR: Prof. Fabio Stella

PhD Candidate: Zanoni Marco

Academic Year 2011-2012

To my wife and my son.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The detection of design patterns [74] is a topic which is gaining importance in the context of reverse engineering [136, 45], and in particular for software architecture reconstruction [66].

A relevant objective of reverse engineering is to obtain representations of the system at a higher level of abstraction and to identify the fundamental components of the analyzed system by retrieving its constituent structures. Getting this information should greatly simplify the restructuring and maintenance activities, as we obtain more understandable views of the system and the system can be seen as a set of coordinated components, rather than as a unique monolithic block.

Design pattern detection can be useful for this purpose. In fact, the main objective of design pattern detection is to gain better comprehension of a software system, and of the kind of problems addressed during the development of the system itself. Design patterns support these goals because the definition of a pattern brings also the *intent* of the developer who applies it. Moreover, the collection of design patterns applied in a system create a *dictionary* of the design solutions, simplifying the communications among developers and maintainers. The presence of design patterns can be considered an indicator of good software design [2], as design patterns are reusable for their self definition. For these motivations they are very important during the re-documentation process, in particular when the documentation is very poor, incomplete or not up-to-date.

Since the introduction of design patterns the research community started to try to detect them automatically. Many different approaches were developed, with different results, but no approach reached a real visibility. Many causes concur to this situation. One is that the empirical validation of the detection of design patterns is a long task, where expert designers must check detected instances to validate and enhance an approach or a tool. Another one is that different experts can assign different evaluations to the same pattern instance. This difference in the evaluation of design patterns derives from the fact that patterns are defined as structures to apply to solve a particular design issue, but their *exact* structure is not enforced. The definitions given by Gamma et al.[74] give examples of UML class and sequence diagrams, examples of source code, and a natural language definition of when to apply the pattern or not, how to apply it and what consequences are to be expected from its application. Many different variants are defined for each pattern definition, and many others can be created as needed.

In this kind of situation it would be desirable to have an agreed reference formal definition of what is a pattern and what is not. While that reference is not available, some solutions can be tried. One of these solutions is to have at least a shared and agreed dataset of pattern instances as a test for design pattern detection tools. Some work has been done in this direction, and it is getting relevance in the research community. An available work is from Fülöp et al. [73, 72] and consists of a web platform for the benchmarking of reverse engineering tools, and another one is

DPDX [119], an exchange format developed by Kniesel et al. for design pattern detection tool results. Another work in the reverse engineering benchmark field, called DPB [15], is developed by our research group and is a web platform for the benchmarking of design pattern detection tools. The development of DPB was part of my research during the PhD, but it is not addressed in this thesis.

Another way of having a better-targeted detection would be to make it subjective to the particular user or community. In fact, if it is possible to consistently evaluate many pattern instances, it could be possible to teach a tool what is the wanted definition of a certain pattern. This thesis proposes a solution to this kind of setup.

The solution proposed in this thesis exploits machine learning technologies [91, 161] to support an iterative learn-by-example process in a design pattern detection tool, called MARPLE-DPD. MARPLE [11] (Metrics and Architecture Reconstruction PLugin for Eclipse) is an Eclipse plugin for software architecture reconstruction able to perform design pattern detection, developed in our Software Evolution and Reverse Engineering Lab [14]. MARPLE-DPD shows a list of pattern candidates to the user, who can evaluate them and train a classification algorithm. The trained algorithm will be able to evaluate new pattern instances, returning a confidence value telling how much each instance conforms to the learned model. The train-evaluation steps are iterative, supporting an incremental analysis of the system, and keeping the evaluation effort as low as possible.

Part of the solution is based on a particular modeling of design patterns, based on *micro-structures*, which are a particular kind of basic information, or facts, about classes or other recognizable pieces of code. Micro-structures have the peculiarity to be formally defined and mechanically-recognizable, so they represent a reliable source of information for the abstraction of a software system. Micro-structures are combined with the roles of design patterns to define the models representing pattern instances, which are seen as tree-structured groupings of role-class mappings.

The two main subsystems of MARPLE implementing the approach described in the thesis are the *Joiner* and the *Classifier* modules. The Joiner module is able to extract design pattern candidates from a software system. It represents the system as a graph, where the nodes are the classes of the system and the edges are the micro-structure detected in the system. Then it applies a graph matching query on the graph for each design pattern, and retrieves structured groups of classes representing the pattern candidates. Each class is assigned to a role for the respective design pattern.

The machine learning part of the detection process is implemented by the Classifier module, which takes the candidates extracted by the Joiner as input. The pattern candidates can be labeled as correct or incorrect by the user, and the Classifier module uses this information to train clustering and classification algorithms. The trained algorithms are then exploited to classify new candidates coming from the Joiner, without the need of another training session. The user can decide when to add other labeled instances to the ones already present and re-run the training of the algorithms.

In this thesis the detection process is carefully explained and some experiments are reported to test the effectiveness of the process on five design patterns: *Singleton*, *Adapter*, *Composite*, *Decorator* and *Factory Method*. The thesis is organized as follows.

In Chapter 2 the literature related to design pattern detection is reviewed, with particular attention to the implemented tools; in particular, Section 2.4 discusses the approaches exploiting machine learning or other kind of soft computing techniques.

In Chapter 3 the modeling aspect of reverse engineering is taken into consideration. A selection of the existing models for reverse engineering and design pattern detection are reviewed, discussing the differences among them and the ones integrated in MARPLE.

In Chapter 4 the MARPLE tool is introduced, describing its architecture and the Eclipse technologies exploited for its different modules. The *Information Detector* and *Software Architecture Reconstruction* modules are described, providing also examples of the reconstructed architectural views. Finally, a prototypal distributed implementation of MARPLE is described.

In Chapter 5 the micro-structures defined for both reverse engineering and design pattern detection are discussed. The concept of micro-structure is defined, and examples are given for each of the kind of micro-structures supported in MARPLE: elemental design patterns, micro patterns, design pattern clues.

In Chapter 6 the first design pattern detection module, called *Joiner*, is described. The chapter explains the matching phase of the Joiner process, where role-class mappings are extracted from source code, and the merge phase were the extracted role-class mappings are grouped to build the final tree-structured pattern candidates. The merge phase is supported by a model for the representation of design patterns, explained in Subsection 6.2.1. Finally, the detection rules for the five patterns evaluated with the Classifier module in this thesis are reported and explained. In Appendix A the definitions of the Joiner rules, for all the patterns of the Gamma et al.'s book that are not used for the experiments, are reported.

In Chapter 7 the approach of the Classifier module is explained. The modeling issues related to the representation of design patterns as feature vectors are addressed, and the strategy to solve these issues is explained and justified; some enhancements to the basic solution are then explained. Finally, the graphical interface and user interaction with the MARPLE-DPD are described.

In Chapter 8 the report of the performed experiments is available. The tested machine learning techniques, the detected patterns and the analyzed projects are described. The experimental setup description is completed by the discussion of the approach employed to perform the parameter optimization for the learning algorithms. Then the experiments results are reported, showing and discussing the performance values obtained on the detection of five different patterns. Finally, the limitations of the approach and the threats to validity are discussed. In Appendix B the parameters of the machine learning algorithms setups whose performance are reported in Chapter 8 are listed.

In Chapter 9 conclusions are drawn for the entire approach, and future work is outlined; future work is about the extension of the experimentation to other patterns and algorithms, and the enhancement of the support software and libraries exploited to perform the experiments.

The list of the papers published in the context of this work can be found at the end of the thesis, in chapter *Publications*.

Chapter 2

Techniques and Tools for DPD

This chapter will introduce some related work about design pattern detection. Every approach defines its different detection strategy and in some cases provides also a tool implementing the strategy itself.

2.1 Classification of design pattern detection approaches

Since the introduction of design patterns by the book from Gamma et al. [74] many design pattern detection approaches were proposed and experimented. They can be classified using the following properties:

Analysis type The majority of the approaches available in the literature is based on static analysis and is typically focused on the evaluation of the structural aspects of source code (e.g. generalization, association, attributes and methods). Other approaches consider useful to verify results coming from static analysis using dynamic analysis, i.e. recording the behaviour of the system (method invocations) at execution time. A little number of solutions relies on certain semantic information (i.e. naming conventions).

Recognition type The recognition of a design pattern instance can be exact or approximate. In the first case the detection tool will consider only the results satisfying all the constraints defined by a detection rule. Some techniques belonging to this category are: graph matching (or isomorphism recognition) [128, 160, 115], pattern matching [3], regular expressions [3, 25], SQL [1] queries, SPARQL [198] queries over RDF [159], constraint solvers [110], first order logic [30]. In the second case the tool defines a minimum threshold on the number of constraints an instance must satisfy to be reported as a result, or is able to give a confidence or probability value to the matching. Some techniques belonging to this category are: machine learning [69], fuzzy logic [138], genetic algorithms [87].

Input type Every tool executes its analysis starting from the information extracted from an input system. The system must be coded in a specific programming language and can be in binary or textual format.

Intermediate representation It is often useful to transform the information extracted from the analyzed system in a more abstract format, to simplify the analysis of the system and the query of the information itself. Among the most common representation types, used by design pattern detection tools, are: Abstract Syntax Trees, Abstract Semantic Graphs, Unified Modeling Language, or other ad-hoc meta-models.

2.2 Design pattern detection tools

This section will introduce some of the best known design pattern detection tools available in the literature. The descriptions are organized following the classification introduced at the beginning of the chapter, exploiting the *Analysis type* and the *Recognition type* criteria. In particular, tools are first subdivided by these categories: tools based on static analysis and tools based on dynamic analysis; each category has a further categorization for tools providing exact or approximated recognition. For each tool, if an official homepage exists, it is specified after the tool's description.

2.2.1 Static analysis with exact recognition

Pat

Pat is a tool based on the approach from Kramer et al. [122]. The tool proposes a methodology based on the structural analysis of the code, targeted to the creation of a Prolog knowledge base. The knowledge base, integrated with a set of rules derived from the design pattern definitions, can be queried to provide the set of detected patterns. The approach is limited to structural patterns.

DP++

DP++, developed by Bansiya [28], proposes an approach to design pattern detection based on the recognition of structural relationships among the components of the analyzed system. The design pattern detection process description is not detailed, so other comments are not possible.

SPOOL

SPOOL a design pattern detection tool built following the approach defined by Keller et al. [114] in the context of the SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) project. The tool proposes a procedure split in two phases: an automatic one and an interactive one. In the first phase some generic pattern structures are identified. Next they are graphically visualized (using UML/CDIF) to the user, who will refine the selection, deleting wrong results and completing the correct or partial ones.

Hedgehog

Hedgehog is a tool defined following the approach from Blewitt et al. [39]. The approach defines a new first order logic language, called SPINE, which allows to describe some structural and behavioural (called *semantic* in the paper) aspects of a design pattern in synthetic syntax similar to Prolog. The structure of a design patter is decomposed into *mini-patterns* and *idioms*. Pattern descriptions can be interpreted by a dedicated system, able to test if a class satisfies the pattern requirements or not.

JBOORET

JBOORET (Jade Bird Object-Oriented Reverse Engineering Tool), developed by Hong et al. [101], analyzes C++ source code, filling a database using the extracted information. That information is then used to reconstruct high level views of the analyzed code. The user can explore the obtained models to visually look for design pattern instances.

Ptidej

Inspired by the approach from Albin-Amiot et al. [4], and furtherly developed by Guéhéneuc et al.[134], Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java) proposes a solution based on the modeling of design patterns as entities in the PADL model, easy to handle and analyze. Models can be interpreted as a set of constraints a piece of code must satisfy to be considered a pattern instance.

Homepage `http://www.ptidej.net`

SPQR

SPQR (System for Pattern Query and Recognition) was conceived basing on the approach developed by Smith and Stotts [170], who propose a detection based on ρ-calculus and on the usage of the OTTER theorem prover[1]. The data necessary to the automatic elaboration by OTTER are retrieved from source code, analyzing the abstract syntax trees exported by the gcc compiler. The abstract syntax trees are analyzed and the retrieved facts are saved in an exchange file, and then transformed to the OTTER input format. The facts about the code are called EDPs (Elemental Design Patterns). OTTER than evaluates the ρ-calculus rules against the system model and determines if pattern instances are present. SPQR is able to detect structural patterns.

CroCoPat

CroCoPat, realized during the development of the research from Beyer et al. [34], is a query tool for relational data. It takes RSF (Rigi [191] Standard Format) files as input, which is a popular exchange and persistence format among reverse engineering tools. From the RSF it derives binary decision diagrams (BDDs). The BDDs, combined to computation techniques based on logic predicates, provide a formal analysis of the system, allowing the detection of patterns using logic queries.

Homepage `http://www.sosy-lab.org/~dbeyer/CrocoPat/`

DPRE

DPRE (Design Pattern Recovery Environment) is a visual environment based on the approach, composed of two phases, developed by Costagliola et al. [51]. In the first phase a UML class diagram is extracted from the source code. The diagram, encoded in SVG, is submitted to a second module having the goal to inspect the diagram, which applies graph matching techniques to detect design patterns in the system.

PINOT

PINOT (Pattern INference recOvery Tool) is a tool based on the approach defined by Shi et al. [167], who argue against the techniques available in the former literature, and propose a re-classification of design patterns, paired with a different per-category detection process. The approach exploits the structural and behavioural characteristics of each group of patterns and promises to provide a more accurate and fast detection. The tool is developed as a modification of the jikes [103] open source Java compiler.

[1] `http://www.cs.unm.edu/~mccune/otter/`

Homepage `http://www.cs.ucdavis.edu/~shini/research/pinot/`

DP-Miner

DP-Miner was developed by Dong et al. [60], who introduce a solution composed of three phases. The first one is the structural analysis of source code: data retrieved from the class diagram are compared to a set of metrics related to the definitions of design patterns. The results of the first phase are refined by a second behavioural analysis, and by a third syntactic analysis, where some naming conventions are verified.

Homepage `http://www.utdallas.edu/~jdong/DesignPattern/DP_Miner/index.htm`

DPJF

DPJF (Detection of Patterns by Joining Forces) is a tool developed by Binun and Kniesel [37]. The authors define a methodology based on the optimized usage of different existing analysis strategies. The presented solution derives from the accurate study of the weaknesses of previous approaches. The reached goals are 100% precision, excellent recall values and execution time, which makes the approach eligible for real-time detection of patterns with the direct support of the user. The tool is also able to raise a warning to the user if a detected pattern is incomplete, and to trigger the correct refactoring to reach a complete implementation.

Homepage `http://sewiki.iai.uni-bonn.de/research/dpd/dpjf/start`

Web Of Patterns

One of the distinguishing feature of Web Of Patterns (WOP), developed by Dietrich and Elgar [57, 59], is the attention paid to the specification of the information associated to the recognized instances. The information is conforming to an OWL ontology, published on the project web page[58]. The technique adopted by the tool is the verification of formulas described using first-order logic.

Homepage `http://www-ist.massey.ac.nz/wop/`

Rational Software Architect

Rational Software Architect (RSA) is a commercial suite developed by IBM. The employed techniques for the detection are not publicly available, but it is possible to conjecture that the tool exploits static analyses, because there is no request for an execution setup. The tool does not associate confidence values to the detection, so it is possible to think that it employs exact matching techniques.

Homepage `http://www.ibm.com/developerworks/rational/products/rsa/`

D-CUBED

A tool developed by Stencel et al. [177]. The design pattern detection rules are specified using first order logic. They are then transformed into SQL queries, to be applied on a database containing a model of the analyzed application.

Homepage `www.yonlabs.com/dcubed.html`

2.2.2 Static analysis and approximated recognition

Columbus

Columbus is a design pattern detection tool for C++ based on the approach from Ferenc et al. [69]. The tool combines machine learning techniques and "classic" approaches based on graph matching. The detection process consists of a first transformation of the code into an Abstract Semantic Graph (ASG), and then matched against the definition of the supported design pattern; the instances are represented using other kinds of information, representing non-structural information, and then submitted to a machine learning system.

DPD-tool

Developed by Tsantalis et al. [195], who choose to represent the analyzed system as an inheritance tree and a set of $n \times n$ matrices (where n is the number of classes), containing structural information (e.g. associations, generalizations, method invocations). The detection starts with an exploration of the inheritance tree and can be expanded to the analysis of the matrices. A score is assigned to each analyzed class, which determines the similarity with a role of a design pattern.

Homepage `http://java.uom.gr/~nikos/pattern-detection.html`

2.2.3 Dynamic analysis with exact recognition

Design Pattern Verification toolKit

DPVK (Design Pattern Verification toolKit) is a tool developed by Wang and Tzerpos [199]. The employed approach is composed of a first static code inspection, where a matching is performed against structural pattern definitions, followed by a dynamic analysis, which allows to refine the results using the information about the behaviour of the classes in the pattern instances.

mb-pde

mb-pde, developed by Marcel Birkner [38], supports the recognition of a great number of design pattern, providing a good flexibility in the definition of recognition rules. The detection process is composed of a first static analysis, where the information extracted from the source code is compared with the UML class diagram of the wanted design pattern. In the next phase, the tool compares the execution trace of an instrumented version of the system with a UML sequence diagram of the same pattern. Then the acquired information is compared to provide the matching degree between the system and the diagrams.

Homepage `http://code.google.com/p/mb-pde/`

ePAD

ePAD is an Eclipse plugin, developed by De Lucia et al. [55], which detects structural and behavioural design patterns. The detection process is composed of a first static analysis phase, which extracts an UML class diagram from the source code and applies pattern matching against a text representation of the diagram. The found instances are then filtered using dynamic analysis over an instrumented version of the system.

Homepage `http://www.sesa.dmi.unisa.it/ePAD/`

2.2.4 Dynamic analysis with approximated recognition

Fujaba-Reclipse

Developed following the approach from Niere et. al. [139], and refined later by Wendehals [200]. The tool performs the translation of the system source code in an abstract syntax graph (ASG). The graph is matched against a set of detection rules. The detection process is helped by the user, to reduce false positives and avoid useless computations. In particular, the user has to set the fuzzy thresholds for the different sub-patterns present in the rule, influencing the detection. Each matching is fuzzy itself, bringing its fuzzy value.

Homepage `http://www.fujaba.de/no_cache/projects/reengineering/reclipse.html`

2.2.5 Details about the reported tools

This section contains three tables, reporting relevant information about the tool described in Section 2.2. The tables are provided as a synthesis of the available information about the tools, and to facilitate comparisons.

Some information contained in the tables is taken from a survey written by Dong et al. [61].

Table 2.1 compares the tools regarding the design patterns they can recognize.

Table 2.2 compares the tools reporting some generic characteristics, i.e. supported languages, export format, documentation availability and last update.

Table 2.3 reports the systems analyzed by the tools' authors during their development and test. The reported data can be used also as an indirect estimation of the reliability and scalability of the tools.

2.3 Theoretical design pattern detection approaches

Other design pattern detection approaches exist in the literature, other than the ones reported in Section 2.2, but they have never been implemented as a tool known to be available freely, commercially or under specific request. This section collects a significant set of these approaches, following the same classification of Section 2.2: static or dynamic analysis combined with exact or approximated recognition.

2.3.1 Static analysis with exact recognition

Seemann and von Gudenberg (1998)

Seemann and von Gudenberg [164] introduce an approach based on the analysis and refining of a graph representing the structure of the analyzed system's codebase, exploiting first order logic. Results are reported on the detection of structural patterns.

Antoniol et al. (1998)

Antoniol et al. [5] define an approach that transforms the source code in an intermediate model, based on the formalization of existing concepts in object-oriented languages. Design patterns are detected matching their definition, formalized using a set of metrics, with the same metrics extracted from the source code model.

Table 2.1: Design patterns supported by the reported tools

Tool	Abstract Factory	Adapter/Command	Builder	Bridge	Chain of Responsibility	Composite	Decorator	Facade	Factory method	Flyweight	Mediator	Observer	Prototype	Proxy	Singleton	Strategy/State	Template Method	Visitor
Columbus		×														×		
CroCoPat	— information not available —																	
D-CUBED	×		×						×						×			×
DP++	— information not available —																	
DPJF					×	×	×					×		×				
DPD-tool		×				×	×		×			×	×		×	×	×	
DP-Miner		×		×		×										×		
DPRE		×		×		×	×							×				
DPVK	— information not available —																	
ePAD		×		×		×	×	×				×		×		×	×	×
Reclipse				×		×										×		
Hedgehog		×		×					×					×	×	×		
JBOORET	— user-driven recognition —																	
MARPLE	×					×												×
mb-pde	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×
Pat		×		×		×	×							×				
PINOT	×	×		×	×	×	×	×	×	×	×	×		×	×	×	×	×
Ptidej	— user-defined recognition algorithms —																	
RSA							×		×			×			×			×
SPOOL		×		×					×					×	×			
SPQR	— user-defined recognition algorithms —																	
WOP	×	×		×		×								×	×		×	

Table 2.2: Main characteristics of cited tools

Tool	Supported languages	Export	Documentation	Last update
Columbus	C++	n/a	n/a	n/a
CroCoPat	Java	RSF	Yes	02/2008
D-CUBED	Java	n/a	n/a	n/a
DP++	C++	n/a	n/a	n/a
DPJF	Java	Prolog	Yes	11/2011
DPD-tool	Java	XML	Yes	05/2010
DP-Miner	Java	XML	Yes	01/2010
DPRE	C++ e Java	n/a	n/a	n/a
DPVK	Eiffel	n/a	n/a	n/a
ePAD	Java	not supported	Yes	10/2010
Reclipse	Java	not supported	Yes	10/2011
Hedgehog	Java	n/a	n/a	n/a
JBOORET	C++	n/a	n/a	n/a
MARPLE	Java	XML	n/a	12/2011
mb-pde	Java	XML	Yes	03/2008
Pat	C++	n/a	n/a	n/a
PINOT	Java	unstructured text	Yes	10/2004
Ptidej	Java	semi-structured text	Yes	04/2006
RSA	Java	XML	Yes	11/2011
SPOOL	C++	n/a	n/a	n/a
SPQR	C++	XML	n/a	n/a
WOP	Java	XML	Yes	1.4.3

Table 2.3: Experimentations made on the cited tools

Tool	Java AWT	Java Swing	Java IO	Java JDK	JHotDraw	JRefactory	JUnit	JEdit	ArgoUML	Eclipse	Batik	JasperReports	StarWriter	TrackerLib	Galib	Libg++	Mec	Others
Columbus													×					
CroCoPat	×			×	×													×
D-CUBED						— information not available —												
DP++						— information not available —												
DPJF	×		×		×			×	×									
DPD-tool					×	×	×											
DP-Miner	×				×		×											×
DPRE															×	×	×	×
DPVK						— information not available —												
ePAD					×													
Reclipse	×																	
Hedgehog	×																	
JBOORET						— information not available —												
MARPLE											×	×						
mb-pde						— information not available —												
Pat						— information not available —												
PINOT	×	×			×													×
Ptidej	×				×		×	×										
RSA						— information not available —												
SPOOL																		×
SPQR														×				×
WOP						— information not available —												

Asencio et al. (2002)

Asencio et al. [24] represent patterns as classes with links to other patterns, software qualities, design concepts, and applicability conditions. Each pattern is associated with one or more recognizers. They employ a set-theoretic, structural specification language for describing recognition assets. Using this language, recognizer authors construct specifications of a pattern's template. Each recognizer is not intended to be a perfect solution, but should be easily changeable and extensible by interested users.

Espinoza et al. (2002)

Espinoza et al. [64] conjecture that design patterns can be described using a model defining the structural relationships existing among their components. The relationships can be: inheritance, aggregation, knowledge or creation (as defined by the original design pattern book [74]). Once described the source code using the same model, the detection is performed matching the definition against the source code model. The approach allows the detection of structural patterns.

Zhang et al. (2004)

Zhang et al. [204] propose to represent an analyzed system using extended graphs, based on these relationships: association, composition and inheritance. The detection of patterns is performed as a graph matching task, looking for isomorphisms, complete or between sub-graphs.

Streitferdt et al. (2005)

Streitferdt et al. [179] introduce a solution based on a new paradigm for the description of design patterns, where the structure of each pattern must be characterized by a set of elements that can be necessary, forbidden (negative) or "don't-care" (to ignore). That solution would allow a better detection of pattern variants.

Kaczor et al. (2006)

Kaczor et al. [111] formulate the design pattern detection as a combinatorial problem, proposing a bit-vector based solution. The proposed algorithm allows an efficient detection, computationally independent from the size of the analyzed program.

Bayley et al. (2007)

Bayley et al. [30] propose an approach based on the description of the detection rules in first-order logic. The rules are applied to a model of the system defined using the GEBNF meta-model, that can be represented using sets and first-order logic. GEBNF is a modified EBNF version that allows the description of any graphic model. The model represents the UML class diagram of the systems. Rules are matched using a theorem prover.

Gupta et al. (2010)

Gupta et al. [88] propose an approach based on graph matching over UML class diagrams. In particular, the approach has a first phase where nodes and edges are labeled in terms of their depth and nature, and a second search phase, where the DNIT search algorithm is applied.

Rasool et al. (2010)

Rasool et al. [158] propose an innovative approach based on the introduction of annotations in the analyzed system's source code. The usage of annotations would allow a simple and direct knowledge transfer among the different people involved in the system design and maintenance. The choice of the specification language to adopt into the annotations is up to the user.

2.3.2 Static analysis with approximated recognition

Guéhéneuc et al. (2004/2010)

Guéhéneuc et al. [84, 86] exploit machine learning to solve the design pattern detection problem. They propose to have a first step of information gathering, where a human agent retrieves and classifies a significant sample of micro-architectures, representing pattern instances. The second step is automated: a tool analyzes the code, matching metrics calculated on the input sample.

Guéhéneuc and Antoniol (2008)

Guéhéneuc and Antoniol [85] introduce a three-tier approach for the identification and traceability of micro-architectures in the source code. The approach uses meta-models, which can be applied to model the source code and the wanted design patterns. A support tool will transform the description of each design pattern in a set of constraints, to be used for the detection of the patterns into the model representing the system.

2.3.3 Dynamic analysis with exact recognition

Heuzeroth et al. (2003)

Heuzeroth et al. [96] propose an approach where a first analysis of the source code is made using the Recoder tool, which is able to extract an abstract syntax tree (AST) from the source code. A static analysis is performed on the AST, which retrieves a list of pattern candidates. The list is refined using dynamic analysis. The approach is able to recognize structural and behavioural patterns.

Hayashi et al. (2008)

Hayashi et al. [95] describe an integrated approach based on static and dynamic analysis. The design patterns supported by the detection process are described as Pree's meta-patterns [153]. This choice brings a reduction in the elaboration time, by simplifying the detection process. The detection of pattern instances is made using search rules written as Prolog predicates.

2.4 Machine learning and design pattern detection

The focus of this thesis is the exploitation of data mining techniques for design pattern detection, so the relevant approaches applying some kind of machine learning or soft computing techniques need to be discussed in more detail.

Ferenc et al. [69], in their Columbus tool, define an approach similar to the one explained in this thesis. The similarities is in the fact that they use machine learning algorithms to filter false positives out of the results of a graph matching phase, trying to provide better precision to the overall output. The main differences with their approach are three: the modeling approach, the employed techniques and the usage of the confidence values of the classification algorithm.

The modeling approach taken in consideration by the authors is to represent each pattern instance as a labeled dataset row, having has features the values of *predictors*, defined for the specific pattern in consideration. Predictors are measures over a pattern instance, reporting the value of a metric or particular property of the entire instance or one of its parts. Predictors are similar in definition and scope to the *micro-structures* (see Chapter 5) used in this thesis as features, but they are defined over an instance and not over classes. Micro-structures are collected indiscriminately on each class of the system and used as features to represent the property of one specific class or pair of classes. Moreover, the value of the micro-structures, when used as features, is boolean.

The techniques employed in the cited paper are C4.5 decision tree and neural networks with back propagation. In the experiments reported in this thesis the focus is more on Support Vector Machines, but also some other common algorithms (including C4.5) have been experimented. In particular, the proposed implementation is "algorithm-agnostic" in the sense that every possible learning algorithm available (or adaptable to) the Weka [91] framework can be employed. No neural network was experimented.

Finally, the last difference is that the MARPLE tool reports the confidence value (if available) of the classifier to the user, enabling a ranking of the reported pattern instances; the confidence value enriches the knowledge about the reported pattern, simplifying the user's choice of which pattern to inspect first.

Another approach based on machine learning is the one from Guéhéneuc et al. [84, 86]. The authors use standard metrics as features for the machine learning process. The main differences are two: machine learning is not employed as a filter and the modeling phase is different.

The approach is based on a single detection step, where for each class of the systems a set of metrics are recorded, and used in the learning and validation by a rule learner. The rule learner, therefore, is not used to filter out false positives produced by an initial matching performed by some other kind of detector. The training set is produced by the user labeling a set of classes as belonging to a role of a design pattern (or not), without tool help.

The dataset construction and modeling is very different from the one described in this thesis: rules for each design pattern role are learned independently, and the dataset is filled with a proportion of true instances vs. false instances of 1 to 3, respectively. As opposed, my approach uses the combination of all the roles belonging to a design pattern to enhance to learning phase.

The main commonality between the two approaches is that the Weka environment is used (not surprisingly, as it is the most common machine learning framework for Java), even if the usage of the JRip learner by the authors could be easily extended to any other classifier supported by Weka. In fact, the only advantage of using JRip is that it produces readable rules that can be interpreted by the user. The Guéhéneuc et al. assert that the learned rules are the "fingerprints" of the design patterns, and build their paper on that metaphor. One could argue that, using the same learning process, and exploiting other classifiers, the internal model of any trained classifier can be considered as the "fingerprint" of the pattern, even if it is not directly comprehensible by the user. After all, also human fingerprints are not distinguishable by people, by eye.

Tsantalis et al. [195] use similarity scoring among matrices to allow partial matching between the definition of a pattern and the reported instances. The main difference is that the similarity algorithm is not a machine learning technique. This is a great advantage from the user's effort point of view, because the algorithm does not need to be trained and is stable from user to user; the only setup task is to set the threshold for the score, which is discussed in the paper. The most significant similarity with the approach presented in this thesis is that the similarity scoring could be reported to the user, giving the idea of the amount of matching.

Similarly, the approach implemented in Fujaba by Niere et. al. [139] is to give to each subpattern matching a fuzzy value, that will influence the scoring of the entire pattern. In the

same fashion, the user can give different fuzzy weights to the different subpatterns to influence the overall matching. The authors wish to automatically set the fuzzy weights using machine learning techniques in future work, which I am not aware of.

The approach of the DeMIMA tool, from Guéhéneuc and Antoniol [85] is a kind of hybrid one. It does not use machine learning, but it exploits a constraint solver, called JPalm [110], to perform an exact detection. JPalm performs an explanation-based constraint solving, which means that the results are paired with the list of the matched (or not) constraints. On top of the constraint solver, an interactive procedure asks the user (if he wants) to remove one or more constraints, therefore relaxing the constraint and including more results, which are imperfect design pattern instances. The amount of relaxation is used by the tool to give a score to the pattern instance. The only learning process in the approach is the interactive relaxation made by the user, who can decide to stop the process. The use of a scoring that uses a fixed scheme and is not inferred from examples can be discussed with the same considerations applied to the two previously discussed approaches.

2.5 Conclusion

Researchers tried to detect design pattern instances in source code since their introduction in 1995. The applied techniques span many areas, like graph theory, constraint satisfaction, fuzzy sets, machine learning, computation of similarity measures. The amount of available literature suggests an optimal solution has not yet been found. Many causes concur to this situation: design pattern definitions, programming languages, the environment that hosts the detection process.

The definitions of design patterns are traditionally given in a semi-formal or informal way, mixing UML diagrams, textual descriptions and code examples. The focus of the definition of the pattern is often on the pattern's *intent*, and not on the solution. This raises the known problem of variants, but also a more subtle problem of agreement around what can be considered a pattern or not. Some authors prefer to use the term *design motif* [85] detection, instead of design pattern detection, because motifs are the concrete realizations of the patterns we want to detect. The point is good, but my personal opinion is that the task does not change by changing its name: the goal is to find subsystems where a particular design pattern was applied. A more useful option would be to start formalizing better the definitions of patterns from the point of view only of the *design*, considering also the mid-level of abstraction of the patterns. In fact, a lot of the debate on the correctness of a pattern instance is in the intent and behaviour, which are not really related to the design. A correctly implemented pattern instance, but implemented for the wrong reason, is still a correct one. From the assessment perspective it would be more useful to have better design information, not necessarily related to particularly high-level intent information. In this context, the specification of *new* pattern definitions could be the way to go.

Design pattern detection is influenced also by the particular constructs of programming languages: every new native construct of a language could implicitly implement (and eliminate the need for) a particular design pattern. In this sense, design patterns could be seen as suggestions for what next-generation languages should allow to *directly* implement.

The last variable for the pattern detection is the kind of usage and interaction expected: some tools tend to implement a traditional assessment task, in a batch fashion, while other ones model the task as an interactive and/or contextual one. Moreover, some approaches want the user to collaborate with the detection algorithm, while others, in a more traditional way, think that the process should be totally automated.

Chapter 3

Model driven reverse engineering

Reverse engineering tasks are often supported by models. Models are the way techniques or tools are able to abstract from the input representation of the systems, i.e. source code or binaries. The abstraction of the concepts related to the features of the programming languages allows reverse engineering tools to be more independent from the single language, focusing on the key concept related to the programming paradigm and the structural information relevant to the reverse engineering task. Moreover, *abstracting* means also representing the input by using less information, and improving the speed of any analysis. Many models have been proposed or implemented in the reverse engineering research community, and also MARPLE defines its own model for the representation of the system and of design patterns. The remainder of this chapter contains a review of a selection of models and meta-models in the reverse engineering area; in particular the available models for the representation of design patterns will be discussed. The related work will be useful to understand the commonalities and differences among the existing models and the ones used in MARPLE.

3.1 Reverse engineering models

3.1.1 Knowledge Discovery Metamodel

KDM (Knowledge Discovery Metamodel) [145] is a meta-model defined by the Architecture-Driven Modernization (ADM) Task Force [137, 143] of the Object Management Group (OMG) [146], defined using MOF [141]. The specification is composed of a set of meta-models that, combined, are able to provide a complete and accurate representation of an existing software system, independent from its programming language and from the technologies exploited for its implementation.

The collection of meta-models composing KDM can be seen as a holistic and extensible representation schema, with the aim of describing the key aspect of the knowledge associated to the different aspect of an enterprise software system.

The goal of KDM is to define a shared and complete representation, able to guarantee the interoperability of different tools, and to efficiently support maintenance, evolution, assessment and modernization activities.

The widescale adoption of KDM would allow to abandon source code as the only agreed knowledge reference of a system, and to adopt a more abstract representation as a knowledge base, which would be independent from the employed technologies and the operational context.

A representation with these characteristics would permit the authors of analysis tools to focus on the development of useful techniques for their goals, and to be able to demand to third-party, standard-compliant, tools the task of the extraction of the model from the source

code. A full separation of the program parsing and analysis would be achieved.

KDM is designed to simplify the analysis process based on incremental activities, where an initial representation, produced by an automatic code inspection, is gradually enriched by design details, coming from other tools or from expert users working on the extracted model.

Structure of KDM

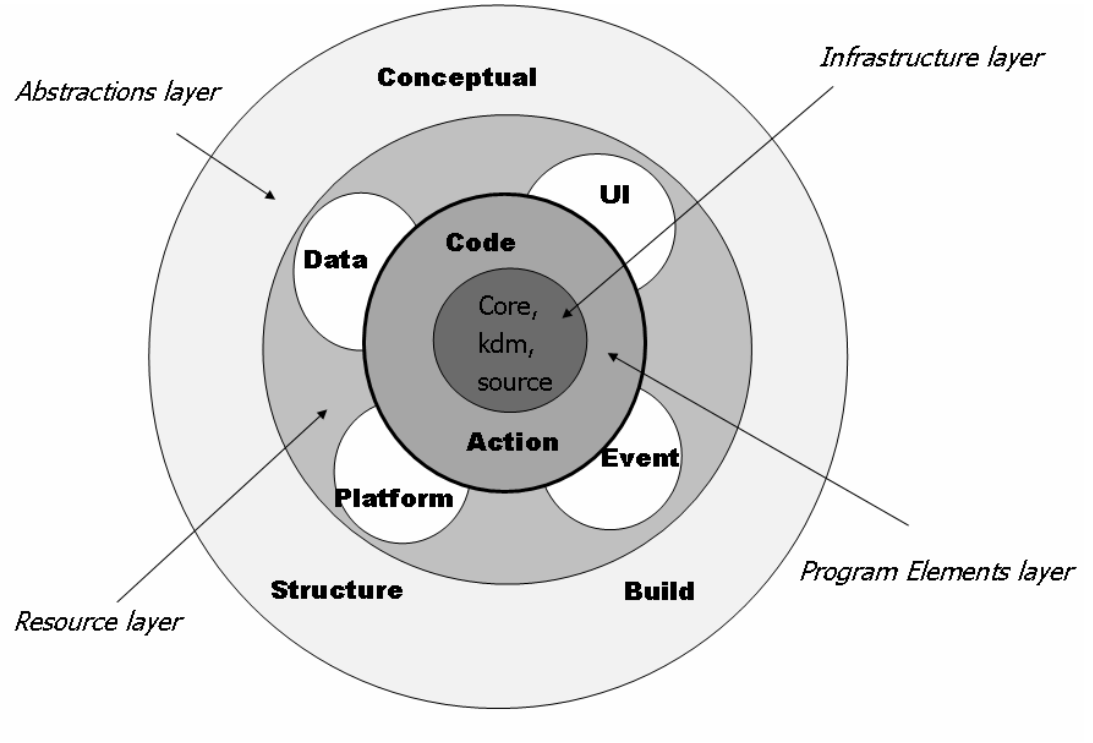


Figure 3.1: KDM: graphic representation of the meta-model

KDM is designed as a meta-model composed of nine meta-models, or *packages*, subdivided in four layers (see Figure 3.1). Each layer provides a new higher abstraction level.

The implementation of a layer requires the adoption of all the underlying layers, and every layer, excluding the first, is optional and must be used with respect to the context and the analysis goals.

In the following there is a short description of the four layers cited above, and their respective packages.

- Infrastructure layer: Gathers the fundamental concepts in KDM.
  - Core: Defines a set of element types, the constraints belonging to these elements, and the relationships it is possible to define among them. Each relationship is binary, and represents a semantic association between the entities it links.
  - Kdm: Defines a set of elements representing KDM's representation framework.
  - Source: Defines the entities used to elicitate the traceability references among KDM elements and the artifacts representing the analyzed system's source code.
- Program elements layer: Contains the definition of entities necessary for the description of the fundamental constructs shared by the majority of programming languages.

  - Code: Defines the necessary entities for the description of the elements, and their structural relationships, contained in an analyzed source code.
  - Action: Defines useful concepts for the description of the behavioural characteristics of a system. In particular, using this package, it is possible to describe control and data flows between pairs of elements extending the Code package.

- Runtime resource layer: Represents the operating runtime environment of a software system.
  - Platform: Defines the basic dictionary to use to describe the elements building the operating environment of the considered system (e.g. OS, middleware) and the logic flows among them at runtime.
  - UI: Defines the entities that can be used to describe concepts related to the system's user interface.
  - Event: Represents the knowledge associated to the events and state transitions that are observable in the runtime phase.
  - Data: Allows describing the artifacts used for data persistence, e.g. RDBMS, structured files.

- Abstractions layer: Represents different types of domain and application abstraction.
  - Conceptual: Defines the practical tools needed to describe the business rules and the domain constraints.
  - Structure: Allows the description of the logical structure of a system in terms of logical sub-components or modules, layers and sub-systems.
  - Build: Represents the characteristics of the resources generated during the build process.

Diffusion

To the best of my knowledge, at the moment of writing this thesis, the usage of KDM is limited to a project, now hosted by the Eclipse Foundation [186], called MoDisco [62, 43]. The low interest demonstrated by the research community towards KDM, could be caused to the complexity of the detailed specification, combined with its level of maturity or to the low visibility of the project in the scientific literature.

Documentation

The KDM meta-model is widely documented in the official specification documents [144] and in some documentation pages on the KDM Analytics organization website [112].

Final considerations

KDM is with no doubt the most complete meta-model currently available in the literature, and the one with the more detailed specification.

The possibility to represent a software system using different levels of abstraction, catching only some aspects and discarding others, allows the developer to move in different directions, following his analysis needs, and to rely on other complementary tools if it is possible to reuse functionalities which have already been developed by others.

The meta-model provides great expressiveness and is perfectly extensible.

3.1.2 FAMIX

FAMIX [56] is a meta-model, defined using the FM3 [123, 172] meta-meta-model, which is a simplification of EMOF [141]. It is designed to represent the static properties of a software system, and to be independent from the programming languages used for the implementation of the system. The meta-model supports procedural and object-oriented languages.

The main goal of this meta-model is the definition of a schema, sufficiently complete to satisfy the needs of reverse engineering tools and refactoring tasks.

It considers a limited number of entities and the resulting schema, even if it is less complete than the one provided by the KDM specifications, has many interesting characteristics that make it adaptable, easily extensible and simply exploitable in a practical context.

Among the distinguishing characteristics of the schema are:

- the support for multiple inheritance;
- the support of statically and dynamically typed languages;
- the possibility to represent exceptions and annotations.

Structure of FAMIX

At the moment of writing, these FAMIX specifications are available:

- FAMIX 2.2 [189];
- FAMIX 3.0 (beta) [190].

The first one is very tied to the object-oriented paradigm and aims to isolate shared characteristics of different programming languages, grouping them in a bottom-up fashion.

The second one, instead, allows abstracting more from the constructs adopted by the single supported languages, and aims to a meta-model that can be sufficiently generic for the comprehension of a larger and heterogeneous set of systems.

In order to have more detailed information about FAMIX 3.0, it is possible to refer to the class diagram represented in the documentation pages on the official web site [190].

Diffusion

FAMIX is adopted, and was born for, the Moose [140, 174] open source project. In fact, the project, started in 1996, has the meta-model's authors among his authors.

The Moose platform allows the translation of the source code of a software system in a FAMIX model. The model, enriched also with metrics information, can be then employed in the analysis of the system. Among the available functionalities of the tool are: modeling, metrics computation, software visualization, duplicated code detection.

The FAMIX meta-model, and the whole Moose framework, are implemented using Smalltalk, to simplify the exploration of the data contained in the model. In fact, one of the benefits of this technological choice is the ability to navigate the contents of the instantiated models using a native query language inherited by the Smalltalk language.

Every FAMIX model can be serialized to file using the MSE [173] format, which is a textual format able to represent any model whose meta-model was defined using FM3.

Documentation

The FAMIX meta-model is partially documented in the original author's PhD thesis [188] and on the Moose project web site [77]

The data in the first document are not up to date and refer to the first version of the meta-model, while the information available on the web site is fragmented and not much formalized. The most up to date information is the source code of the FAMIX implementation in Moose.

Final considerations

FAMIX shares many characteristics with the Program elements layer of KDM. Each of them is used to describe the static characteristics of a system, at the level of the constructs existing in the source code.

KDM offers a more detailed and clear reference documentation, while FAMIX is more limited in the number of defined entities and in the documentation quality. Despite the lack of documentation, FAMIX has many interesting characteristics and a hierarchical structure that is simple to understand. The probable motivation of this difference is that FAMIX was built relying on the experience gained during the development and testing of the Moose platform, in research and industrial projects, while KDM, which comes from an industry-oriented consortium, is designed as a holistic standard proposal. Finally, if FAMIX had a more organic documentation, it could be considered as a reduced version of the KDM's Program elements layer, because they share the same kinds of abstraction.

3.1.3 Dagstuhl Middle Model

Dagstuhl, or Dagstuhl Middle Model (DMM), is a meta-model, compatible with the GXL format [97], developed in the context of an international research project started in 2001 within the Dagstuhl Seminar on Interoperability of Re-engineering Tools [124].

The main feature of the meta-model is to base its definition on the distinction, in the representation of a software system, of the distinct and well-separated macro areas: references to source code (ModelObject), code elements (SourceObject) and relationships among code elements (Relationship).

The overall goal of DMM is to define a meta-model able to abstract from low-level details (i.e. the particular features or constructs of specific programming languages), and at the same time to ignore features usually modeled by other high-level meta-models, e.g. components, pipes, filters.

DMM was not born as a complete and omni-comprehensive model, but is proposed instead as an agile tool to be practically applied in reverse engineering projects.

The authors of DMM did the choice of ignoring some kind of entity:

- references to method or procedure calls;
- assignments and control statements;
- local variables.

The consequences of these choices can be summarized by these assertions:

- DMM does not model enough details to reconstruct the original source code after the population of the model;
- DMM does not model enough details to generate a data flow diagram.

Structure of DMM

As already introduced, DMM poses the focus on the distinction among SourceObject, ModelObject and Relationship (as shown in Figure 3.2) elements. The complete specification of the model contains a rich inheritance of elements, each one extending one of these three elements.

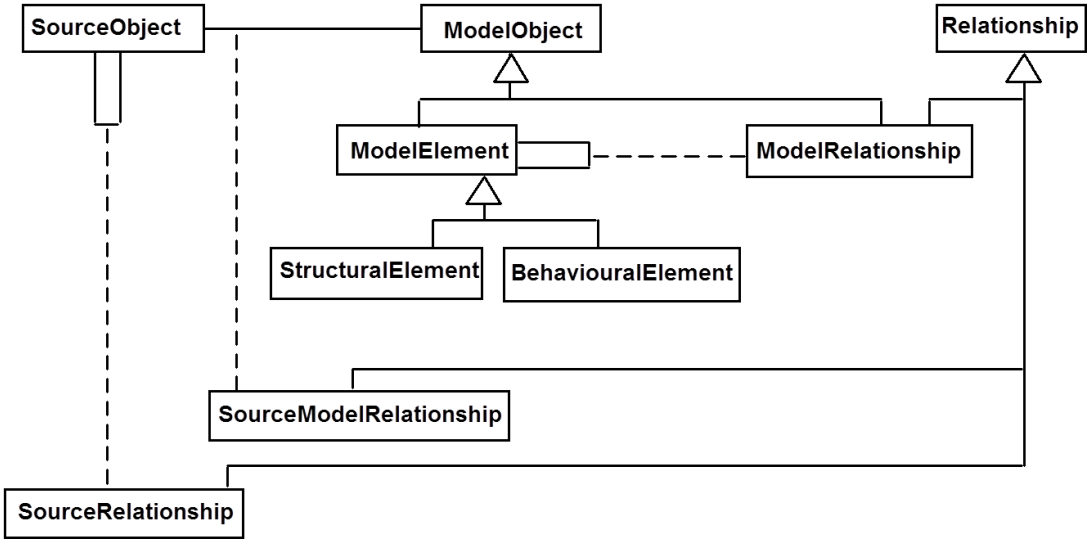


Figure 3.2: DMM: graphical representation of the meta-model

The first can be interpreted as a simple link to source code. By using the entities extending it, it is possible to contextualize a ModelObject representation.

ModelObject can represent any type of programming language construct, in the procedural or object-oriented paradigm. The representable entities can be, for example, methods, routines, classes, variables and other constructs that are significant for a structural or behavioural static analysis.

Relationship, finally, represents relationships that can exist among elements of type SourceObject↔SourceObject, ModelObject↔ModelObject and SourceObject↔ModelObject.

The elements described in this subsection can be extended and enriched is needed. The interoperability of the model is guaranteed by the presence of the common superclasses defining the basic dictionary, which is common to all the tools supporting the meta-model.

Each model can be translated in every format supporting the concepts of class, instance and association among instances, e.g. GXL [102, 99], XMI [142], RDF [159].

Diffusion

The Dagsthul Middle Model was experimented in the context of a project for metrics calculation from Java code [132].

Documentation

The meta-model is documented in the PhD thesis of the author [126] and in a document published by Lethbridge et al. [125]. The project has no official web site and it is possible to conjecture that was abandoned.

Final considerations

The Dagsthul Middle Model is an interesting simplified variant of other more complete meta-models, like KDM (limiting to the Program elements layer) and FAMIX. Its strength is in the extreme simplification of concepts and their clear distinction.

With respect to FAMIX it is possible to notice:

- the absence of an explicit management of exceptions: no entities were defined to represent exceptions and the entities with type Method have no reference to other classes in case of exception;
- the presence of entities able to explicitly manage aggregation classes like Collection or Enumeration;
- the presence of a wider and more logically organized hierarchy of entity relationships;
- the presence of a number of additional attributes in the Method entity, e.g. isConstructor, isDestructor, isDinamicallyBound, isOverridable;
- the absence of an entity representing packages or namespaces.

Unfortunately, the absence of an exhaustive documentation and of a reference implementation drastically reduces the value of the meta-model. The extensibility of the model is guaranteed by the compliance with the GXL specification.

3.1.4 Pattern and Abstract-level Description Language

Pattern and Abstract-level Description Language (PADL) [155] is a meta-model developed for the Ptidej tool from Gueheneuc et al. [85].

Structure of PADL

Three different levels of abstraction are used to model programs, which are called *abstract-level models* (see Figure 3.3):

- ICodeLevelModel represents the information about a program that can be directly extracted from the program representation;
- IIdiomLevelModel represents a model of a program where some idioms have been reified, typically binary class relationships computed using one of PADL Analyses, and some extra data have been added, for example the length of the methods. From the documentation it appears that an effort was made to classify all the different relationships present in UML, i.e. generalisation, implementation, specialisation, creation, association, aggregation, composition.
- IDesignLevelModel represents a model of a program where design data is available. Typical design data is extracted from the idiom-level model, for example occurrences of design motifs.

A fourth model exists to describe design motifs, used by the Ptidej Solver: IDesignMotif.

Diffusion

The meta-model is employed in Ptidej and its side projects.

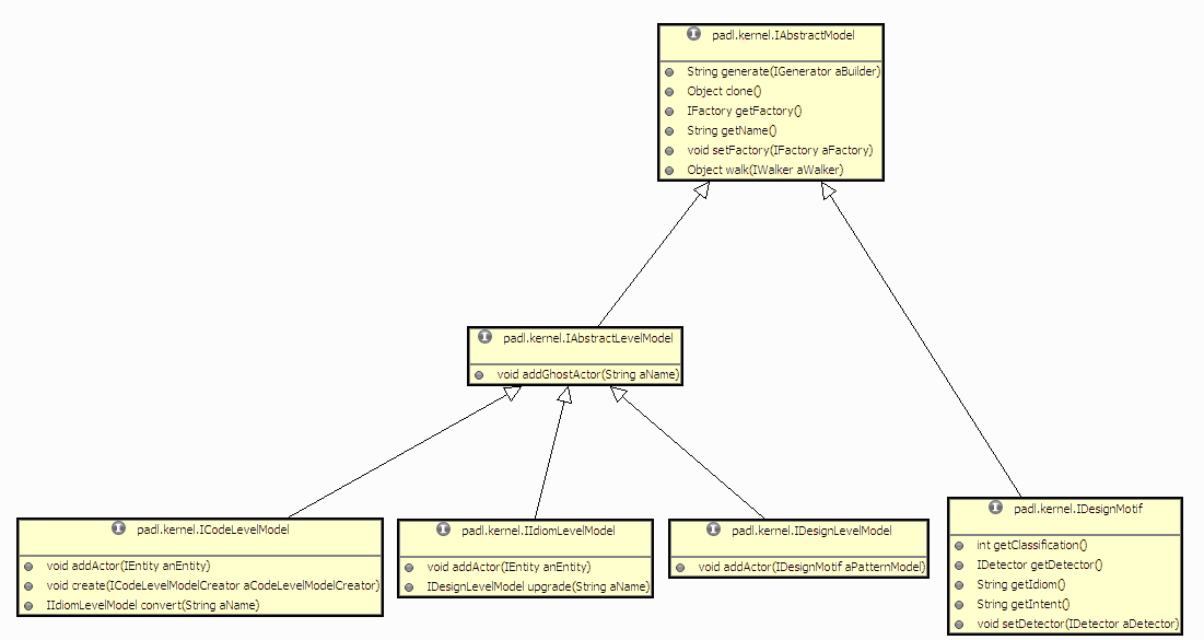


Figure 3.3: PADL: graphical representation of the meta-model

Documentation

Documentation is available through the Ptidej Wiki [155], and its usage is explained in several papers regarding the Ptidej [82], DECOR [134, 133], and DeMIMA [85] projects. A dedicated documentation or specification for the metamodel is not available. The most reliable source of information for the model is the source ode implementing it.

Final considerations

The PADL meta-model appears to be a model built bottom-up for the needs of the Ptidej tool. It is aligned with FAMIX, for example, in terms of the abstraction provided. It is not possible to formulate other meaningful considerations, as the documentation is not sufficient, and a review of the source code was not performed.

3.1.5 MARPLE meta-model

MARPLE uses an internal meta-model for the representation of the systems it analyzes, developed specifically for the tool needs. The model focuses on the practical needs of a tool that supports different activities, e.g. software architecture reconstruction (SAR) [66], design pattern detection and anti-pattern [36, 42, 201] detection [134, 133].

The meta-model entities are subdivided in different categories, representing logical or physical entities of the system, and other higher level information kinds, which support deeper analyses.

Among the information contained in the meta-model we find:

- metrics, e.g. NOC, LOC (Table 4.1 contains the complete list);
- micro-structures, e.g. design pattern clues, elemental design patterns, micro-pattern (see Chapter 5 for descriptions and references).

Structure of the MARPLE meta-model

Three kinds of information are present in the representation of the system (see Figure 3.4):

- information about the *logical* structure of the system: here we find entities belonging to the procedural and object-oriented programming paradigms, e.g. classes, methods, attributes;
- information about the *physical* structure of the system: the files and folders containing the code and every other resource composing the system;
- meta-data associated to logical or physical entities, i.e. metrics and micro-structures.

The three kind of information are combined in a single meta-model, which relates them in a simple and practical way.

The central element of the meta-model is CodeEntity, which is specialized into different subtypes. A CodeEntity is a Measurable entity, so it is possible to associate a collection of Metrics to it for further analyses. Each CodeEntity can be linked to other CodeEntity elements by BasicElement entities[1] (representing micro-structures). The CodeEntity acts also as a link to code for any external model, e.g. meta-model for the representation of design patterns.

Diffusion

As already introduced, the meta-model is used in the MARPLE project (see Chapter 4 to see the usage of the meta-model). It is generic enough to be compatible with other existing meta-models, e.g. FAMIX, KDM.

Documentation

The meta-model is documented in a research paper I am co-author of [13].

Final considerations

The meta-model employed by MARPLE is targeted to be simple and concise. It is possible to extend it by linking other models to its entities, but in most cases the flexibility brought by the meta-data entities is enough, providing a good off-the-shelf modularity.

The inclusion of both qualitative (BasicElement entities) and quantitative (Metric entities) information contribute to enrich the global representation of the system, and providing direct support to the analysis process. Every association is modeled as bidirectional, improving the ease of navigation of the models.

Some differences with respect to FAMIX and DMM are:

- the representation of the hierarchical structure of files and directories;
- the absence of commonly modeled relationships, e.g. use, call, inheritance;
- the absence of an entity representing comments;
- the absence of an entity representing annotations.

The model is designed to support a variety of languages, both procedural and object-oriented, but it is tested mainly against the Java programming language.

[1] "basic elements" is the previous name for micro-structures. The definition and implementation of the model will keep that name until a new major revision will be available.

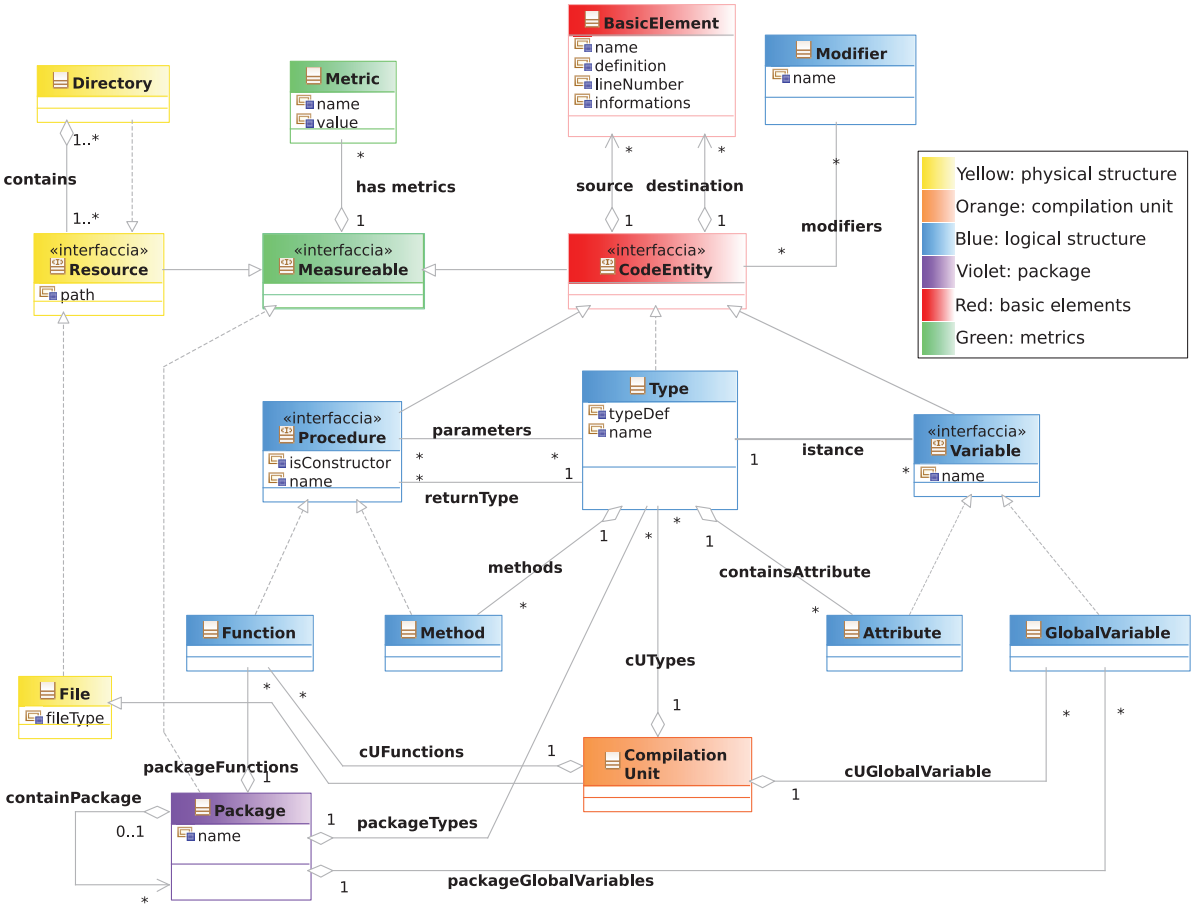


Figure 3.4: UML class diagram for the MARPLE meta-model

3.1.6 Other models

Columbus Ferenc et al. [70, 68] developed in 2002 a meta-model in the context of the Columbus design pattern detection tool (reviewed in Section 2.2.2). The meta-model supports only C++ and, considering the context of the application, it is quite complex, as it directly represents most of the features of the language. Its diffusion is apparently limited to the projects mentioned in the early publications.

Datrix A meta-model developed by Bell Canada Inc. [98] in 2000. It supports C, C++ and Java but, due to the lack of documentation and reference implementation, it can be considered unsupported.

Dynamix A meta-model [80, 79], compatible with MOF [141], created to describe the dynamic behaviour of an application.

3.2 Models for design pattern detection tools

This section contains the description of a meta-model available in the literature for the representation of design pattern instances. The only other meta-model available for the same purpose is the one used in MARPLE (see Subsection 6.2.1). The same model was employed to build DPB, a web platform for the benchmark of design pattern detection tools [15], defined and implemented by my research group.

The described meta-model is DPDX, proposed by Kniesel et al. [119] in 2010 for the modeling of design patterns. It is composed of these parts:

- the Schema meta-model specifies the entities to use in the description of design pattern schemas;
- the Program element meta-model describes the elements that are used to represent source code, and that are referenced by the modeled design pattern instances;
- the Result meta-model collects the entities needed to model the design pattern instances found in an analyzed source code.

All the instances in a result meta-model, referring to the same design pattern, must comply with the same schema, defined by an instance of the schema meta-model. Result meta-model instances keep references to source code artifacts using links to the Program element meta-model.

The goal of the meta-model is to provide a common representation to be used as a standard encoding for every tool in the design pattern detection area.

The introduction and the adoption of a common exchange format by the active design pattern detection tools would bring many advantages and would simplify the interaction among existing tools able to provide detection, validation, visualization, comparison and fusion of design pattern instances.

The approach adopted by the authors of DPDX is to define a representation that is sufficiently generic to satisfy the needs of any kind of tool of this community.

3.2.1 Structure of DPDX

- Schema meta-model (Figure 3.5): this meta-model specifies the structure of the definition of a design pattern. Each definition is seen as a collection of roles, which are related to each other using arbitrary relationships. Instances of the meta-model are graphs composed by interconnected roles.
- Result meta-model (Figure 3.6): this meta-model represents design pattern instances detected in a certain source code base. Each instance consists of a collection of RoleAssignment entities, each one filled with the information regarding the particular instance. Each RoleAssignment refers to the code using an association to a ProgramElement entity.
- Program element meta-model (Figure 3.7): this meta-model defines the fundamental entities needed to represent the interesting parts of software system's source code. The choice of the elements represented by the model is limited to the ones interesting in a design pattern detection task, and is not intended to be a generic software representation model. The categorization of the entities is based on the distinction in four different types, and seems to be generic enough to represent a large number of programming languages.

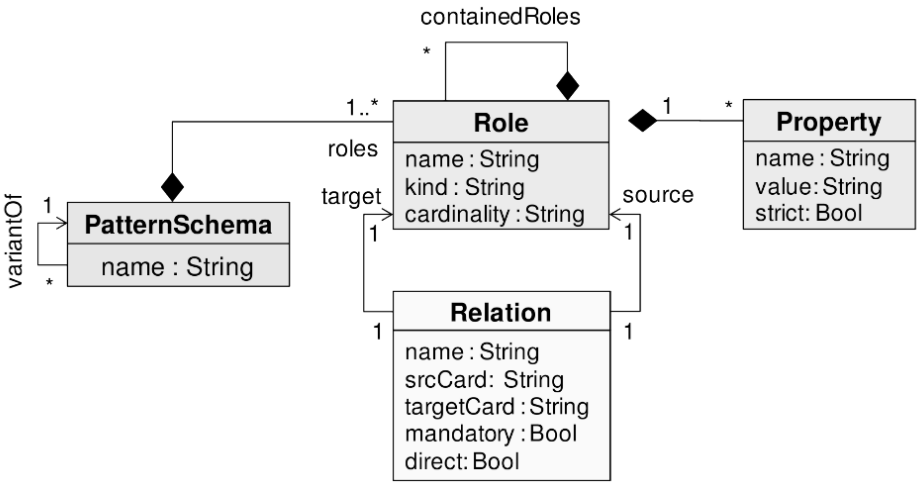


Figure 3.5: DPDX: Schema meta-model

3.2.2 Diffusion

No known reference implementation is available for DPDX.

Documentation

The DPDX meta-model was published in an international conference paper by Kniesel et al. [119] and in extended technical report from the same authors [117].

Final considerations

DPDX promises to be the next de facto standard for the representation of the data produced and consumed by the tools in the design pattern detection community. Unfortunately, probably due to the young age of the meta-model, and to the lack of a detailed and formal specification, its application is still limited.

The first impression of the model is that it is wide and generic; this kind of impression can limit its adoption by tool developers who want to use it for their practical needs. A shared and agreed set of design pattern schemas (using the schema meta-model) and example pattern instances would invite developers of design pattern detection tools to embrace the model, going in the direction of real interoperability.

Finally, the choice to specify the schemas of design patterns using the XSD [187] format can be considered a point of possible improvement, as XSD does not allow the authors of new definitions to automatically verify their formal correctness, because no formal meta and meta-meta models were employed.

Some consideration about the differences between the model employed in MARPLE and the DPDX model are reported in Subsection 6.2.1.

3.3 Conclusion

This chapter reviewed the available literature in reverse engineering modeling, with a particular focus on design pattern detection tasks. A lot of work was done by the research community to define meta-models to be used for tooling and data exchange. The current diffusion of the available models seems generally low and limited to the projects they were created for.

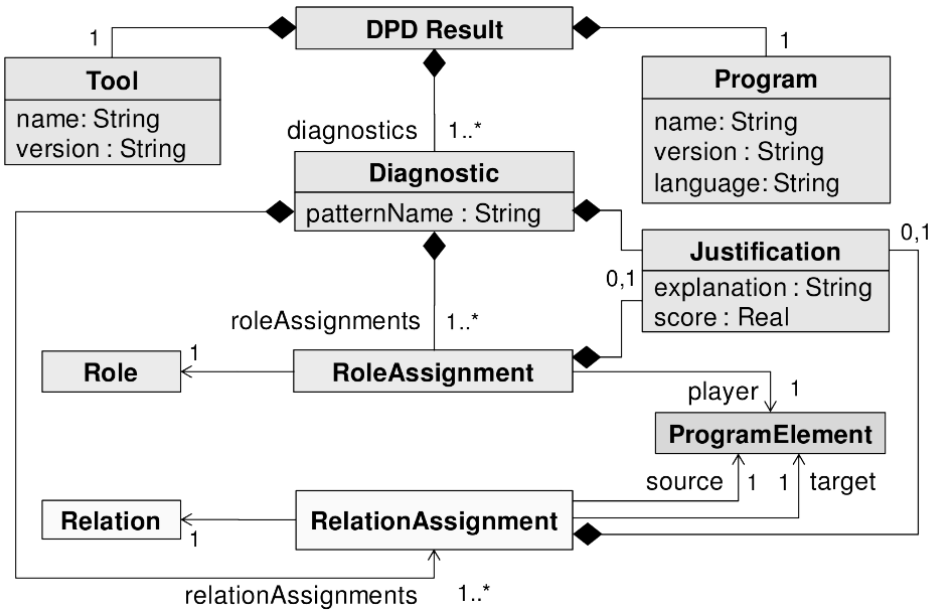


Figure 3.6: DPDX: Result meta-model

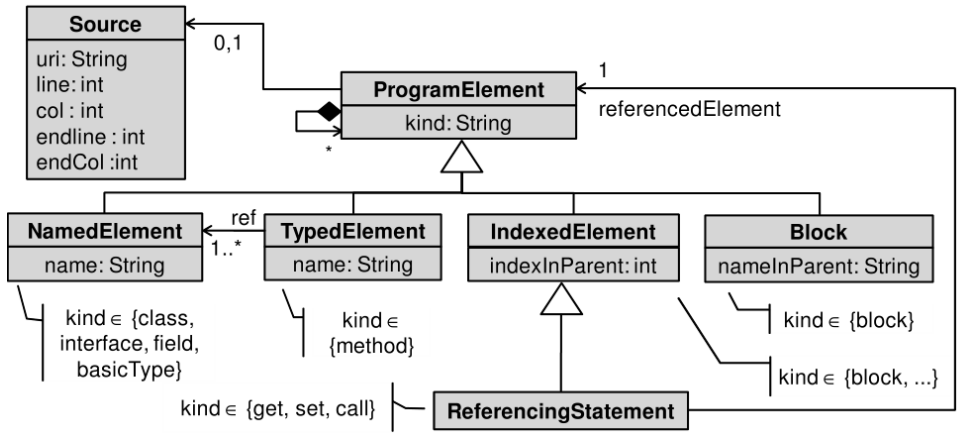


Figure 3.7: DPDX: Program element meta-model

The model used by the major number of projects seems to be FAMIX, mainly because it is the oldest among the cited models. KDM was designed by the Object Management Group to be the next standard for reverse engineering modeling, but its adoption is slow, probably because its specification is huge, and a full implementation, at least in one language, is not yet available. The particular community of design pattern detection is younger and only two models are available: DPDX aims to be a complete specification, and the one employed in MARPLE is more concise and application oriented, while imposing stronger modeling constraints. PADL contains a model for the representation of design patterns, but its specification is not documented.

Chapter 4

An introduction to MARPLE

MARPLE (Metrics and ARchitecture Reconstruction PLugin for Eclipse) is the tool implementing all the functionalities developed for this thesis.

An overview of the principal activities performed through MARPLE is depicted in Figure 4.1, which shows the general process of data extraction, design pattern detection, software architecture reconstruction and results visualization. The information used by MARPLE is obtained by an Abstract Syntax Tree (AST) representation of the analyzed system.

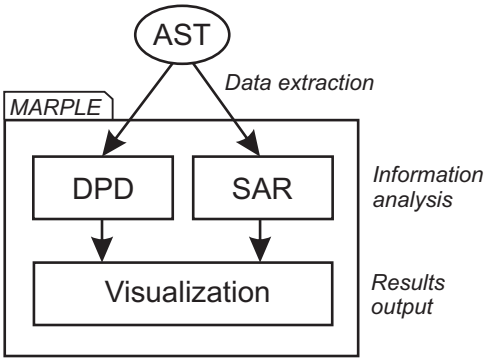


Figure 4.1: An overview of the general process

Design pattern detection and software architecture reconstruction activities both work on information extracted from the ASTs of the analyzed system. This information set is composed of elements, or facts, which are called *micro-structures* (see Chapter 5), and of metrics that have been measured inside the system. Both of these kinds of information are used to have an abstract and more consistent view of the system, instead of relying on the code or on the AST directly.

4.1 Architecture

The overall architecture of MARPLE is depicted in Figure 4.2. It is constituted of five main modules that interact with one another through a common model. The five modules are the following:

Information Detector Engine: builds the model of the system and collects both micro-structures and metrics, starting from an AST representation of the source code of the analyzed project; micro-structures and metrics are stored in the model. This module implements the first level of abstraction provided by MARPLE.

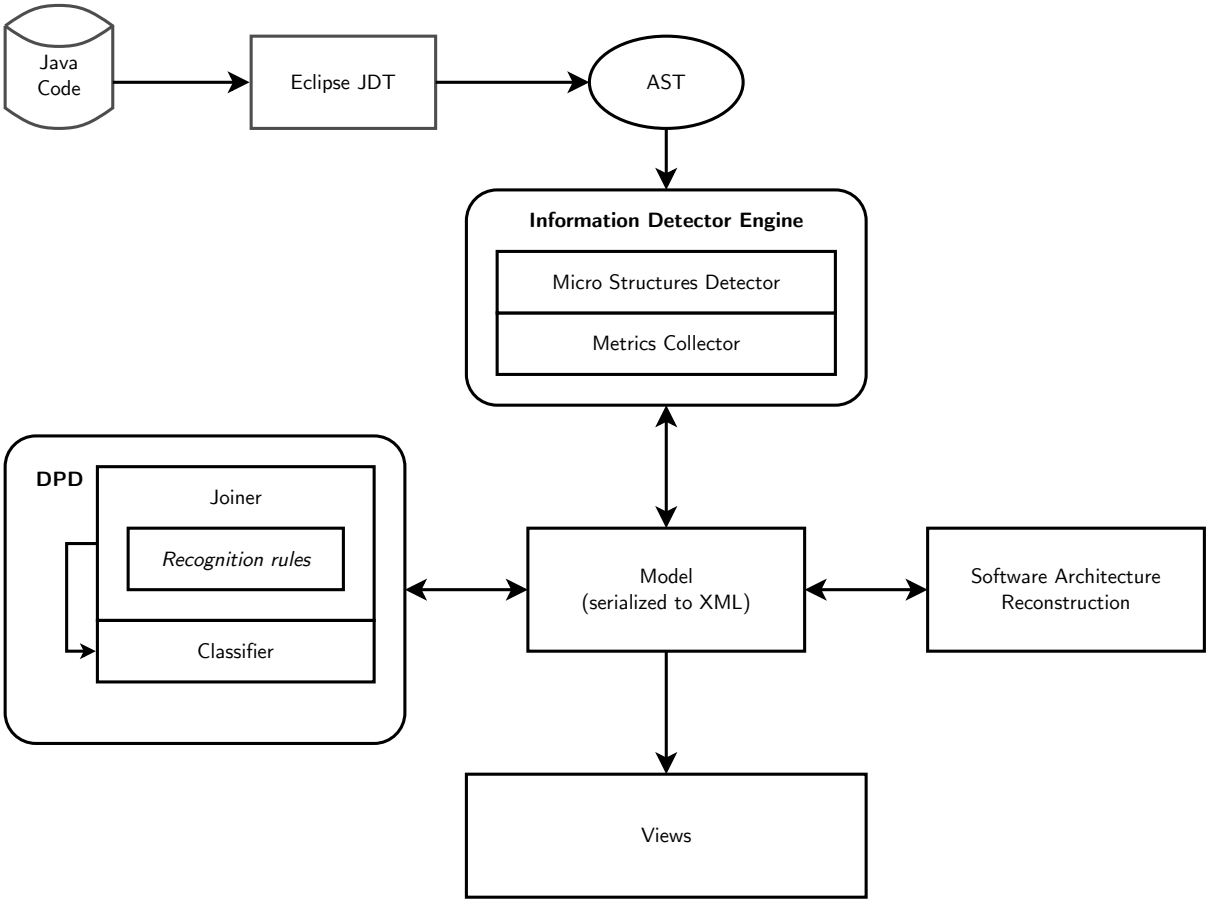


Figure 4.2: The architecture of MARPLE

Joiner: extracts all the potential design pattern candidates that satisfy a given definition, working on the micro-structures extracted by the *Information Detector Engine*.

Classifier: tries to infer whether the groups of classes detected by the *Joiner* could effectively be realizations of design patterns or not. This module helps to detect possible false positives identified by the *Joiner* and to evaluate the similarity with correct design pattern implementations, by assigning different confidence values.

Software Architecture Reconstruction: obtains abstractions from the target project basing on the available micro-structures and metrics.

Views generation: provides an organic view of the project analysis results. Through this activity, the user will see both the results produced by the detection of design patterns and the views provided by the SAR module.

The union of the Joiner and the Classifier module form the design pattern detection module: MARPLE-DPD.

The following sections describe the technologies involved in the development of MARPLE, and the next two discuss the Information Detector and Software Architecture Reconstruction modules. The Joiner and Classifier modules are discussed respectively in Chapter 6 and Chapter 7, because they address the central arguments of this thesis.

4.2 Technologies

MARPLE is an Eclipse plugin, and relies on technologies available in the Eclipse framework in order to implements its functionalities. The remainder of this section introduces the main Eclipse technologies exploited in MARPLE.

4.2.1 Java Development Tools

Java Development Tools (JDT) [180] is the main plugin present in the most common Eclipse installations; it provides all the Java IDE functionalities in Eclipse, like compiling, syntax highlighting, debugging. It provides also the implementation of the ASTs used for the analysis of Java code: every Java file (called *Compilation Unit*) can be explored through its AST, implementing an interface that follows the *Visitor* design pattern. A feature of this particular AST implementation is that it is able to link the nodes of the AST to other descriptors, filled in by the compiler. For example, it is possible to retrieve information about the type of a local variable from one of its references (different from the declaration), or to determine if a method overrides another one. All this pieces of information are very useful for our analysis purposes and are not directly available using other general purpose parser generators, like ANTLR [148], CocO/R [135] or JavaCC [105].

4.2.2 Eclipse Modeling Framework

Eclipse Modeling Framework [181, 176], (EMF) is an Eclipse plugin that provides a way of defining models using a formal meta-meta-model, called ECore. ECore is an implementation of EMOF (Essential MOF), a variant of the MOF [141] standard from OMG, that is used also for the definition of UML. EMF lets the user define new models using ECore, and can generate the Java code implementing the model; the generated code has interesting features, like constant-time reflection using the meta-model definition constants, bidirectional associations and events over the entities and their properties. The framework is able also to serialize models to XMI [142], off the shelf; other serialization formats can be defined by the user. ECore is translatable also to XMLSchema [187]: therefore a common serialization format is the XML direct translation of the model, which has to be defined in the details by the user. Other plugins provide also persistence over existing object oriented persistence frameworks, i.e. Hibernate [106] or EclipseLink [182]. Finally, EMF provides a query language over the model that resembles SQL.

4.2.3 Graphical Editing Framework

Graphical Editing Framework (MVC)

Graphical Editing Framework (GEF) [183] is the plugin used for the generation of views. The graphical environment of Eclipse is based on the SWT [185] library, that provides all the standard GUI toolkit functionalities, like menus, windows, text fields, etc; it also provides canvases for graphical views or editors. In Eclipse, the library used to draw on these canvases is Draw2d. GEF relies on Draw2d and defines a way of using it following the MVC pattern: it defines separate abstractions to define the model of the graphical representation, the visual components and ways to manipulate them, if needed. It is a neat general solution to a problem that normally tends to get more and more complicated.

Zest

Zest is a recent plugin in the Eclipse community. It is a specialized library for drawing graphs. It is used instead of GEF for the SAR views that represent graphs in MARPLE. It provides graphs layouts and models off the shelf, and it is simpler to use than GEF for this purpose, because GEF is more generic. It uses Draw2d like GEF, but it integrates well also with JFace [184], which is another library that abstracts the use of SWT.

4.3 Information Detector Engine

4.3.1 Micro Structures Detector

Currently, the *Micro Structures Detector* (MSD) module has been completely developed and tested. As already introduced, its goal is to detect the so-called *micro-structures*, which are explained deeply in Chapter 5. In order to have an idea of what micro-structures are, it is possible to see them as facts about a code element or a pair of them.

Micro-structures are not ambiguous (as on the contrary design patterns may be), and once a micro-structure has been specified in terms of the source code details that are used to implement it, it can be correctly detected.

Micro-structures are extracted by visitors that parse an AST representation of the source code; each visitor reports instances of the micro-structures if the analyzed classes or interfaces actually implement them. Usually a visitor is able to detect only one kind of micro-structure, but in some cases it detects more than one, i.e. when two or more micro-structures have similar detection algorithm. For example, a visitor that has to check if a class is declared as **abstract** is almost equal to a visitor that has to check if the class is declared **final**, as both keywords are flags of the class declaration; it is therefore a good idea to put the detection of both micro-structures in the same visitor.

The information is acquired statically and is characterized by 100% rate of precision and recall. This value is due to the fact that these kinds of structures are meant to be *mechanically recognizable* [76], i.e. there is always a 1-to-1 correspondence between a micro-structure and a piece (or a set of pieces) of code, and the algorithm that detects the micro-structure is defined in terms of source code structure and properties.

By their definition, micro-structures can be recognized in a software system using different techniques, and the only constraint is that they have to refer to two (or also only one) *code entities* in the system; usually a code entity is a type, but it can be also a method or an attribute. Code entities are the central element of the representation model exploited in MARPLE. In general we can say that a code entity can be any element of the source code that can be identified in the system by its name and its namespaces. The micro-structures detected at the moment use simple conditions on the ASTs, or some simple algorithm, but it would be possible to exploit also (if needed) more complex analyses, e.g. data flow analysis [131], call graph analysis [81], dynamic analysis [27] or simulation, symbolic execution [154, 165], points to analysis [89].

The actual implementation of the MSD module exploits the JDT library that provides the classes and interfaces used to access the ASTs contained in a set of projects of the same workspace. The MSD module runs the analysis on a set of selected projects. The micro-structures are collected by a set of visitors, invoked sequentially on the ASTs of the classes constituting the project. The visitors work only on those nodes that may contain the information they are able to detect. For example, the visitors that look for method calls only analyze nodes that represent a method invocation, i.e. instances of the MethodInvocation class.

The instances of micro-structures coming from the visitors are then stored in the central model, which is explained in Subsection 3.1.5. In particular cases visitors can query the model in

Table 4.1: Metrics measured by MARPLE

Name	Definition	Name	Definition
AHNL	Number of overridden attributes	NMAA	Number of attribute accesses
C1M	Number of Function Objects attribute types	NMA	Number of added methods
CBO	Coupling Between Objects	NME	Number of extended methods
CC	McCabe's Cyclomatic Complexity	NMI	Number of inherited methods
DIT	Depth of Inheritance Tree	NMO	Number of overridden methods
DITI	Number of superinterfaces	NOA	Number Of Attrbutes
DITS	Number of superclasses	NOC	Number of nested subclasses
EC	Number of break or continue statements	NOCH	Number Of Children
HNL	Number of superclasses	NOIF	Number Of Interface
LCOM	Lack Of Cohesion in Methods	NOP	Number of parameters
LOC	Lines of Code in methods	NOS	Number of statements
MHNL	Number of overridden methods	WLOC	Lines of Code in classes
MSG	Number of sent messages in a method	WMSG	Number of sent messages in a class
NGA	Number of global accesses	WNI	Number of incoming calls
NIA	Number of inherited attributes	WNMAA	Number of field accesses

order to compose the information they found on the ASTs and the information already present in the model. In this way it is possible to define dependencies among different visitors. For this reason it is possible to define the order of invocation of visitors, by putting them in groups that are invoked sequentially.

The MSD module has been developed also for the .NET environment in a previous work [7].

4.3.2 Metrics Collector

The *Metrics Collector* module evaluates some object-oriented metrics that are useful for SAR. These metrics are exploited in the generation of some of the architectural views described in Section 4.4. Table 4.1 lists the metrics currently supported by MARPLE.

4.4 Software Architecture Reconstruction

One of the objectives of MARPLE is to support the user with the visualization of abstractions about the analyzed systems. Currently, the SAR module generates six kinds of views on a system:

- The *package diagram* of all the packages that form the analyzed system; it is a graph showing all the packages as nodes and their dependencies as edges. An example of the package diagram, for the JHotdraw v.6.0b1 system, can be seen in Figure 4.3. The view allows moving and reordering entities, achieving a more readable diagram, like the one shown in Figure 4.4.

- The *package metrics* view is a table reporting a set of dependency-related metrics about each package. Figure 4.5 shows an example of the table on the same JHotdraw version.

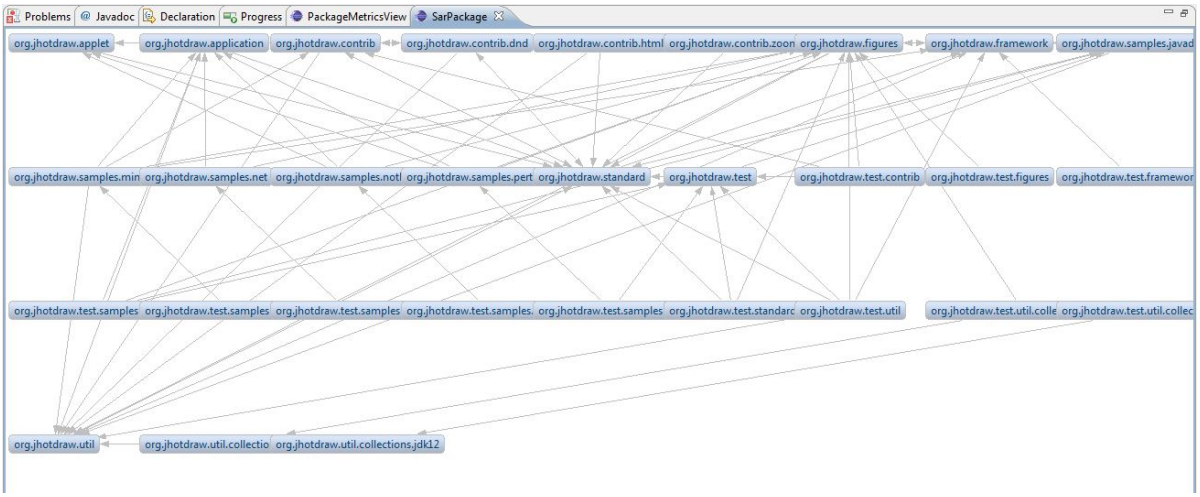


Figure 4.3: Package View example: JHotdraw 6.0b1

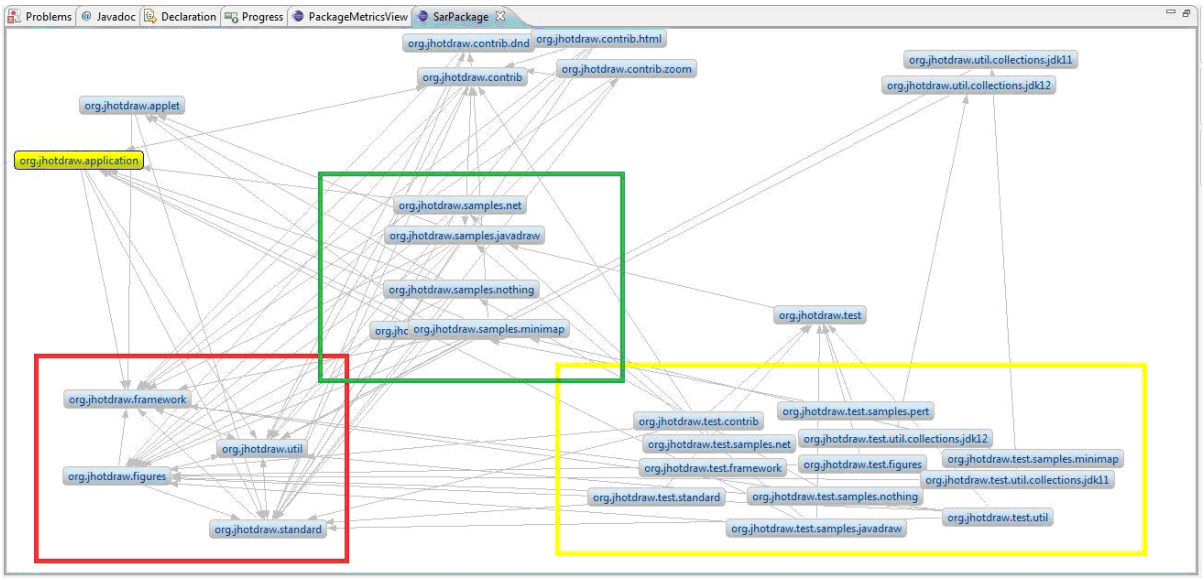
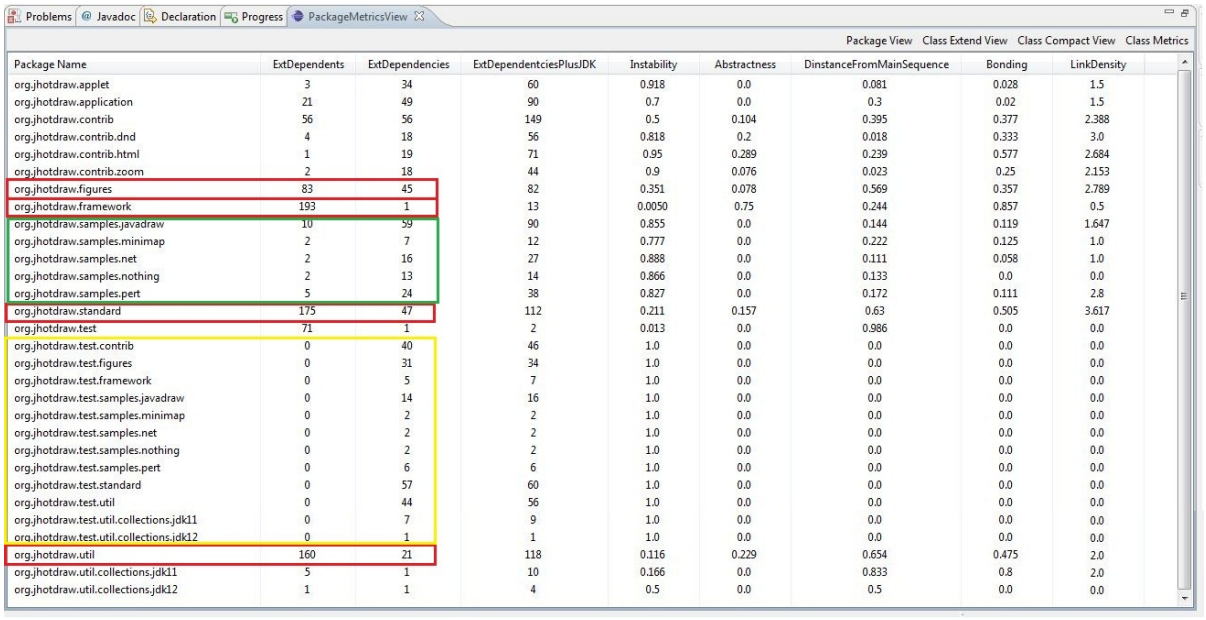


Figure 4.4: Package View example: JHotdraw 6.0b1 reorganized



Problems | Javadoc | Declaration | Progress | PackageMetricsView

Package View | Class Extend View | Class Compact View | Class Metrics

Package Name	ExtDependents	ExtDependencies	ExtDependentciesPlusJDK	Instability	Abstractness	DinstanceFromMainSequence	Bonding	LinkDensity
org.jhotdraw.applet	3	34	60	0.918	0.0	0.081	0.028	1.5
org.jhotdraw.application	21	49	90	0.7	0.0	0.3	0.02	1.5
org.jhotdraw.contrib	56	56	149	0.5	0.104	0.395	0.377	2.388
org.jhotdraw.contrib.dnd	4	18	56	0.818	0.2	0.018	0.333	3.0
org.jhotdraw.contrib.html	1	19	71	0.95	0.289	0.239	0.577	2.684
org.jhotdraw.contrib.zoom	2	18	44	0.9	0.076	0.023	0.25	2.153
org.jhotdraw.figures	83	45	82	0.351	0.078	0.569	0.357	2.789
org.jhotdraw.framework	193	1	13	0.0050	0.75	0.244	0.857	0.5
org.jhotdraw.samples.javadraw	10	59	90	0.855	0.0	0.144	0.119	1.647
org.jhotdraw.samples.minimap	2	7	12	0.777	0.0	0.222	0.125	1.0
org.jhotdraw.samples.net	2	16	27	0.888	0.0	0.111	0.058	1.0
org.jhotdraw.samples.nothing	2	13	14	0.866	0.0	0.133	0.0	0.0
org.jhotdraw.samples.pert	5	24	38	0.827	0.0	0.172	0.111	2.8
org.jhotdraw.standard	175	47	112	0.211	0.157	0.63	0.505	3.617
org.jhotdraw.test	71	1	2	0.013	0.0	0.986	0.0	0.0
org.jhotdraw.test.contrib	0	40	46	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.figures	0	31	34	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.framework	0	5	7	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.samples.javadraw	0	14	16	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.samples.minimap	0	2	2	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.samples.net	0	2	2	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.samples.nothing	0	2	2	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.samples.pert	0	6	6	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.standard	0	57	60	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.util	0	44	56	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.util.collections.jdk11	0	7	9	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.test.util.collections.jdk12	0	1	1	1.0	0.0	0.0	0.0	0.0
org.jhotdraw.util	160	21	118	0.116	0.229	0.654	0.475	2.0
org.jhotdraw.util.collections.jdk11	5	1	10	0.166	0.0	0.833	0.8	2.0
org.jhotdraw.util.collections.jdk12	1	1	4	0.5	0.0	0.5	0.0	0.0

Figure 4.5: Package Metrics View example: JHotdraw 6.0b1

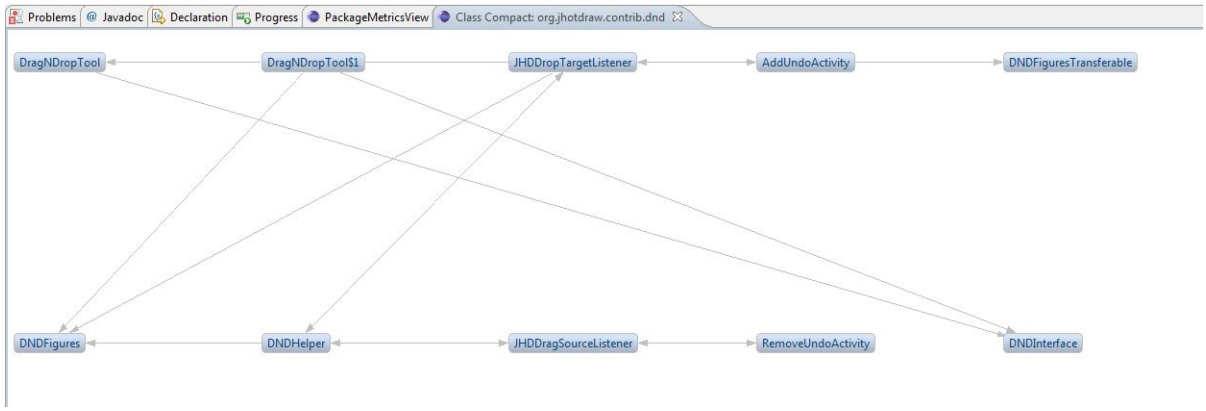


Figure 4.6: Class compact view example: package org.jhotdraw.contrib.dnd of JHotdraw 6.0b1

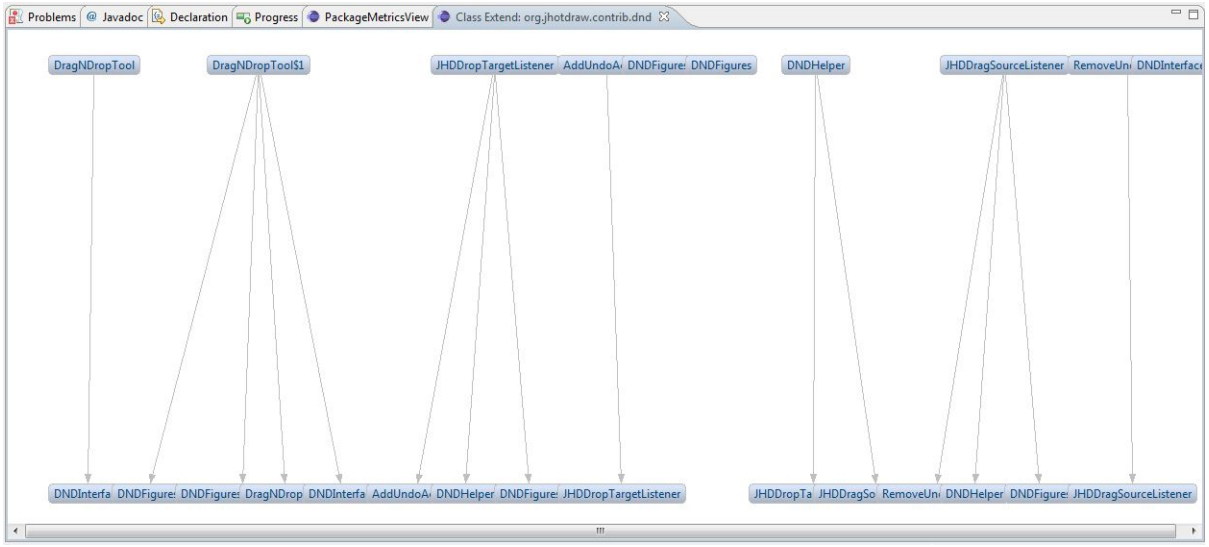


Figure 4.7: Class extended view example: package org.jhotdraw.contrib.dnd of JHotdraw 6.0b1

- The *class compact diagrams* of each package constituting the system. In this view (an example is shown in Figure 4.6), all the classes and interfaces belonging to the package are shown as a single graph, where the nodes correspond to the classes and interfaces, while the edges are the relationships connecting them. Relationships represent a selected set of micro-structure, like method calls

- The *class extended diagrams* of each package constituting the system. This view is characterized by many graphs, one for each class or interface belonging to the package. Each graph (an example is shown in Figure 4.7) reports only the relationships its subject class or interface has with the other classes or interfaces that constitute the system. In this way, the graphs will not be overwhelmed with a huge number of edges, letting the users focus on single classes without minding to the rest of the system.

These views are obtained by different kinds of information: the package and the class diagrams exploit the output coming from the MSD. More specifically, the SAR functionalities related to the generation of the *class compact* and of the *class extended* diagrams are achieved only through the analysis of the *elemental design patterns* [168] detected by the Micro Structure Detector. These elements revealed themselves very useful for the identification and definition of the relationships that are typical of class diagrams and which link the various classes constituting the analyzed project. These relationships, underlining the architectural constraints, let the users have a general overview of the classes' structures and aggregations.

The integration of the information coming from the design pattern detection module in the software architecture reconstruction views could enhance the program comprehension experience of the developer, helping to visually locate pattern instance in the system, and their relation with the other classes. The integration is one of the planned future developments.

4.5 Distributed MARPLE

An experimental prototype was developed to allow MARPLE to work in a client-server environment; the prototype's architecture is summarized in Figure 4.8). The prototype was

developed because loading the projects ASTs using the Eclipse APIs requires a large amount of memory (i.e. the AST of Batik, a system composed by 1643 java files, requires about 1700Mb of memory). The distributed version of MARPLE splits the classes of the system into k sets and the BED[1] nodes analyze only their own set. This solution allows us to reduce the memory requirement for the nodes; this solution is used in the normal version of MARPLE, serializing the analysis of each set using only one BED instance. This choice has the same memory benefits of the distributed one, but is more practical and easy to setup.

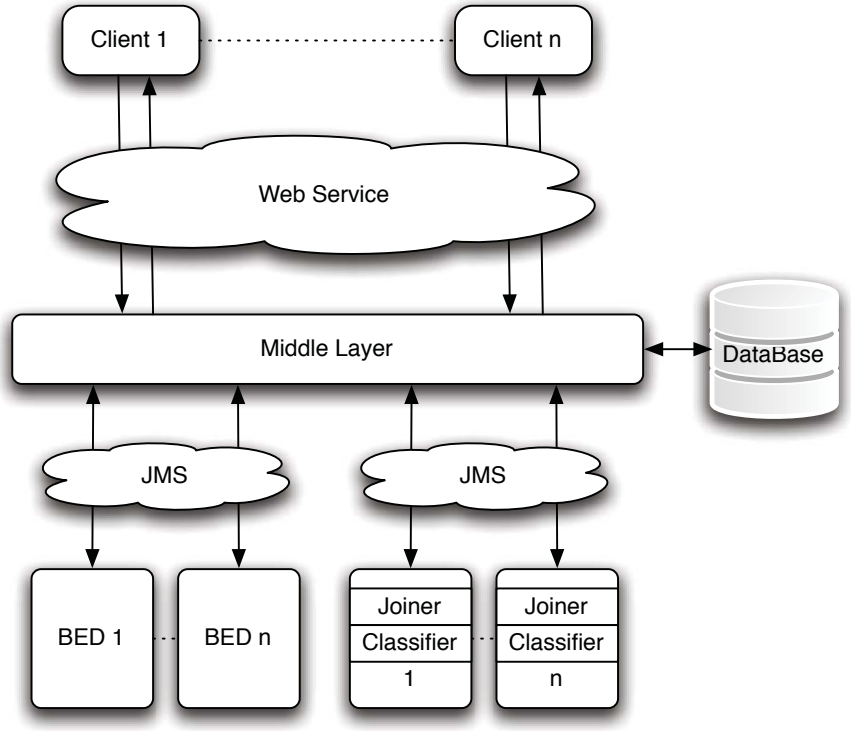


Figure 4.8: The architecture of distributed MARPLE.

Another problem that convinced us toward the development of the distributed version is the computational requirement of the classifier module. It is well known that some machine learning algorithms require a long time in order to build the model of their input dataset.

For all of these reasons, we decided to implement a distributed version in order to allow users to run their analysis through the internet on the elaboration server and to asynchronously check the results of the elaboration. This project is based on the J2EE v.5 platform and precisely on the Glassfish [147] application server.

The system architecture (see Figure 4.8) is composed of four main modules:

Client: this module is developed as an Eclipse plugin (maintaining MARPLE's interface) and allows users to use the elaboration service. Users using this service can send their projects, they can check current elaboration status, and they can also manage (retrieve, modify, delete) all the already performed analyses.

Server: this module is developed on a J2EE Application Server, and it implements the middle layer of the system. It also implements the Web Service in order to receive users' requests,

[1]BED stands for Basic Element Detector. The name refers to "basic element", which was the name given to micro-structures in the early development stages

manages the persistence backend (Database) that contains the information about the users, and finally starts the elaboration of a project. When an elaboration starts, first the server calls in parallel all the BED nodes, and at the end of their elaboration it calls in parallel all the Joiner-Classifier nodes. This job serialization is necessary because, in order to go in execution, the Joiner-Classifier node has to own all the detected basic elements.

BED Node: this module receives from the server, through JMS [93], a set of classes and the entire project source code; then it simply runs the BED on this set and returns to the server the detected micro-structures.

Joiner-Classifier Node: this module receives a rule from the server, through JMS, which specifies the pattern to find and all the detected micro-structures. Next it sequentially runs the Joiner and the Classifier Module and then it returns to the server all the found pattern instances with their classification values.

4.6 Conclusion

This chapter described the architecture of MARPLE, the tool implementing the design pattern detection functionalities described in this thesis through its MARPLE-DPD module. MARPLE is also capable of reconstructing several views of analyzed software, to facilitate its comprehension. The design pattern detection and software architecture reconstruction functionalities of MARPLE are based on the detection of *micro-structures*: Chapter 5 describes the concept of micro-structure and the different kinds of micro-structure integrated in MARPLE.

Chapter 5

Micro structures

In tasks of system modernization, program comprehension, system architecture reconstruction, and static software analysis in general, a common approach to the solution is to represent explicitly useful pieces of information about the system. This information is found directly on the observed system, and summarizes some mid/low level concepts that are useful for further analyses. In the MARPLE project, all these pieces of information are called *micro-structures*.

A micro-structure is defined as a fact or relationship between two entities in the code (e.g. classes, attributes, methods). It abstracts a concept that is possible to exactly retrieve from the source code. In fact, every micro-structure has the property of being mechanically recognizable, which means it is possible to write an algorithm able to retrieve every micro-structure instance in the source code without errors. Because of this property, the extraction of micro-structures is characterized by 100% precision and recall.

Different kinds of micro-structures have been proposed in the literature, with different objectives, like design pattern detection, identification of common programming techniques and extraction of architectural relationships. As far as design pattern detection is concerned, the approaches based on the recognition of micro-structures inside the code and other input generally exploit static analysis of source code.

The relevance of micro-structures in the general design pattern detection process is substantial. To obtain an effective detection process, with good rates of precision and recall, micro-structures should help to identify those aspects that are fundamental for the presence of patterns inside the code, because they are the primary mean for its representation.

Next sections explain the kinds of micro-structures integrated in the MARPLE project.

5.1 Elemental Design Patterns

Elemental Design Patterns (EDPs) [169] were introduced by Smith and Stotts, in the scope of the SPQR tool [171]. SPQR (System for Pattern Query and Recognition) is a tool for design pattern detection for C++ that uses EDPs to build the representation of the system it analyzes; the definition and implementation of EDPs for Java was performed by our research group. EDPs were defined to inherit from design patterns their ability to capture design intents, but being significantly simpler than design patterns. In SPQR EDPs and their composition rules are expressed formally in terms of ρ-calculus, which represents a sub-set of σ-calculus extended with new reliance operators. Design patterns and their implementation variants are not statically described, but they are dynamically inferred through the formalized rules.

There are sixteen EDPs subdivided into three categories:

Object Elements contains three EDPs related to the creation and the referencing of objects as

well as to the presence of abstract methods inside an abstract class, or interface methods inside an interface;

Object Behavioural collects twelve EDPs which represent the various forms of possible method calls;

Type Relation contains a single EDP representing the inheritance relationship between two classes.

EDPs are defined with the same description structure used in [74] for the presentation of the design patterns. For a complete description of each EDP refer to [169]. As examples, we describe the Create Object EDP and the Delegate EDP.

5.1.1 Create Object EDP

Intent

This EDP represents the construction of an object of a certain class (see Figure 5.1).

Category

Object Elements.

Structure

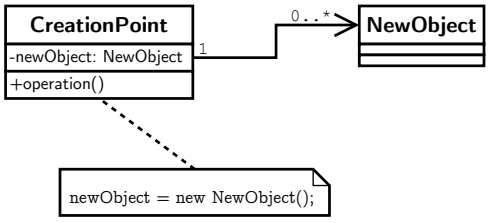


Figure 5.1: Create Object EDP UML diagram

Sample Code

```
public class CreationPoint {
  private NewObject no;
  public void operation(){
    no = new NewObject();
  }
}

public class NewObject {
  public NewObject(){...}
}
```

5.1.2 Delegate EDP

Intent

This EDP delegates part of the current work to a method of another class (see Figure 5.2).

Category

Object Behavioural.

Structure

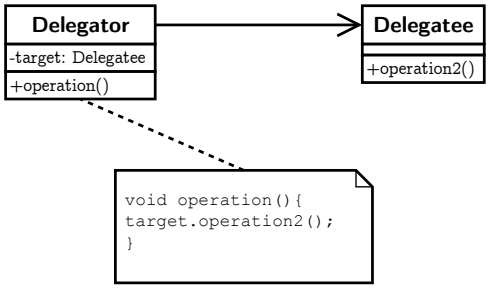


Figure 5.2: Delegate EDP UML diagram

Sample Code

```
public class Delegator {
  private Delegatee target;
  public void operation(){
    target.operation2();
  }
}
public class Delegatee {
  public void operation2(){
    ...
  }
}
```

5.1.3 Elemental Design Pattern catalog

Table 5.1 shows the list of all the EDPs and their respective category. The ones in the *Object Behavioural* are all different types of method invocations. Their differences are characterized over three parameters: the referred instance, the signature of the method and the type of the class. The referred instance can be {same, different}, the signature can be {same, different}, and the class type can be {same, parent, sibling, unrelated}. Not all the combinations are possible: the same instance can have only the same type. The example Delegate EDP represents the (instance: different, signature: different, type: unrelated) combination.

5.2 Micro Patterns

Micro patterns were introduced by Gil and Maman [76] in order to capture very common programming techniques. Micro patterns can be thought of as class-level traceable patterns, *i.e.*, structures similar to design patterns which can be mechanically recognized and which stand at a class abstraction level. A micro pattern is traceable if it can be expressed as a simple formal condition on the attributes, types, name and body of a software module and its components. Currently, there are 27 micro patterns subdivided into eight categories. Authors do not assert

Table 5.1: The list of the Elemental Design Patterns

Name	Category
CreateObject	Object Creation
Abstract Interface	Object Structural
Inheritance	Type Relation
Retrieve	Object Structural
Delegate	Object Behavioural
Redirect	Object Behavioural
Conglomeration	Object Behavioural
Recursion	Object Behavioural
RevertMethod	Object Behavioural
Extend Method	Object Behavioral
DelegatedConglomeration	Object Behavioural
RedirectedRecursion	Object Behavioural
DelegateInFamily	Object Behavioural
RedirectInFamily	Object Behavioral
DelegateInLimitedFamily	Object Behavioural
RedirectInLimitedFamily	Object Behavioural

that the set of the identified micro patterns is complete or exhaustive. The eight micro pattern categories are:

Degenerate State and Behaviour this category includes micro patterns describing interfaces and classes whose state and behaviour are degenerated. In most cases this means that the interface or class does not define any attribute or method;

Degenerate Behaviour these micro patterns are related to classes with no methods or with very simple ones;

Degenerate State this category is related to classes which have no state (*i.e.*, attributes), or their state is shared with other classes or they are immutable;

Controlled Creation the micro patterns belonging to this category describe special protocols for creating objects;

Wrappers this category collects micro patterns dealing with classes which have a single central instance field and methods working on it, so that the main functionalities are delegated to this field;

Data Managers these micro patterns are related to classes whose main purpose is to manage the data stored in a set of instance variables;

Base Classes the micro patterns belonging to this category describe different ways in which a base class makes preparations for its subclasses;

Inheritors the micro patterns in this category correspond to three ways in which a class can use the definitions of its superclass, *i.e.*, abstract method implementation, method overriding and interface enrichment.

The complete description of micro patterns is presented in [76]. We provide two examples of micro patterns: Restricted Creation and Designator.

5.2.1 Restricted Creation Micro pattern

An instance of this micro pattern can be found in those classes which do not have any public constructors and have at least one static field of the same type as the class.

Category

Controlled Creation.

Source Code

```
public class RestrictedCreationClass {
  static RestrictedCreationClass rcc;
  private RestrictedCreationClass() {
    ...
  }
}
```

Many classes which represent singletons satisfy these constraints, as the class java.lang.Runtime.

5.2.2 Designator Micro Pattern

One of the simplest micro patterns is the Designator micro pattern. It represents interfaces which are completely empty, *i.e.*, they do not declare any methods and do not define any static field or method, except inheriting them from one of its super interfaces.

Category

Degenerate State and Behaviour.

Source Code

```
interface DesignatorInterface {}
```

An example of this micro pattern can be found in the java.lang.Cloneable interface.

5.2.3 Micro Patterns catalog

Table 5.2 shows the list of all the micro patterns. The categories subdivided in a "Main Category" and "Additional Category", because some of the micro patterns belong to more than one category. It happens for the micro patterns addressing different concepts; for example, Record belongs to the "Data Managers" category, but being a class without methods belongs also to the "Degenerate Behavior" one.

Table 5.2: The Micro Patterns list

	Main Category	Pattern	Short description	Additional Category
Degenerate Classes	Degenerate State and Behavior	Designator	An interface with absolutely no members.	
		Taxonomy	An empty interface extending another interface.	
		Joiner	An empty interface joining two or more superinterfaces.	
		Pool	A class which declares only static final fields, but no methods.	
	Degenerate Behavior	Function Pointer	A class with a single public instance method, but with no fields.	
		Function Object	A class with a single public instance method, and at least one instance field.	
		Cobol Like	A class with a single static method, but no instance members.	
	Degenerate State	Stateless	A class with no fields, other than static final ones.	
		Common State	A class in which all fields are static.	
		Immutable	A class with several instance fields, which are assigned exactly once, during instance construction.	
	Controlled Creation	Restricted Creation	A class with no public constructors, and at least one static field of the same type as the class.	
		Sampler	A class with one or more public constructors, and at least one static field of the same type as the class.	
Containment	Wrappers	Box	A class which has exactly one, mutable, instance field.	
		Compound Box	A class with exactly one non primitive instance field.	
		Canopy	A class with exactly one instance field that it assigned exactly once, during instance construction.	Degenerate State
	Data Managers	Record	A class in which all fields are public, no declared methods.	Degenerate Behavior
		Data Manager	A class where all methods are either setters or getters.	
		Sink	A class whose methods do not propagate calls to any other class.	
Inheritance	Base Classes	Outline	A class where at least two methods invoke an abstract method on "this"	Degenerate State
		Trait	An abstract class which has no state.	
		State Machine	An interface whose methods accept no parameters.	Degenerate State and Behavior
		Pure Type	A class with only abstract methods, and no static members, and no fields.	
		Augmented Type	Only abstract methods and three or more static final fields of the same type.	
		Pseudo Class	A class which can be rewritten as an interface: no concrete methods, only static fields.	
	Inheritors	Implementor	A concrete class, where all the methods override inherited abstract methods.	
		Overrider	A class in which all methods override inherited, non-abstract methods.	
		Extender	A class which extends the inherited protocol, without overriding any methods.	

5.3 Design Pattern Clues

Design Pattern Clues are a kind of micro structure defined originally by Stefano Maggioni [129, 130], in our research group.

The definition started with the comparison of other types of micro-structures for design pattern detection [8], focusing on their relevance in the identification of GoF design patterns [74]; the result was that the detection of one kind of micro-structure is not enough to detect design patterns.

Hence we decided to study and propose a new category of micro-structures, named *design pattern clues*, with the aim to identify hints, conditions and concepts useful for design pattern detection. The aim to introduce a new kind of micro structure was to try to complement the information that can be extracted through other micro-structures, to obtain information to be used in a design pattern detection approach. So we started to analyze design pattern instances, extracted from examples and real systems, and we tried to understand, for each specific design pattern, what information the other micro-structures (in particular EDPs) were not able to capture. We put together the causes of bad detection for each pattern, and we tried to specify more precisely what all these causes had in common. Then we tried, when possible, to specify how to detect these causes in the code without ambiguity; clues were the output of this process. After having done this work for each pattern, we also made a further analysis of the clues coming from different patterns in order to avoid duplication and let their definition become more stable. It was a bottom-up task, done starting from the real implementations of patterns rather than from their theoretical definition.

Currently, we have identified 46 design pattern clues (definitions are available in Subsection 5.3.1), subdivided into the following nine categories:

Class Information: collects clues that can be identified analyzing a class declaration or that characterize a single class;

Multiple Class Information: collects clues that can be identified by the comparison among two classes (or more) and their contents;

Variable Information: gathers information about particular variables;

Instance Information: contains clues regarding particular instances of a certain class, and one clue representing a controlled instantiation mechanism;

Method Signature Information: collects clues that are identifiable analyzing the signature of a method;

Method Body Information: contains those clues that can be identified by only analyzing the body of any kind of method;

Method Set Information: collects clues whose details can be deducted analyzing the whole set of methods the involved classes declare and implement;

Return Information: includes those clues regarding various possible return modes from a method;

Java Information: collects clues which are strictly bound to the Java language.

All 46 clues can be automatically detected from source code, as they are representations of implementation issues which can be easily understood through an analysis of it. The clue catalogue is reported in Subsection 5.3.1. Each design pattern clue is automatically recognizable from source code using the Micro Structures Detector.

5.3.1 A Catalogue of Design Pattern Clues

Table 5.3 reports the complete catalogue of the design pattern clues. Each clue is identified by its name, its meaning, the design pattern it belongs to and the correspondent design pattern category ("C" for creational, "B" for behavioural, or "S" for structural design patterns), and eventually the other clues it depends on. In fact, the existence of some clues is subordinated to the presence of some others. For example, asserting that the Template implementor clue depends on the Template Method clue means that the existence of the Template Method clue is a necessary condition for the detection of the Template implementor.

Table 5.3: A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
Class Information	Final class	The class is declared final.	Singleton	C	
	Interface and class inherited	The class implements an interface and extends a class, providing therefore the only mechanism to simulate multiple inheritance in the Java language.	Adapter (based on classes)	S	
	Multiple interfaces inherited	The class implements n interfaces, with $n > 1$.	Adapter (based on classes)	S	
	Object structure child	The class is a *Visitable class* and it has at least an ancestor which is either an interface or an abstract class.	Visitor	B	Cross relationship, Visitable class
	Template implementor	A class extends another class implementing a *Template Method*	Template Method	B	Template Method
Multiple Classes Information	Façade method	The body of a method consists uniquely of method calls to classes which are not related with it, *i.e.* which are not a superclass, an implemented interface or the class itself. A facade method could also contain some object creations, but no other statements besides object creations or method calls.	Façade	S	
	Proxy class	A class implements an interface or extends an (abstract) class, and owns a reference to a class that implements the same interface or extends the same (abstract) class.	Proxy	S	
Variables Information	Private flag	The class maintains a control flag that is declared private. A flag belongs to a simple type, typically boolean; numerical fields are considered flags when their value is compared to form a boolean expression in a control statement.	Singleton	C	

Table 5.3: A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
	Static flag	The class maintains a control flag that is declared static. A flag belongs to a simple type, typically boolean; numerical fields are considered flags when their value is compared to form a boolean expression in a control statement.	Singleton	C	
Instance Information	Controlled self instantiation	The instantiation of an object of the same class occurs inside an if (or a switch) block, therefore under a condition.	Singleton	C	
	Private self instance	The class owns a private instance of the same class. Access to this instance can occur only from within the same class.	Singleton	C	
	Static self instance	The class has a static instance of the same class. Therefore this instance is unique inside the system.	Singleton	C	
	Single self instance	The class maintains a unique instance of the same class, no matter if it is static or not.	Singleton	C	
	Instance in abstract class	An abstract class maintains a reference to a different class.	Bridge	S	
	Same interface container	A class contains some kind of collection of objects that are compatible with an ancestor of the declaring class.	Composite, Interpreter	S	
	Same interface instance	A class contains a reference to an object whose type is compatible with an ancestor of the declaring class.	Decorator	S	
Method Signature Information	Controlled parameter	A method of a certain class receives as input a parameter used inside it to make some controls (*i.e.* the parameter is used in the condition of some `if` or `switch` block). If a method controls more than one of its input parameters, each one of these parameters will be an instance of this clue.	Abstract Factory, Builder, Factory Method	C	
	Factory parameter	A method of a certain class receives as an input parameter an object that belongs to a class defining some *Concrete product getter* methods.	Abstract Factory, Builder, Factory Method	C	Concrete product getter
	Protected instantiation	All the constructors within a given class are declared private.	Singleton	C	
	This parameter	A method receives the caller object as a parameter.	Observer, Visitor	B	

Table 5.3: A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
	Adapter method	Two types of Adapter method exist. It can be a method which is an implementation of an interface method and that calls a method belonging to the parent class; or it can be an overridden method from the parent class which calls a method belonging to a class that does not share common ancestors with the adapter method declaring class.	Adapter	S	Interface method (only in the first case)
	Interface method	A class implements a method declared inside an interface.	Adapter (based on classes)	S	
	Overriding method	A class overrides a method belonging to its superclass.	Adapter (based on objects)	S	
	Component method	A class declares a method that takes an object of the same class as its single parameter.	Composite	S	
	Cross relationship	Given two classes $C1$ and $C2$, $C1$ declares a method which accepts a reference to $C2$ as one of its parameters, viceversa $C2$ declares a method which accepts a reference to $C1$ as one of its parameters.	Visitor	B	
	Abstract cyclic call	A method invokes an abstract method within a cycle.	Iterator, Observer	B	
	Factory Method	A method contains a class instance creation statement and overrides a method belonging to the superclass or to one of the superinterfaces of the subject class.	Factory Method	C	
Method Body Information	Instance in abstract referred	A method of a class implementing *Instance in abstract class* invokes a method on the declared instance.	Bridge	S	Instance in abstract class
	Multiple redirections in family	A method contains a *Redirect in Family* [168] method invocation that is contained within a cycle.	Composite	S	
	Proxy method invoked	A proxy class invokes a method on the referred subject using a Redirect in limited family [168] method call EDP.	Proxy	S	Proxy class
	Visitable class	A method has a *Cross relationship* clue and passes the owner object (this) to the target method of the *Cross relationship*.	Visitor	B	This parameter, Cross relationship
	Template Method	A method calls at least an abstract method within its body.	Template Method	B	

Table 5.3: A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
Method Set Information	All methods invoked	A class invokes all of the public methods declared in a target class.	Adapter	S	
	Leaf class	A class extends another class without implementing or redefining the methods that are concerned with the handling of classes that are compatible with the same interface (therefore tagged with a *Component method* clue), or giving an empty implementation for such methods.	Composite	S	Component method
	Node class	A class extends another class implementing or redefining the methods that are concerned with the handling of classes that are compatible with the same interface (therefore tagged with a *Component method* clue).	Composite	S	Component method
Return Information	Concrete product getter	A class declares one or more methods that return objects of a type different from itself.	Abstract Factory, Prototype	C	
	Concrete products returned	A method returns objects that belong to subclasses of the declared return type.	Abstract Factory, Factory Method, Builder	C	
	Empty concrete product getter	A class declares one or more methods that return objects belonging to some other classes, but the implementation of these methods is empty, *i.e.* it consists only of a default return statement (as, for example, `return null`).	Builder	C	
	Empty method	A class declares one or more methods that return simple types, but their implementation is empty, *i.e.* it is only formed by a default return statement (for example, `return false` for the `boolean` data type).	Builder	C	
	Multiple returns	A method provides several possible return points.	Abstract Factory, Factory Method, Builder	C	
	Void return	A class defines a method that instantiates an object without returning it.	Builder	C	
	Cross hierarchy return	A method returns an object of a class belonging to a different hierarchy.	Iterator	B	

Table 5.3: A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
Java Information	Clone returned	A method returns a clone of a certain instance.	Prototype	C	
	Cloneable implemented	A class implements the java.lang.Cloneable interface.	Prototype	C	
	Prototyping constructor	A method defines a constructor which receives objects that can be cloned, as instances of classes implementing the java.lang.Cloneable interface.	Prototype	C	Cloneable implemented
	Controlled exception	A method of a class can throw an exception inside a control block.	Singleton	C	

5.4 Example of micro-structures in a design pattern instance

Considering the structural design pattern category, we propose the basic implementation of the *Composite* design pattern and we discuss the design pattern clues and EDPs that can be identified in it. The description of design pattern clues can be found in Subsection 5.3.1, while the description of EDPs are available in a separate catalog [168]. Next we show a simple Java implementation of the *Composite* design pattern:

```
public abstract class Component{
  public abstract void operation();
  public void add(Component c){}
  public void remove(Component c){}
}

public class Composite extends Component{
  private List<Component> components = new Vector<Component>();

  public void operation(){
    for (Component c : components)
      c.operation();
  }

  public void add(Component c){
    components.add(c);
  }

  public void remove(Component c){
    components.remove(c);
  }
}

public class Leaf extends Component{
  public void operation(){ ... }
}
```

5.4.1 Design Pattern Clues

Seven design pattern clues can be found in this basic implementation of the *Composite*:

Abstract cyclic call: (method signature information category) in the Composite class the method operation() invokes the Component.operation() abstract method within a cycle; therefore the Composite class contains an Abstract cyclic call;

Component method: (method signature information category) the two methods Component.add() and Component.remove() are instances of this clue, as they receive as parameter an object belonging to the same class;

Node class: (method set information category) Composite extends a class (Component) declaring Component methods and overrides them;

Leaf class: (method set information category) Leaf extends a class (Component) declaring Component methods without overriding them;

Same interface container: (instance information category) Composite contains a list of Components, which are objects that share the same interface with the Composite class; so Composite has a Same interface container clue;

Multiple redirections in family: (method body information category) the Redirect in Family EDP is detected inside a cycle (into the Composite.operation() method), therefore it is supposed to work on a set of elements. In this case, the operation() method is invoked on each Component object belonging to the Components list.

5.4.2 Elemental Design Patterns

In the implementation above of the Composite the following EDPs have been detected:

- one Abstract Interface EDP states that the Component class declares an abstract method, and consequently is an abstract class;
- two Inheritance EDPs connect the Composite and Leaf class through an extension relationship;
- a Create Object EDP can be found in Composite, where a list of Components is instantiated;
- finally a Redirect in Family EDP is detected in the Composite.operation() method. This method invokes a method with the same signature belonging to Composite's superclass.

5.5 Conclusion

The set of micro structures explained in this chapter is not intended to be complete. The reported micro structures are mainly the kinds that are detected by the micro structures detector.

My research group did an extensive comparison [16] of four micro-structures types, precisely: Design Pattern Clues, Elemental Design Patterns [169], Sub-Patterns [139] and Micro Patterns [76], with the aim to provide in the future a unified catalog of micro-structures. That work can be used in order to have a deeper insight of the topic. The exploitation of Design Pattern Clues in the refinement of third-party design pattern detection results has been investigated in another paper I wrote with my research group [21]. That research combines clues and EDPs in a way similar to the one the Joiner does (see Chapter 6), but with a different aim.

Both clues and EDPs are used, because they provide different kinds of information. For example, clues are strictly focused on formalizing constructs that are typical in the implementation of design patterns, while EDPs depict basic programming constructs (like object instantiations or method invocations) that are independent from the presence of design patterns inside the system to be analyzed. Clues and EDPs share the same detail level, as in general they can be detected by the analysis of single statements and elements of a class, like method invocations or field declarations. EDPs capture object-oriented best practices and are independent of any programming language; clues aim to identify basic structures peculiar to each design pattern. In spite of the differences between them, these micro-structures can be used both for the construction and the detection of design patterns.

Chapter 6

Joiner: extracting pattern instances from source code

The Joiner is the MARPLE module that uses the information coming from the micro-structures detected on the system to extract the pattern instance candidates from the analyzed system.

The approach used by the Joiner is to represent the system as a graph, where the micro-structures are the edges and the nodes are the types[1].

The task accomplished by the Joiner is to extract groups of classes from the system, where each group is a pattern candidate. A candidate is a group of classes, where a role is assigned to each class and the classes are organized according to a particular structure that fits well the conceptual organization of the pattern. The roles are assigned to each class exploiting the information represented by the micro structures (the edges of the graph), using graph matching techniques. The Joiner takes in input the graph of the system and the rules describing how to match each particular pattern; the rules are given in a declarative form. Next sections give all the details of the detection process.

6.1 Matching

The Joiner does not handle the system through its Abstract Syntax Tree (AST) representation, but it manages it as an *Attributed Relational Graph* (ARG) [115], where the set of vertices corresponds to the set of types (*i.e.* classes and interfaces) the project is constituted of, while the edges are the set of micro-structures that connect the types with one another. In fact, each micro-structure can be seen as a relationship between a type and another one (therefore depicted as an edge between two nodes of the graph), or as a relationship between a class and itself (depicted as a self loop on a graph node). Figure 6.1 shows an example of a graph built using the micro-structures already shown in the *Composite* pattern example reported in Section 5.4.

Micro-structures can be also useful for representing some kind of behavioural information, such as the resolution of polymorphic method calls. The representation of a software system using graphs is not new, as many other tools and exchange formats in the reverse engineering area use graphs as the basis of the system representation [191, 102, 59].

The system graph representation is directly derived from the output generated by the *Information Detector Engine*. As I briefly anticipated, this module tries to extract architectures that match a target structure, defined in terms of *Joiner rules*. In the simpler case a *Joiner rule*

[1]Methods and attributes are already supported, but due to the legacy implementation of the visitors collecting micro-structures, their exploitation is considered future work.

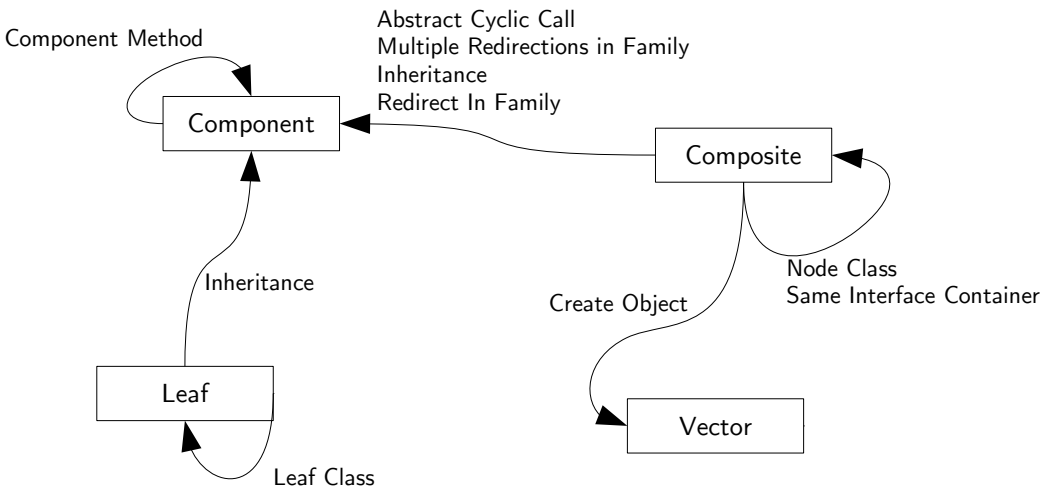


Figure 6.1: Example of an input graph of the Joiner

is a graph that collects roles and the micro-structures (edges) that must be present among the roles in order to satisfy the rule. The roles in the rule are the roles of the target design pattern, which are usually listed in the pattern definition. For example, if we want to extract the couple of roles $(R1, R2)$, where $R2$ is connected with an Inheritance to $R1$ and $R1$ has an Abstract Type, we may represent this rule as shown in Figure 6.2.

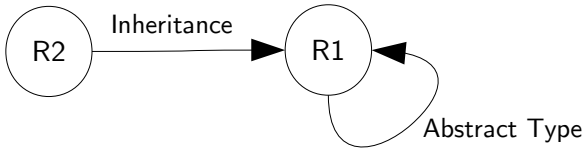


Figure 6.2: An example of Joiner rule

The original version of the Joiner module tried to match and extract this kind of architecture from the graph representing the system through an ad-hoc graph matching algorithm. The algorithm has been demonstrated to have linear complexity (in the average case) in the number of the classes of the system. All the details of the algorithm and the complexity demonstration can be found in [203]; the complexity of the detection is compatible with the graph matching literature [128], that demonstrates that it is possible to solve the subgraph matching problem in polynomial time when the input graph has bounded valence.

In the general case a Joiner rule is a propositional logic expression (plus predicates), where the micro structures are used as predicates and the roles of the pattern are used as variables. The current version of the Joiner has been enhanced by the exploitation of the SPARQL [150, 52, 198] query language, using the ARQ [107] Jena [108] library; this choice permitted to enhance the expressiveness of the rules, from a graphic form to a full propositional logic one. The original graphic form is interpreted as propositional logic expression where all the predicates are in conjunction and are not negated. The transformation for the graphical notation to the propositional one is quite straightforward: roles are the variables, and micro structures are translated to predicates on the roles, which are true when the micro-structure exists between the

instantiated roles. So testing a graph matching is the same as trying to match the translated expression, which consist of instantiating the variables (representing the roles) in a combination that satisfies the predicate composition. The advantage of using full propositional logic in graph matching is that the rules can specify also negations and disjunctions, augmenting the expressiveness of the rule.

As an example the rule represented in Figure 6.2 could be translated in this simple expression:

$$Inheritance(R2, R1) \wedge AbstractType(R1, R1)$$

The use of some kind of logic to express recognition rules for patterns is not new in the research area; it is also quite common to use predicates in order to represent properties of classes (or methods and attributes) or group of classes. For example Zhu and Bayley propose an approach to design pattern detection [31] that uses rules written in first order logic to detect patterns. Another example of usage of logic for pattern detection is in SPQR [171], where the EDPs are used in the detection rules, expressed using ρ-calculus. These and the other existing approaches for design pattern detection are reviewed in Chapter 2. The approach used in the Joiner uses a lighter logic form: only propositional logic with predicates is used, without the use of quantifiers or other higher-order constructs.

In order to be able to run a SPARQL query over the graph representing the system, the graph itself has to be translated in RDF [159] form. RDF is basically a way of specifying a graph for web applications and technologies. The use of it in the Joiner is internal and not really related to its features. The Joiner simply takes its graph representation, builds the equivalent data structure representing the RDF graph using the Jena framework and runs the SPARQL query over the data structure.

The output of this matching phase is a list of mappings of the roles on the types of the system that match the rule.

Following the example rule shown in Figure 6.2, and applying the rule to the example found in Section 5.4, the output would be the one found Table 6.1. The most important thing to note is that the Component class is present in both the mappings.

Table 6.1: Example of Joiner matching output

Roles →	$R1$	$R2$
Mapping 1 →	Component	Composite
Mapping 2 →	Component	Leaf

If we think about the example rule as the specification of a virtual "pattern" (we can call it *Implementor*), where the role $R1$ is the Abstract class and $R2$ is the Concrete class, the result shown in Table 6.1 is not expressive enough in order to represent instances of that pattern. In fact, as already underlined, there are two separate results coming from the same sample code, which should be obviously reported as *one* pattern instance. The consequence of this fact is that the Joiner rule needs to specify also how to merge the results matched in the analyzed system. Next section addresses this issue.

6.2 Merging

In the second phase of the detection, the matches in the graph are grouped to identify single pattern instances having more than one class for the same role. In the example rule, shown in Figure 6.2, we can impose that the results will be grouped by the assignment of $R1$,

obtaining tree-wise instances, each one formed by an assignment of $R1$ and a set of related $R2$ assignments. The model that fully implements these basic concepts has been proposed for generic pattern instance modeling [9], and it is exploited in a parallel project of our research group that implements a benchmark platform for design pattern detection tools [20, 15, 6].

6.2.1 DP representation model

To be able to work on design pattern instances we need a way to represent them in some kind of data structure. The model used by the Joiner specifies that a design pattern can be defined from the structural point of view using the roles it contains and the cardinality relationship between couple of roles.

Roles [118] are duties that can be fulfilled by program elements (e.g. types, methods) relations (e.g. inheritance, association) and collaborations in a design pattern.

A design pattern is defined as a tree whose nodes are called *Levels*; each Level has to contain at least one of the roles of the pattern and it can contain other nested Levels, recursively. In Figure 6.3 it is possible to see the tree structure of the LevelDef class (representing the level definition), and the RoleDefs it owns; finally, DpDef defines that a design pattern definition is a tree having as root one LevelDef.

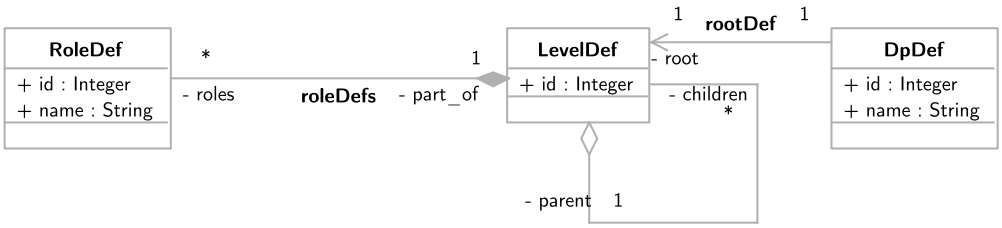


Figure 6.3: DP Definition UML class diagram

When two roles are contained in the same level, they are in a one-to-one relationship; when a role is in a nested Level, instead, it means that, for each instance of the role set in the parent level, there can be many sets of roles of the child level. The most common case is when a pattern defines that a class must extend another class. In most cases we identify a single instance of that pattern as the parent class connected with all the children classes. Following the example Joiner rule shown in Figure 6.2, if we want to specify that concrete classes referring to the same abstract class must be grouped together, we have to specify two *Levels* $L1$ and $L2$, containing respectively the Abstract and Concrete roles. Figure 6.4 shows the UML object diagram representing that rule.

Instances are modeled as in Figure 6.5; the model is simply an extension of the definition, as it models the instantiation of the concepts contained in the definition: a RoleAssociation is the realization of a RoleDef, a LevelInstance is the realization of a LevelDef, and so on. The only complex detail is the splitting of Level and LevelInstance; the explanation is that each LevelDef is instantiated as a LevelInstance when the RoleAssociations are filled, but to define a child Level we need to specify which particular parent instance it belongs to.

Applying the rule shown in Figure 6.4 to the results shown in Table 6.1 the Joiner merger result is like the one shown in Figure 6.6: there is only one instance for the LevelDef $L1$, that

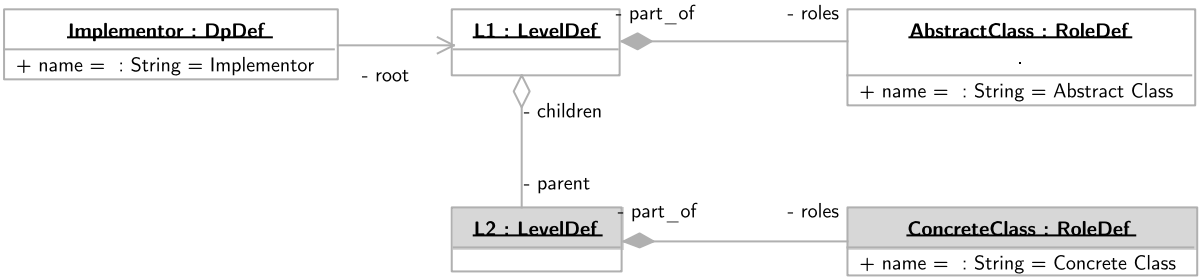


Figure 6.4: UML object diagram of the DP definition example

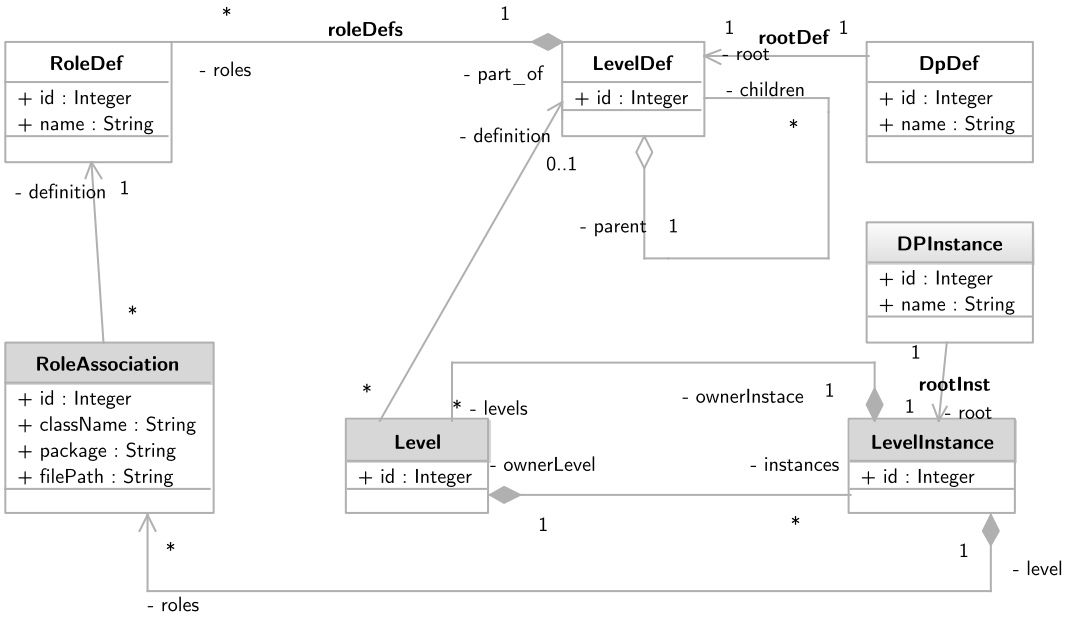


Figure 6.5: Model UML class diagram

contains only one role association to the class Component. The Composite and Leaf classes are in two different sublevels that depend on the upper one.

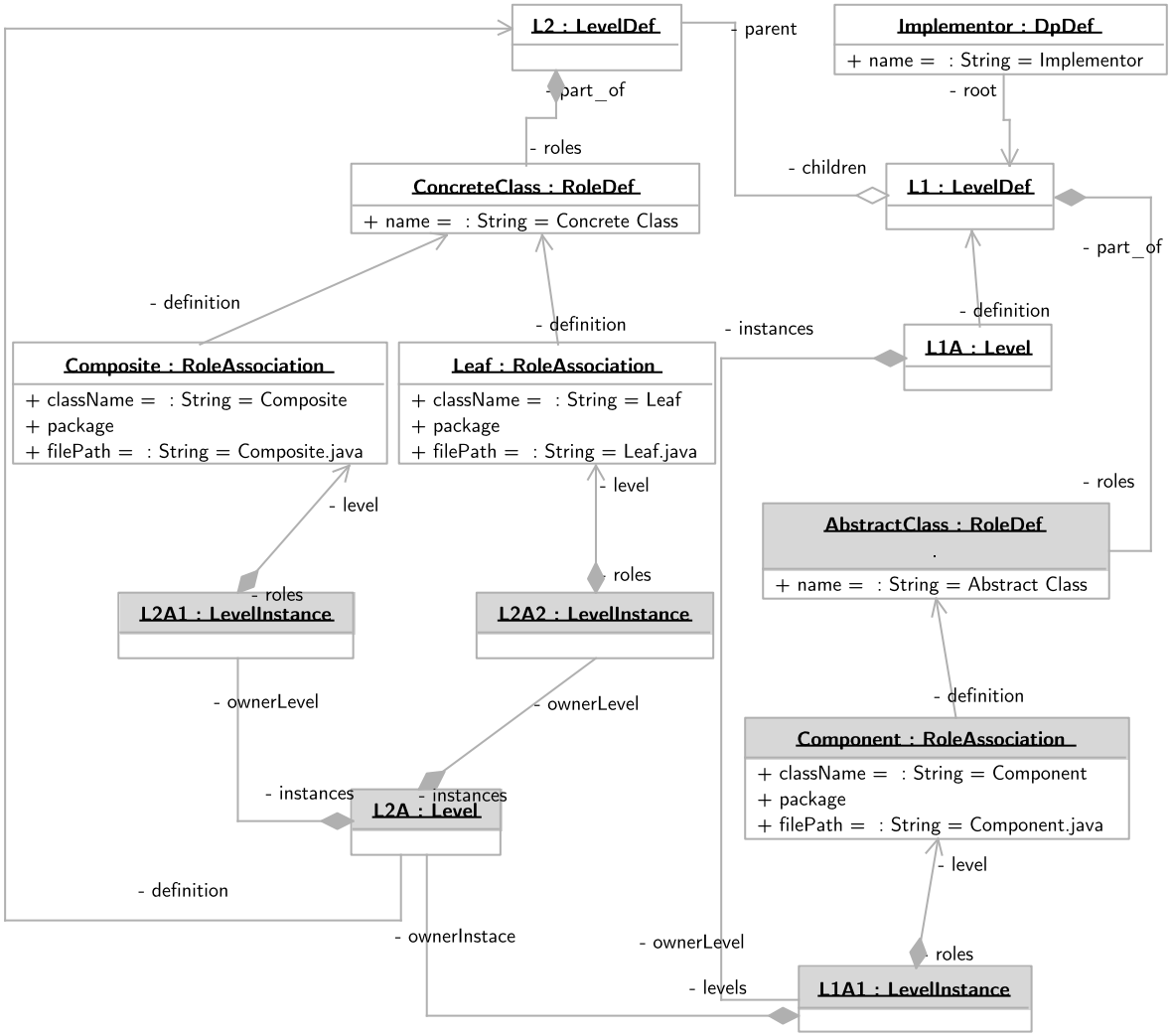


Figure 6.6: UML object diagram of the DP result example

In Section 3.2 another model for the representation of design pattern instances and definition, called DPDX, is discussed. DPDX is a wider model with a different aim, but the main differences among this model and DPDX are two. First, DPDX specifies a Program element meta-model able to represent a software system, and defines a protocol to identify the classes or methods referred by design pattern instances, because its target is the exchange of models among different tools. In MARPLE the solution of this issue is externalized to the particular implementation, i.e. there is a reference from the RoleAssociation entity to a CodeEntity in the model for the representation of the system. The second difference is more conceptual: MARPLE does not model the reason why instances are grouped in a particular way, but only how they are grouped; DPDX instead specifies the Relation, RelationAssignment and Justification entities for this purpose. The two different approaches have different goals: MARPLE tends to be minimal, without repeating information that can be reached by other means (e.g. the match rule, the micro-structures),

while DPDX includes all the possible information about the found instances to enable a better comprehension of the modeled instances.

6.2.2 Merging the mappings

All the retrieved mappings, like the ones shown in Table 6.1, have to be merged in order to build design pattern candidate instances, following the structure defined in the merge rule, like the one shown in Figure 6.4. The example of the overall process the Joiner has to complete can be resumed using Figure 6.7: the application of the rule to the system extracts many disjoint mappings that have to be merged in order to form well-structured pattern candidates.

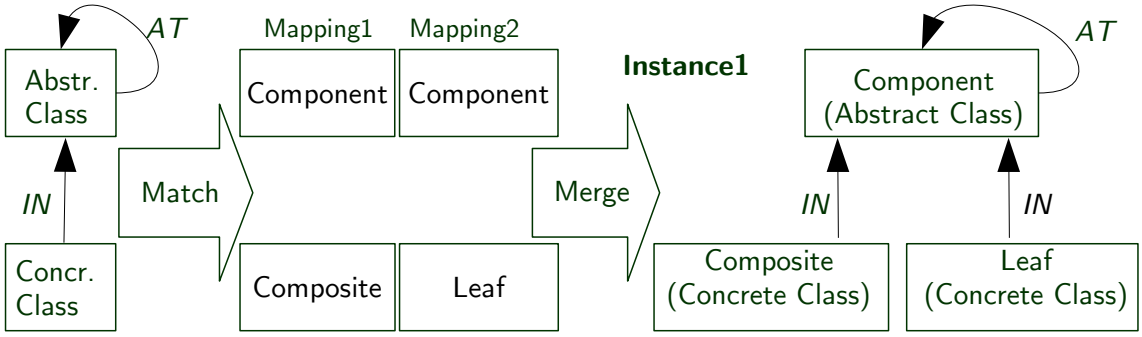


Figure 6.7: Merge process example. IN: Inheritance, AT: Abstract Type

The merge rule is interpreted as a rooted tree of *levels* defining the cardinality relationships between the instances of those levels; there are rules to follow in order to correctly interpret the rule, which implement the concepts introduced in the previous subsection:

- if a level is child of another level, it means that for each instance of the parent level there can be more than one instance of the child level; in the previous example, level $L1$ contains level $L2$, because we want to group all the subclasses (contained in $L2$) of the same abstract class (contained in $L1$);
- if two (or more) roles are in the same level, it means that the two roles uniquely identify that level; therefore, when applying the first rule, to consider two LevelInstances "equal", all the respective role associations have to be the same; it is possible to say that the roles act as *primary keys* for the levels;
- each level cannot contain duplicate roles.

The merge algorithm builds all the level instances found in a mapping, and, starting from the root, it adds those instances to the model, instantiated as a multi-root structure (the roots are design pattern instances). For each level it applies this simple rule: if the level already contains a level instance having the same role associations as the input, it recursively applies to the children levels of that level instance, otherwise it adds the input level instance to the level and then it descends like in the first case. The procedure adds a new level instance only when it is not present in the particular path the algorithm is examining, building the tree in a recursive fashion.

For example, referring to Figure 6.7, the algorithm takes the first mapping and creates the two level instances containing the two role associations; the level instances create a tree with only two role associations, and a root that is the level instance containing the Component class.

When the second mapping is added, the first level instance generated is equal to the one acting as the root of the tree, so only the second one is added to the same root. The algorithm works recursively using this behaviour.

An interesting property of the merge structure is the fact that children levels depend on their parent level instance. This implies that if a user knows, after the merge, that a certain level instance is not valid (the role associations it contains are not correct) he can prune the tree deleting that level instance and all its children. This is semantically correct because deleting a level instance means deleting all the mappings having a certain subset of role association. Finally, the correctness of the tree is guaranteed if, after the pruning, every node not being at least in a path from the root to a leaf of the tree will be deleted (it would represent a broken mapping). This behaviour will be coded (in future work) in the GUI of MARPLE, in the results inspection form.

The models instantiated by the Joiner are then inspected by the *Classifier* module, which tries to infer whether they can represent instances of design patterns or not. In Chapter 7 the details of the classification process will be explained. Next section lists the detection rules for the design patterns supported by MARPLE.

6.3 Detection rules

This section contains the detection rules, for both matching and merging, exploited for the detection of the patterns used for the experimentation of the classification approach. Detection rules have been created also for the other design patterns of the GoF book, and they are reported in Appendix A.

To fully understand the reported rules it is necessary to understand the syntax used to specify them. Match rules are specified using a SPARQL query. A SPARQL query is composed (simplifying) of a **SELECT** statement and a **WHERE** statement. In the match rules the **SELECT** statement lists the name of the roles of the design pattern, with a ? prefix. Then the **WHERE** statement is a list of constraint in conjunction, i.e. all of them must be satisfied for the rule to have a correct match. Constraints are separated by . (full stops). To create a negative constraint (to specify an illegal role assignment) the constraint must be put into a **NOT EXISTS** block. **UNION** blocks instead act as disjunctions, i.e. the whole block is matched if at least one of the constraints is matched. Finally, **OPTIONAL** blocks are not required to be matched, but it is possible to test if their variables were matched or not, allowing more sophisticated rules. The SPARQL implementation of the example rule described in Section 6.1, and represented in Figure 6.2, is the following[2]:

```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?AbstractClass ?ConcreteClass
WHERE {
?ConcreteClass BE:Inheritance ?AbstractClass.
?AbstractClass BE:AbstractType ?AbstractClass.
}
```

The merge rule is specified instead using an XML syntax. A pattern is defined by a <pattern> tag, which has a name. A <pattern> contains a list of <role> tags and <sublevel> tags. <role> tags have a name, while <sublevel> tags have no attribute. The nesting of <sublevel> tags (<pattern> is a particular <sublevel>) determines the placement of roles, following the principles described in Section 6.2. The XML representation could be sometimes difficult to read, and the UML object diagram notation (an example is can be seen in Figure 6.4) is verbose at best. For

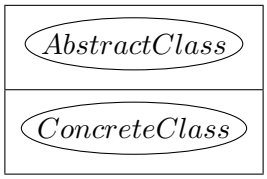
[2]recall that $(R1, R2)$ in the original example are respectively (AbstractClass, ConcreteClass) in the rule

this reason a simple graphical notation was developed to better understand merge rules. The graphical (diagram) notation uses the level nesting as boxes. A box represents a LevelDef (or a <pattern> or <sublevel> tag in the XML). When a box is *under* another box, and has the same width (or less) it means that it is a sublevel of the upper box. When two boxes are side-by-side it means that they are sibling levels. Named ellipses in the boxes represent roles. The example rule used in Section 6.2 and depicted in Figure 6.4 can be represented by the following XML rule and diagram, which are shown respectively in Listing 6.1 and Table 6.2.

Listing 6.1: Example of XML merge rule

```
<pattern name="Inheritance">
  <role name="AbstractClass" />
  <role name="ConcreteClass" />
</pattern>
```

Table 6.2: Example of merge rule diagram



The remainder of the section contains the rules for each design pattern.

6.3.1 Creational Design Patterns

Factory Method

Match rule The *Factory Method* pattern is detected looking for classes able to create instances of arbitrary (i.e. related to it or not) classes and returning them using an abstract interface. In the detection rule the direct creation of the ConcreteProduct object by the ConcreteCreator is required, because factory methods are the simplest indirection mechanism for the creation of objects. The same rule does not apply for *Abstract Factory*, for example, because it is an aggregator that provides creation of related classes in one of the ways that make it possible (i.e. direct creation, redirection to other factories, prototypes).

The match rule for the *Factory Method* pattern allows the two abstractions (the products and the creators) to be collapsed, in order to detect instances that are simpler and less structured than their theoretical description.

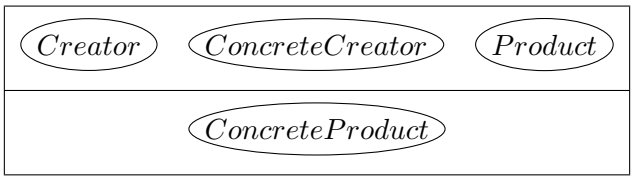
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Creator ?ConcreteCreator ?Product ?ConcreteProduct
WHERE {
?Creator BE:ProductReturns ?Product.
?ConcreteCreator BE:CreateObject ?ConcreteProduct.
{{?ConcreteCreator BE:ExtendedInheritance ?Creator.
}UNION{?ConcreteCreator BE:SameClass ?Creator.}}.

{{?ConcreteProduct BE:ExtendedInheritance ?Product.}
UNION{?ConcreteProduct BE:SameClass ?Product.}}.
```

```
NOT EXISTS{?Product BE:ExtendedInheritance ?Creator.}.
NOT EXISTS{?Product BE:ExtendedInheritance ?ConcreteCreator.}.

OPTIONAL{
    ?Creator BE:ExtendedInheritance ?y.
    ?y BE:ProductReturns ?Product.
}FILTER(!bound(?y)).
}
```

Merge rule The focus of the merge rule for the *Factory Method* pattern is the on fact that this pattern describes a way of giving the responsibility of the creation of a class instance to a method different from the constructor. Each implemented method is an instance of the pattern, and it belongs to the ConcreteCreator role. There exists only one Creator (the superclass) and Product (the return type) for each ConcreteCreator, so all the three of them are in the same level. Many ConcreteProducts may exist for the same pattern instance.

```
<pattern name="FactoryMethod">
  <role name="Creator" />
  <role name="ConcreteCreator" />
  <role name="Product" />
  <sublevel>
    <role name="ConcreteProduct" />
  </sublevel>
</pattern>
```

Merge rule diagram



Evaluation During the manual evaluation, one of the strongest criteria used is that a *Factory Method* must have a method which is in charge of creating objects and that decides if creating them or not. The criterion includes static factories (for the same class or not). The only exceptions to the criterion are factories using prototyping constructors or similars; even if the clone (or equivalent) method is a factory method for the same class, it is a very specific case and it belongs to the *Prototype* design pattern.

Discussion Sometimes the documentation of software systems describes some methods as a *Factory Method* for something, but the ConcreteCreator simply delegates the creation to some other object; in this case it is simply an *Adapter* for some other *Factory Method*, and should be reported as such. An extreme example of this situation is the instantiation of objects using the reflection API, e.g. Class.newInstance(), in Java; those mechanisms are *Factory Method*s themselves, allowing the instantiation of an object without knowing its real type.

It is possible to imagine some kind of "pattern arithmetic" that would allow the composition of patterns, and to state that, for example, an "*Adapter* for a *Factory Method* is a *Factory Method* itself, transitively", but this needs future investigations.

The *Factory Method* pattern rule puts the ConcreteCreator in the root level. This has been done to underline the lightweight nature of the *Factory Method*. Despite this, after the evaluation

of many instances, having the ConcreteCreator in its own sublevel would be another good option, producing a lot less instances and grouping the different ConcreteCreators and Products related to the same Creator.

Singleton

Singleton is a simple pattern, with only one role. The detection rule is therefore not particularly complicated.

Match rule Next the match rule for the *Singleton* pattern, defined in SPARQL. Basically there are four variant managed, that use different mechanisms to ensure that the class exists in a single instance.

```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Singleton
WHERE {
{?Singleton
    BE:CreateObject ?Singleton ;
    BE:PrivateStaticReference ?Singleton ;
    BE:PrivateConstructor ?Singleton . }
UNION {?Singleton
    BE:CreateObject ?Singleton ;
    BE:ProtectedStaticReference ?Singleton ;
    BE:PrivateConstructor ?Singleton . }
UNION {?Singleton
    BE:CreateObject ?Singleton ;
    BE:OtherStaticReference ?Singleton ;
    BE:PrivateConstructor ?Singleton . }
UNION {?Singleton
    BE:CreateObject ?Singleton ;
    BE:PrivateStaticReference ?Singleton ;
    BE:ControlledException ?Singleton . }
UNION {?Singleton
    BE:StaticFlag ?Singleton ;
    BE:ControlledException ?Singleton . }
UNION {?Singleton
    BE:CreateObject ?Singleton ;
    BE:StaticFlag ?Singleton ;
    BE:ControlledInstantiation ?Singleton . }
}
```

Merge rule Next the merge rule for the *Singleton* pattern, given in XML. There is only one role, so there is also only one level.

```
<pattern name="Singleton">
  <role name="Singleton"/>
</pattern>
```

Merge rule diagram

6.3.2 Structural Design Patterns

Adapter

Match rule The particular constraints in the *Adapter* match rule are that Target and Adaptee must not know each other, and the Adapter has not to be an empty class. The Adapter class must extend a class or interface Target, implementing at least one method. One call to an Adaptee object must exist, coming from at least one implemented method, considering also other methods (**public** or **private**) defined in the same class, transitively; invocations on objects coming from parameters of the overridden method are not considered, including objects or expression directly derivable from the parameters using chained method calls. Passing parameters to external methods is not considered as a method call, but as a delegation, even if the target operation, behaviour or return value is obvious/known.

Method calls originated from other methods and constructors are clearly ignored. Adaptees can be instantiated on the fly, they can be *Singleton*s, and they can be generically retrieved from a static method (also transitively).

It is improbable for the Adaptee to be a superclass of Target; it means they come from the same library or project, and it would seem more of a *Decorator*.

```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Target ?Adapter ?Adaptee
WHERE {
{ # Class Adapter clause
    ?Adapter
        BE:ExtendedInheritance ?Target ;
        BE:ExtendedInheritance ?Adaptee.
        ?Adaptee BE:SameClass ?Adaptee.

    {{?Adapter BE:RevertMethod ?Adaptee}
        UNION{?Adapter BE:ExtendMethod ?Adaptee}
        UNION{?Adapter BE:Conglomeration ?Adaptee}}.

    {{?Target BE:Interface ?Target.}
    UNION {?Adaptee BE:Interface ?Adaptee.}}.
}
UNION
{ # Object Adapter clause

?Adapter BE:ExtendedInheritance ?Target.
?Adaptee BE:SameClass ?Adaptee.

{{?Adapter BE:Delegate ?Adaptee.} UNION {?Adapter BE:Redirect ?Adaptee.}}.

{
    {?Adapter BE:ProtectedInstanceReference ?Adaptee.}
UNION{?Adapter BE:PrivateInstanceReference ?Adaptee.}
UNION{?Adapter BE:OtherInstanceReference ?Adaptee.}
UNION{?Adapter BE:ProtectedStaticReference ?Adaptee.}
UNION{?Adapter BE:PrivateStaticReference ?Adaptee.}
UNION{?Adapter BE:OtherStaticReference ?Adaptee.}
UNION{
    {{?Adapter BE:Delegate ?f.}
     UNION{?Adapter BE:Redirect ?f.}
     UNION{?Adapter BE:DelegateInFamily ?f.}
```

```
      UNION{?Adapter BE:RedirectInFamily ?f.}
      UNION{?Adapter BE:DelegateInLimitedFamily ?f.}
      UNION{?Adapter BE:RedirectInLimitedFamily ?f.}}.
     ?f BE:ProductReturns ?Adaptee.
}
UNION{
     ?Adapter BE:CreateObject ?Adaptee.
}
}.

} # End of Object adapter clause

# Target and its supertypes must not "know" the Adaptee
OPTIONAL {
     {{?Target BE:ExtendedInheritance ?x.}UNION{?Target BE:SameClass ?x.}}.

     {{?x BE:Delegate ?Adaptee.}
     UNION {?x BE:Redirect ?Adaptee.}
     UNION {?x BE:Conglomeration ?Adaptee.}
     UNION {?x BE:Recursion ?Adaptee.}
     UNION {?x BE:RevertMethod ?Adaptee.}
     UNION {?x BE:ExtendMethod ?Adaptee.}
     UNION {?x BE:DelegatedConglomeration ?Adaptee.}
     UNION {?x BE:RedirectRecursion ?Adaptee.}
     UNION {?x BE:DelegateInFamily ?Adaptee.}
     UNION {?x BE:RedirectInFamily ?Adaptee.}
     UNION {?x BE:DelegateInLimitedFamily ?Adaptee.}
     UNION {?x BE:RedirectInLimitedFamily ?Adaptee.}
     UNION {?x BE:ReceivesParameter ?Adaptee.}
     UNION {?x BE:ProtectedInstanceReference ?Adaptee.}
     UNION {?x BE:PrivateInstanceReference ?Adaptee.}
     UNION {?x BE:OtherInstanceReference ?Adaptee.}}.
}
FILTER(!bound(?x)).


# Target must not "know" Adaptee
NOT EXISTS {?Target BE:SameClass  ?Adaptee .}.
NOT EXISTS {?Target BE:ExtendedInheritance  ?Adaptee .}.


NOT EXISTS {?Adaptee BE:SameClass  ?Target .}.
NOT EXISTS {?Adaptee BE:ExtendedInheritance  ?Target .}.
NOT EXISTS {?Adaptee BE:Delegate  ?Target .}.
NOT EXISTS {?Adaptee BE:Redirect  ?Target .}.
NOT EXISTS {?Adaptee BE:Conglomeration  ?Target .}.
NOT EXISTS {?Adaptee BE:Recursion  ?Target .}.
NOT EXISTS {?Adaptee BE:RevertMethod  ?Target .}.
NOT EXISTS {?Adaptee BE:ExtendMethod  ?Target .}.
NOT EXISTS {?Adaptee BE:DelegatedConglomeration  ?Target .}.
NOT EXISTS {?Adaptee BE:RedirectRecursion  ?Target .}.
NOT EXISTS {?Adaptee BE:DelegateInFamily  ?Target .}.
NOT EXISTS {?Adaptee BE:RedirectInFamily  ?Target .}.
NOT EXISTS {?Adaptee BE:DelegateInLimitedFamily  ?Target .}.
NOT EXISTS {?Adaptee BE:RedirectInLimitedFamily  ?Target .}.
```

```
NOT EXISTS {?Adapter BE:Joiner ?Adapter .}.

# Select the more specialized Target in the Adapter supertypes
OPTIONAL {
    ?y BE:ExtendedInheritance ?Target.
    ?Adapter BE:ExtendedInheritance ?y.
}
FILTER(!bound(?y)).
}
```

Merge rule There is basically no grouping for the adapter pattern. *Adapter* is an "opportunistic" pattern which aim is to transform an interface into another one in order to reuse some existing code.

```
<pattern name="Adapter">
  <role name="Target" />
  <role name="Adapter" />
  <role name="Adaptee" />
</pattern>
```

Merge rule diagram

Discussion Object-oriented programming consists of defining classes that are interconnected and delegate each other the work which is their responsibility. The arrangement of responsibility of classes is one of the major causes of good or bad design quality. The *Adapter* pattern brings the concept that some objects are responsible to adapt a Target interface/protocol to be implemented using some existing facility, which is in turn another class. The pattern is a way of describing the fact that some objects are mainly glue code, and some others are defined in order to be (if necessary) implemented by other classes for their needs. Unfortunately the definition of *Adapter* is not able to express how much adapting must be done by the Adapter, how many Adaptees can be used, if the same Adapter can adapt more than one Target, and so on. The main concern of the *Adapter* design pattern definition is to explain a concept that is useful and widely used. The ambiguity of the definition causes problems, also in relation to other design patterns from the same catalogue, like the *Strategy* design pattern. In fact, *Strategy* is the most abstract pattern in this sense. An Adapter can coincide with a ConcreteStrategy, if the strategy is to rely on other objects that are already able to execute some tasks, but also the Strategy can be the Adaptee and the Adapter its Context, if the concrete type of the Adaptee is unknown to the Adapter. As an overall conclusion, object-oriented programming is dominated by delegation of responsibility and behaviour. In the context of software analysis and assessment, where the intents and purposes of the developers (of the software we are analyzing) are difficult to understand (in some case) also to expert software engineers and programmers, does it really make sense to try to automatically distinguish between delegation for *Adapters*, *Strategies*, *Commands* and so on? Would it be possible to define some measure of this intrinsic characteristic of the object-oriented paradigm that is able to help people at least to understand quickly how software was organized? Another perspective could be the one of dependency analysis. In that discipline, software is seen as a graph of nodes, representing packages, classes, methods or each of them (for different purposes); the nodes are connected by edges that represent the dependencies between the nodes, creating

a graph. A dependency is usually one (or each of) of this kind of constructs: method calls, field declarations, passed parameters, etc... More generally, in the context of static analysis, a dependency [32] is every kind of explicit static usage of a different code entity, that explicitly ties (at least) its external interface to the type. This fact has consequences on the maintenance effort, because every time the interface (or worse, the behaviour) of a dependency is changed, all its dependents need to be checked.

The dependency analysis perspective can help to define a method for analyzing software in a way similar (or with similar aims at least, when talking about reverse engineering) to *Adapter* or *Strategy* design patterns discovery, but with a more formalized definition. Both of these patterns define a different motivation of delegating a task to another object, possibly without knowing about its implementation. Given the previous discussion, and the many variants defined in the GoF book [74] or in real systems, from a detection point of view most of the times a delegation exists we are in the presence of one these patterns; this extreme diffusion lowers their relevance in the knowledge of the system: it would be more useful to measure, for example, *how much* adaptation is present in a class.

We need to define a strategy to achieve this goal. Some analysis techniques can be useful: using call graph analysis it is possible to extract all the method call paths from a given starting point, and using data flow analysis it is possible to track all the possible assignment of a given variable in a certain point. Each of these two techniques can suffer from performance issues when applied to an entire system, but not when applied to a single class.

Imagine to track all the call paths starting from a method implemented in a class, without following method calls to different types (and direct or indirect recursions), simply tagging external method calls for later use. Each of the external method calls is sent to a certain static type, resolved at compile time (in assessment tasks polymorphism is less relevant than in other disciplines, and overcomplicated to analyze for the purpose). Using data flow analysis, it is possible to know the types of the objects referred by the variables that received the external method calls; the retrieved types are limited to the ones which are explicitly resolvable in the context of the class. Tracking all this information (the feasibility should be clear), it is possible to get this particular kind of dependency analysis: which types (and which methods of them) are used by a method to execute its task?

Following the criteria reported above, it is possible to take the new dependency analysis and customize it a little: for each overridden method in a class, which types/method does it use to execute its task, without considering those retrieved from the method parameters? In the end the result will be a list of types/methods that the type decided to rely on when trying to achieve its goals. It is possible to make different considerations over these data, like the ones described in next items.

- The simplest one is reporting the types used to implement the method, grouped by the overridden type: this gives the idea of which classes were used by the class to be able to implement (or properly override) the behaviour of each particular supertype. From the *Adapter* design pattern detection point of view this means that, considering every class a potential Adapter, we point, for each of its supertypes (the potential Targets), to all its Adaptees. If the *Adapter* design pattern is intended in a more subtle way, it would be possible to remove from the calculation the method calls sent to instances which are derived from other ones (remember we already removed everything coming from the root method's parameter), and therefore we will end up with the "root" Adaptee types, the ones that are able to provide everything else is used to implement the Adapter's behavior. Another variant could be to associate to each type the number of methods or the number of calls used in the overall paths. This would be a possible measure of how much a type is an Adaptee for an Adapter.

- The second example is less related to design pattern detection. If we take the grouping made in the first example, considering the analysis of a whole system, and group another time, over the Adapter's type, we end up with the Adaptees used to implement each Target (it is possible to group another time collapsing Targets that inherit each other). This kind of analysis can describe different situations where, for example (the opposites): a Target is always implemented by the same (or a small number of) type(s) or it is implemented by a very high number of types, or if some type is never/very used as Adaptee. This "proportion" analysis is typical of dependency analysis, but is deeper than simply stating how many times a method/type has been used and from which other type: it measures the quantity of object composition used in the system, and the analysis of limit cases (or outliers) it could be useful to find, e.g., design patterns, code smells and anti-patterns.

- A possible variant of this particular dependency analysis can be the application of a simple filter, which is used to decide to consider (or not) the types coming from particular libraries (or packages, in the example of Java). The filter could be useful because, in a first approximation, it is very probable that standard libraries (or other utility libraries and frameworks) will be pervasive in a system, especially as Adaptees, in this particular parallel example; most classes will use Strings and Collections , for example, and in most cases this information is not relevant (but if the analyzed project is some kind of utility over the standard library it will be useful to keep everything).

The one described above is only an example of how it is possible to use a dependency analysis approach to make a discovery that is reliable, because it is not approximated, that is feasible and that can be used for different purposes, potentially discovering patterns or flaws. Using the same approach, but with a slightly different analysis, it would be possible to track all the fields and variables that are used only for storage purposes, and, measuring the number of paths, it would be possible to measure the amount of, e.g., code used for delegation, "glue code" [29], and "data store" code.

In this hypothetical context what is the definition of an *Adapter* design pattern? A class that forwards calls only to another "root", one for each of its direct supertypes? And a *Strategy*? A class that delegates to something else? It could be possible to define how much *Adapter* and how much *Strategy* there is in each class or method, and it would be a more precise information in an assessment task. Naturally this analysis useful only for some patterns (but it is not a design pattern-oriented analysis). For example the *Template Method* design pattern does not need any analysis of this kind, because it is based on another kind of object-oriented construct, i.e. inheritance. Curiously, the *Template Method* is one of the patterns where it is possible to define a formal definition that is able to extract it with no (or very few) errors. The motivation is simply the fact that the programming language (in Java, at least) supports directly the pattern without the need of particular conventions or arrangements.

Composite

Match rule The *Composite* design pattern is made by three roles Component, Composite and Leaf. The essence of the design pattern is that it lets the client treat a Composite like an indistinct Component, without knowing the type and the number of receivers of the Component's operations. Moreover, this goal is achieved using a tree structure, where Composites are non-terminal nodes, able to have children, and Leaves are terminal nodes. The rule focuses on the inheritance tree and the redirection of method calls from the Composite to the Component, and the absence of redirection in the Leaf.

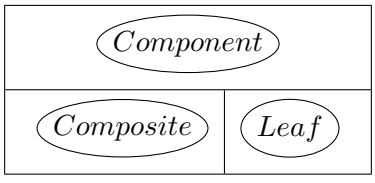
PREFIX BE: <http://essere.disco.unimib.it/marple/BEs#>

```
SELECT  ?Component ?Composite ?Leaf
WHERE {
?Composite   BE:ExtendedInheritance ?Component;
             BE:RedirectInFamily ?Component.

?Leaf    BE:ExtendedInheritance ?Component;
             BE:SameClass ?Leaf.

NOT EXISTS{?Leaf BE:ExtendedInheritance ?Composite.}.
NOT EXISTS{?Leaf BE:RedirectInFamily ?Component.}.
NOT EXISTS{?Leaf BE:AbstractClass ?Leaf.}.
NOT EXISTS{?Leaf BE:Interface ?Leaf.}.

# Select the more specialized Component
OPTIONAL {
    ?y BE:ExtendedInheritance ?Component.

    #Repeating rules
    ?Composite   BE:ExtendedInheritance ?y;
                 BE:RedirectInFamily ?y.
    ?Leaf    BE:ExtendedInheritance ?y.
}
FILTER(!bound(?y)).
}
```

Merge rule The merge rule simply describes the fact that a *Composite* design pattern is identified by its abstraction, i.e. the Component, which has a collection of realizations that are subdivided in Composites and Leaves.

```
<pattern name="Composite">
  <role name="Component" />
  <sublevel>
    <role name="Composite" />
  </sublevel>
  <sublevel>
    <role name="Leaf" />
  </sublevel>
```

Merge rule diagram



Evaluation In my evaluation I consider a *Composite* pattern as correct when both tree and delegation characteristics are present. If only the tree structure is present, e.g. because all operations are delegated to Visitor roles (see for example the SimpleNode class and its subclasses in PMD), the structure is not considered a *Composite* pattern. If I did otherwise, every class having a reference to a collection of superclasses would be considered to belong to a *Composite* pattern.

Discussion The detection of (typed) aggregation or collection is a tricky and delicate argument. In this thesis I decided not to deepen the analysis of this aspect to focus on other ones, but in the future it will be one of the important problems to solve. The detection rule for the *Composite* pattern does not enforce the aggregation of Components in the Composite, but relies on the redirection of methods from the Composite to children Components (in the rule), and the presence of calls from Composite to Components in cycles (in the classifier).

From the design/implementation (not only the intent) point of view, a *Composite* design pattern is simply the generalization of a *Decorator* pattern to many children objects, instead of a single one. The *Decorator* detection rule can be more precise in the definition of the pattern structure because a single reference can be better detected than a collection or an aggregation.

An interesting distinction in the *Composite* design pattern roles is the one between Composite and Leaf. Composite is an intermediate node able to host children Components, which can be mixed Composites or Leaves. Most of the times this distinction is enough. In complex hierarchies there are also intermediate classes (maybe abstract) that are used for example to put together common functionalities, as for example to be able to host children, without using them; this is the case where different Composites are available, but they share the same children management implementation (but not the same composition of children). In a detection rule (that looks for a cyclic composition) such classes can be seen as Leaves, but they are not. And they are not Composites, so it is better to leave them out of the pattern (inspecting the pattern they will come up anyway). In the detection rule this is achieved removing from the Leaves abstract classes and interfaces.

Decorator

Match rule The match rule for the *Decorator* pattern allows the Decorator and the ConcreteDecorator roles to be assigned to the same class, and does not allow the ConcreteDecorator to be abstract. The other issues addressed by the rule are the inheritance tree and the separation of ConcreteComponents and Decorators.

Decorator is, from the implementation point of view, a "degenerate" *Composite* having only one child. The rule defines that a *Decorator* enforces constraints similar to the ones used in the *Composite* rule, but has different roles number and organization.

```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Component ?Decorator ?ConcreteDecorator ?ConcreteComponent
WHERE {
?Decorator BE:ExtendedInheritance ?Component.
?ConcreteComponent BE:ExtendedInheritance ?Component.

{{?Decorator BE:RedirectInFamily ?Component.}
UNION
{?Decorator BE:DelegateInFamily ?Component.}}.

{
    {?ConcreteDecorator BE:ExtendedInheritance ?Decorator.
    {{?ConcreteDecorator BE:ExtendMethod ?Decorator.}
    UNION{?ConcreteDecorator BE:RevertMethod ?Decorator.}
    UNION{?ConcreteDecorator BE:RedirectInFamily ?Component.}
    UNION{?ConcreteDecorator BE:DelegateInFamily ?Component.}}.
}UNION{
    ?ConcreteDecorator BE:SameClass ?Decorator.
}
}.
```

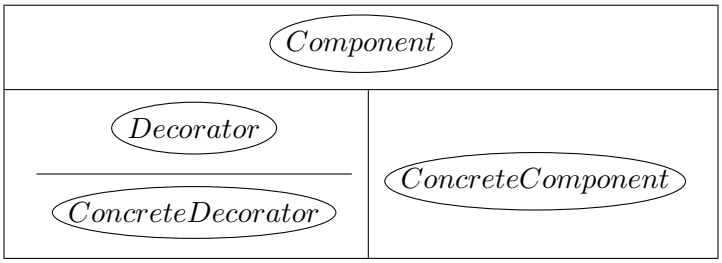
```
NOT EXISTS {?ConcreteDecorator BE:AbstractType ?ConcreteDecorator.}.

NOT EXISTS {?ConcreteComponent BE:ExtendedInheritance ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:SameClass ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:SameClass ?ConcreteDecorator.}.

NOT EXISTS{?ConcreteComponent BE:RedirectInFamily ?Component.}.
NOT EXISTS{?ConcreteComponent BE:DelegateInFamily ?Component.}.

NOT EXISTS {?ConcreteComponent BE:Delegate ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:Redirect ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:Conglomeration ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:Recursion ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:RevertMethod ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:ExtendMethod ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:DelegatedConglomeration ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:RedirectRecursion ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:DelegateInFamily ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:RedirectInFamily ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:DelegateInLimitedFamily ?Decorator.}.
NOT EXISTS {?ConcreteComponent BE:RedirectInLimitedFamily ?Decorator.}.

# Report the more abstract Decorator if more than one are available
OPTIONAL{
    ?Decorator BE:ExtendedInheritance ?x.
    ?x BE:ExtendedInheritance ?Component.
    {{?x BE:RedirectInFamily ?Component.}
    UNION
    {?x BE:DelegateInFamily ?Component.}}.

    {{?ConcreteDecorator BE:ExtendMethod ?x.}
    UNION{?ConcreteDecorator BE:RevertMethod ?x.}
    UNION{?ConcreteDecorator BE:RedirectInFamily ?Component.}
    UNION{?ConcreteDecorator BE:DelegateInFamily ?Component.}}.

    NOT EXISTS {?ConcreteComponent BE:ExtendedInheritance ?x.}.
    NOT EXISTS {?ConcreteComponent BE:SameClass ?x.}.
    NOT EXISTS {?ConcreteComponent BE:Delegate ?x.}.
    NOT EXISTS {?ConcreteComponent BE:Redirect ?x.}.
    NOT EXISTS {?ConcreteComponent BE:Conglomeration ?x.}.
    NOT EXISTS {?ConcreteComponent BE:Recursion ?x.}.
    NOT EXISTS {?ConcreteComponent BE:RevertMethod ?x.}.
    NOT EXISTS {?ConcreteComponent BE:ExtendMethod ?x.}.
    NOT EXISTS {?ConcreteComponent BE:DelegatedConglomeration ?x.}.
    NOT EXISTS {?ConcreteComponent BE:RedirectRecursion ?x.}.
    NOT EXISTS {?ConcreteComponent BE:DelegateInFamily ?x.}.
    NOT EXISTS {?ConcreteComponent BE:RedirectInFamily ?x.}.
    NOT EXISTS {?ConcreteComponent BE:DelegateInLimitedFamily ?x.}.
    NOT EXISTS {?ConcreteComponent BE:RedirectInLimitedFamily ?x.}.


}FILTER(!bound(?x)).

# Report the more concrete Component if more than one are available
OPTIONAL{
```

```
    ?y BE:ExtendedInheritance ?Component.

    ?Decorator BE:ExtendedInheritance  ?y.
    ?ConcreteComponent BE:ExtendedInheritance  ?y.

    {{?Decorator BE:RedirectInFamily ?y.}
    UNION
    {?Decorator BE:DelegateInFamily  ?y.}}.
    {
        {?ConcreteDecorator BE:ExtendedInheritance ?Decorator.
        {{?ConcreteDecorator BE:ExtendMethod ?Decorator.}
        UNION{?ConcreteDecorator BE:RevertMethod ?Decorator.}
        UNION{?ConcreteDecorator BE:RedirectInFamily ?y.}
        UNION{?ConcreteDecorator BE:DelegateInFamily  ?y.}}.
    }UNION{
        ?ConcreteDecorator BE:SameClass ?Decorator.
    }
    }.

    NOT EXISTS{?ConcreteComponent BE:RedirectInFamily ?y.}.
    NOT EXISTS{?ConcreteComponent BE:DelegateInFamily  ?y.}.
}FILTER(!bound(?y)).
}
```

Merge rule The choice done in the merge rule is to put together different Decorators for the same Component, focusing the merging on the detection of the "extension by composition" feature of the Decorator pattern. Keeping the Component at the root level lets the user to see all the possible ways of decorating it. The choice does not create side-effects because a Decorator cannot decorate other classes than its Component.

```
<pattern name="Decorator">
  <role name="Component" />
  <sublevel>
    <role name="Decorator" />
    <sublevel>
      <role name="ConcreteDecorator" />
    </sublevel>
  </sublevel>
  <sublevel>
    <role name="ConcreteComponent" />
  </sublevel>
</pattern>
```

Merge rule diagram



Evaluation Sometimes Composites keep a reference to the parent Component. When some method call is forwarded to the parent, the Composite could be seen as a Decorator for the parent component. As this is a known option of the *Composite* pattern, these instances have been evaluated as incorrect.

A problem for the detection is not having the information of which method is calling another one: it can lead to lots of false positives, e.g. classes having a static method that build a new instance of the class and calls its inherited method will be considered a DelegateInFamily EDP. This can create a false *Decorator* report. Most of these instances will should be filtered by the classifier if, for example, the Decorator class has no direct reference to the Component (which is not needed but is true in the majority of correct cases).

Discussion The *Decorator* pattern can also be considered "object inheritance", because one class extends the interface of another one, but can vary the implementation of the superclass. In some contexts this behaviour is called *consulting* (e.g. definition and references are reported by Kniesel [116] in the context of the Darwin project). Considering all the variants specified in the definition, it is simple to see that there is no (or very little difference) among the *Decorator*, *Proxy* and *Chain of Responsibility* design patterns. Most of the differences explained in the book [74] are about the intent and not the structure or even the behaviour. The classical forms of these patterns have slight differences because *usually*:

- Decorator receives the Component during its initialization;
- Proxy can be able to instantiate its Subject;
- the Handler of a *Chain of Responsibility* knows how to reach its successor and exposes it to its subclasses.

Despite these differences, it is possible to find variants of each of these patterns that can be seen exactly as one of the other two. The motivation is that these patterns are different ways of decoupling the sender of a message from the concrete receiver(s). In these sense the *Composite* pattern can be seen as another variant of these three patterns. Each of these patterns let the designer to define (or reuse) an interface and leave the choice (or combination) of the implementation to the run-time. And each one of these patterns has the impact on the overall design of preventing the explosion of the number of classes of the system, in the case some feature must be added to all the classes implementing some interface.

For these reasons one could argue that the separation of these patterns (in two different categories) is good for educational purposes, but not from the maintenance and assessment point of view, because the mechanism employed in these patterns are always the same.

Taking another point of view a *Decorator* is like an *Adapter*, where the Adaptee is the Component. The difference is that in the *Decorator* the Target and the Adaptee coincide. Here a quote from the book: *"Adapter: A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface."*

Following this line it is possible to apply the same analysis described in the *Adapter*'s discussion in order to activate another form of detection of the *Decorator* (but also *Proxy* and *Chain of Responsibility*) pattern, more oriented to the identification of which methods are decorated, how much, and if the same Decorator is decorating two different components (for different interfaces, otherwise it would be a *Composite* pattern).

6.4 Conclusion

The Joiner module was completely developed and enhanced from its early development stages. It is now possible to reliably match any SPARQL query against a model of the system represented as a graph where the nodes are code entities (currently only types) and the edges are the micro-structures collected by the Micro Structure Detector. The output of the Joiner module is taken as input from the Classifier module, which is introduced in the next chapter.

Chapter 7

Classifier: ranking pattern candidates

The *Classifier* module takes all the candidate design pattern instances identified by the Joiner and evaluates their similarity to the searched design pattern, to be able to rank them. Figure 7.1 shows an example of the classification process. This chapter introduces the approach, explains the motivation of the choices made during its design, the found solutions and the enhancements applied during its development and experimentation. Finally, the interaction of MARPLE-DPD with the user is described, reporting some example screenshots of the user interface.

7.1 Introduction to the learning approach

The rationale behind the approach is that the Joiner module finds all the instances matching an exact rule; this rule is written trying to keep it very general, so the matching tends to produce a large number of instances (having very high recall), but many of them are false positives (so the precision is low). However, the returned instances also carry a lot of information that the recognition rule does not use: all the micro-structures found in the classes belonging to the instances. A classifier has the possibility to choose the right instances among the ones extracted by the Joiner module, exploiting the micro-structures not used by the Joiner. Without the support of the Joiner, a simpler approach to the problem would be to submit every class, or worse, every possible group of classes of the system to the classification algorithm; such an approach would be very resource-demanding, and would require a lot of correct pattern instances to training the classifiers.

The approach described here has the benefit of submitting a smaller number of candidates to the classifier, but with a higher percentage of true instances. In this way, the performance estimation is more accurate and the dataset contains a smaller quantity of noise, because it is focused only on a specific subset of the domain. The choice of using a classification process after an exact matching is one of the principal features characterizing the approach; the same rationale is used also, for example, by Ferenc et al. [69], while other authors tried the full-classification approach, e.g. Guéhéneuc et al. [86]. A discussion of machine learning approaches for design pattern detection can be found in Section 2.4.

In some cases the same pattern can have different well known structural alternatives; these cases are handled in my approach in two ways:

- If the different structural alternatives have the same set of roles, and they are organized in the same structure, the alternatives are handled by a single rule composed of the union of different constraints; for example, an *Adapter* design pattern is known in two major

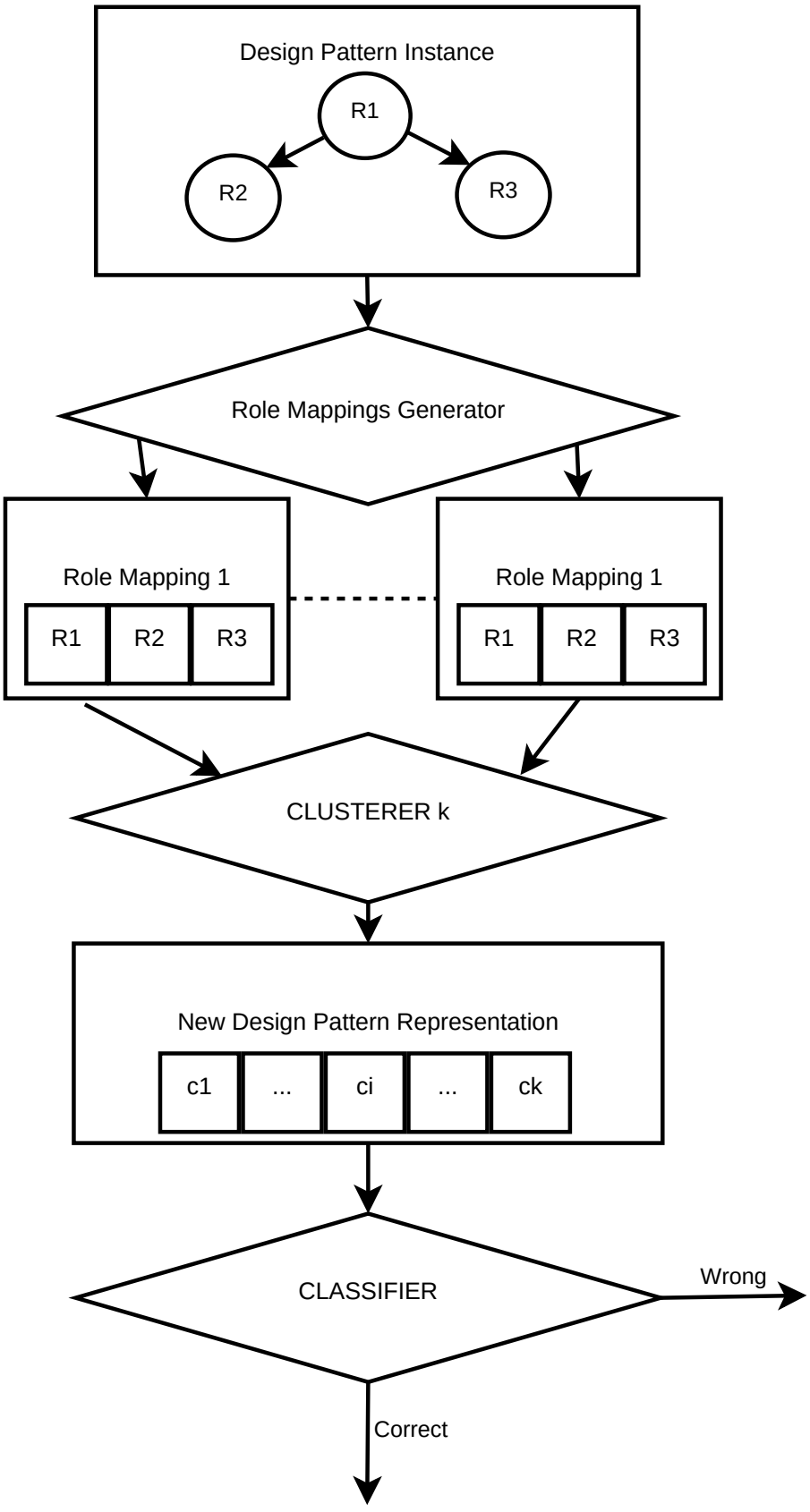


Figure 7.1: Classification process

variants: object *Adapter* and class *Adapter*. The two variants are composed of the same roles, i.e. Target, Adaptee and Adapter, and with the same organization. So the two variants are handled in the same detection rule.

- If the set of roles of the different alternatives are different, or organized in a different structure, the alternatives are handled like different design patterns, and computed independently. Alternatives of this kind are for example derived from different interpretations of the pattern or from some design decisions, e.g. to include or exclude the Client role from the pattern definition.

The integrated handling of structurally different alternatives of a design pattern will be handled by at the user interface level, adding some modeling effort to track the different variants. This is out of the focus of the thesis and it is considered future work.

Through the current approach, depicted in Figure 7.1, I generate every possible valid role mapping $\{(R_1, C_1), (R_2, C_2), \ldots, (R_n, C_n)\}$ for each pattern instance, where each C_i is the class that is supposed to play the role R_i inside the pattern. These mappings are all of a fixed size (one element for each pattern role) and each class has a fixed number of features, where the features are the micro-structures retrieved in the class. In this way each mapping can be represented as a vector of features whose length is given by $(num\_features * num\_roles)$. These vectors are grouped by a clustering [47, 104] algorithm, producing k clusters; each pattern instance is represented as a k-long vector, having in each position i the absence/presence of the i-th mapping. Since we know that an instance is a design pattern (or not) directly from the training set, we can label each vector with the class attribute and use the resulting dataset for the training of a supervised classifier. The early version of the approach was published in an international workshop [10], and then published in an international journal [19] in 2011.

7.2 Motivation

The design of a machine learning oriented solution to a classification problem leads to the modeling of the input and output format of the algorithms to employ. The typical expectation of a learning algorithm is to receive a *dataset* as input, i.e. a list of vectors (a matrix) where each vector represents one of the subjects of the classification. The representation is achieved by the usage of *features*: the i-th cell of the vector represents the value of the i-th feature used to describe the subject. During the learning of a supervised classifier, a special feature is expected that represents the *class* value of the subject. The class is the target of the classification problem, i.e. in a design pattern detection problem it can be a boolean value telling if the subject (a pattern candidate) represents a correct pattern instance. Table 7.1 shows an example of an input dataset for a design pattern classification problem, composed of n features, from F_1 to F_n, a Class attribute allowing two values (correct, incorrect) and representing two instances, named Instance 1 and Instance 2; the one represented in Table 7.1 is therefore the *target* format to make possible the exploitation of supervised classification algorithms.

Table 7.1: Example of the typical input format of a supervised classification algorithm

	F_1	F_2	...	F_i	...	F_{n-1}	F_n	Class
Instance 1			...		...			correct
Instance 2			...		...			incorrect

A deeper look to the target format makes the modeling problem clear: how can we represent a pattern instance as a feature vector, knowing from Subsection 6.2.1 that a design pattern

instance is a *group* of classes, of *unknown* size, and organized in a tree structure? What kind of features can be exploited?

In Chapter 5 the concept of micro-structure was introduced and a lot of micro-structures of different kinds were described. Micro-structures are employed by the Joiner module (explained in Chapter 6) for the extraction of pattern instances, because they provide a way of representing different aspect of analyzed system, by exposing different properties independently and with the same syntax. A single micro-structure can match different kinds of code pieces that share a certain characteristic, and they are not ambiguous. They are designed to abstract from the details of the code and expose more abstract concepts. Given their properties, it would be logical to use micro-structures as features into our dataset representation.

Table 7.2: Example format of a dataset representing classes using micro-structures as features

	MS_1	MS_2	MS_3	MS_4	MS_5
Class 1	1	1	1	0	0
Class 2	0	0	1	1	1
Class 3	0	1	0	1	1

The direct representation of classes in a dataset form using micro-structures as features would lead to a dataset like the one shown in Table 7.2. The dataset represents three classes, on the rows, described by five different values of micro-structures (from MS_1 to MS_5); each cell contains 1 if the micro-structure is present in the class, and 0 if not. It is clear that the representation is very far from the target representation needed. The major issue is that the number of classes in a design pattern instance is unknown. If this was not true, it would be possible to build a dataset representation made by the micro-structure values of the classes belonging to a pattern *concatenated*, leading to vectors whose length is determined by $N_{ms} \cdot N_{cl}$, where N_{ms} is the number of micro-structures used as features, and N_{cl} is the *fixed* number of classes belonging to a pattern. Each row would contain all the information available regarding a pattern instance. Unfortunately, the number of classes in a design pattern definition is not fixed. But the number of *roles* is fixed. The number of roles is the only fixed decomposition given by the definition of a design pattern.

Applying the last modeling hypotheses to roles, instead of classes, and exploiting the way the Joiner module works it is possible to create a representation of each *role mapping* detected by the Joiner. Recalling the example used in Chapter 6 to explain the detection process, and in particular Figure 7.2, which gives an overview of the merging process applied to role mappings to create pattern instances, we can see that many role mappings concur to the creation of a single pattern instance.

Keeping track of the role mappings use to create a pattern instance it is possible to undo the process and producing all the mappings building the detected instance. Figure 7.3 depicts a more complex example of the generation of all the mappings that concurred to the building of an *Abstract Factory* pattern instance. The classes are represented by circles, whose names are built composing a prefix string representing the name role played by the class, and a numeric suffix representing the particular class. The meanings of the prefixes are: af $\rightarrow$ Abstract Factory, ap $\rightarrow$ Abstract Product, cf $\rightarrow$ Concrete Factory, cp $\rightarrow$ Concrete Product.

The list of mappings is clearly made by fixed size elements, and each element is a class. Composing this kind of representation with the direct micro-structure representation of classes shown in Table 7.2 it is possible to achieve a new representation of mappings as dataset rows, shown in Table 7.3 as an example composing the previous ones. In the example the two mappings (the rows) are represented by fifteen features, created combining three roles (R_1, R_2, R_3) with five micro-structures (from MS_1 to MS_5). Every single feature R_iMS_j tells if the j-th micro-

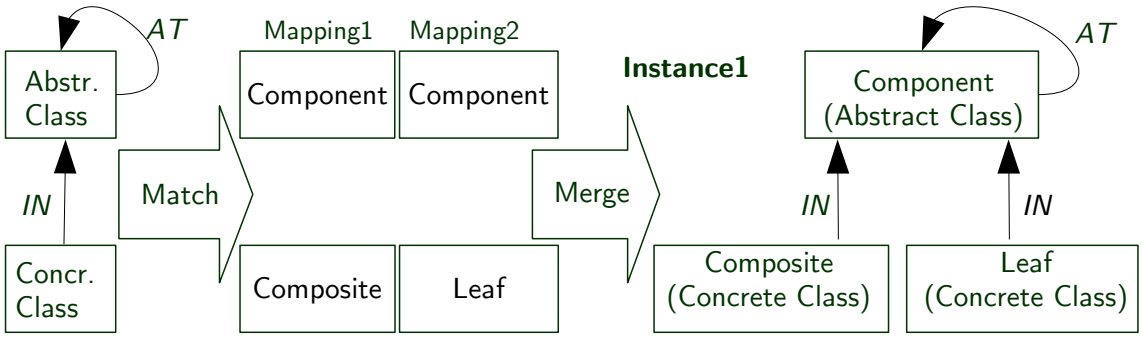


Figure 7.2: Merge process example (recalled). IN:Inheritance, AT: Abstract Type

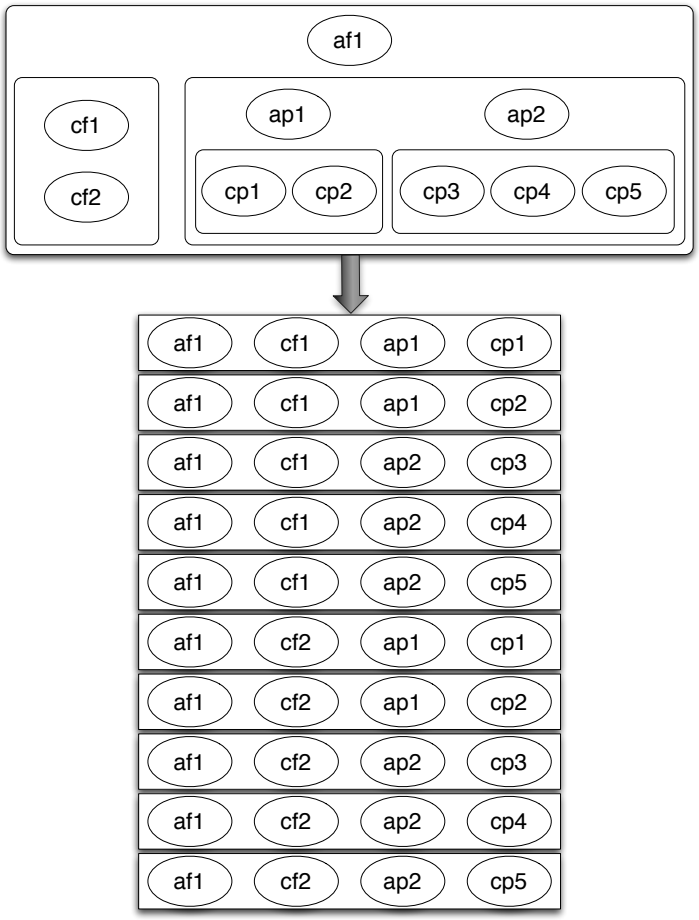


Figure 7.3: Mapping generation example for an *Abstract Factory* instance

structure is present in the class playing the i-th role in the respective mapping. Please notice that in this kind of dataset the information about pattern instances is lost: the rows represent the mappings, not the instances. There is no reference to the instance the mapping belongs to, and no class label; those pieces of information are kept out of the dataset and exploited in another moment. Summarizing, the dataset describes the classes contained in the mappings using only the micro-structures and the role assignments.

Table 7.3: Example format of a dataset representing role mappings using micro-structures

	R_1MS_1	R_1MS_2	R_1MS_3	R_1MS_4	R_1MS_5	R_2MS_1	R_2MS_2	R_2MS_3	R_2MS_4	R_2MS_5	R_3MS_1	R_3MS_2	R_3MS_3	R_3MS_4	R_3MS_5
Mapping 1	1	1	1	0	0	0	1	1	1	0	1	1	0	1	1
Mapping 2	0	1	1	1	0	0	0	1	0	0	1	0	1	1	0

A dataset having this format is not compatible with supervised classification, because each row is a role mapping, and not a pattern instance. Despite the incompatibility, the representation has the interesting property of being suitable as input for machine learning algorithms, while representing the information at a manageable abstraction level. In fact, the information about the belonging of a mapping to a pattern instance can be kept out of the dataset, and used in a different moment.

For this reason a new elaboration step was added before the supervised classification in order to obtain only one feature vector for each instance. This step groups the mappings in clusters, and represents each pattern instance using the set of clusters its mappings belongs to. Figure 7.4 shows the follow-up of the previous example shown in Figure 7.1: The mappings are grouped by a clusterer algorithm in $k = 10$ clusters. Then the pattern instance is represented as a vector in a dataset, having k features (one for each cluster). Each cell of the vectors for the i-th feature tells if the instance (the row) contains a mapping that was clustered in cluster i. Adding the class label to the vector the target dataset form shown in Table 7.1 is obtained.

The overall process handles the problem of the unknown size of the pattern instance imposing a number of clusters, therefore limiting the number of features to represent a single pattern instance to a fixed number. If the meaning of the dataset used for the clustering process is quite clear and directly related to the system and the pattern definition, the meaning of the last dataset is less obvious and requires a little discussion.

The clustering process groups mappings, which can belong to different pattern instances, using the properties (described by micro-structures) of the classes they are composed of. No information about patterns is exploited. This means that the clusters represent a number of similar mappings, which we can call mapping types. An example can clarify the approach: a hypothetical clustering approach would be, for example, to create completely homogeneous clusters, i.e. where every element is identical to the others. A clustering approach like that would be feasible and could produce clusters containing more than one instance, because the representation of a class using micro-structure represents a projection of its structure in the micro-structure space, and there is no guarantee that the set of micro-structures employed in the experiments is able to uniquely identify a class in a system. Returning to the example, this particular clustering, applied to the generation of the second dataset, would produce a dataset where each feature represents one of the available combinations of {classes, roles, micro-structures} in the retrieved mappings. The example highlights two important problems. The first is that, as the number n of employed micro-structures grows, the number of possible combinations to represent a mapping grows exponentially, with the value of 2^{rn}, where r is the number of roles of the pattern. The second problem is that the learning process is composed of

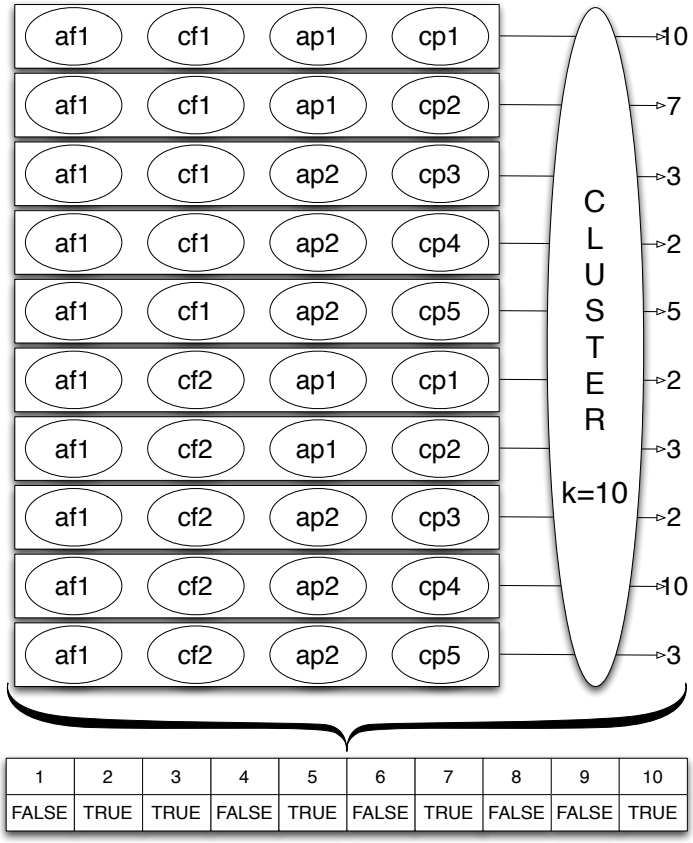


Figure 7.4: Cluster Generation

two steps: training and test, which correspond respectively to the moment when an algorithm learns about the domain, and creates its internal representation of it, and the moment in which new representations of the domain are evaluated. Given the learning process, a clustering algorithm like the one described in the example would not be able to handle new representations, different from the ones it already knows, and it would fail. For this reason a real clustering algorithm can be seen as a solution able to approximate that behaviour, having the advantage to be able to reasonably choose a cluster (in a fixed size set) for any new correctly encoded instance of the domain.

The classification dataset is therefore an encoding of the input mappings, which uses information directly retrieved from the content of the system in the form of micro-structures. The encoding can always be created from a mapping because micro-structures are not ambiguous. Moreover, micro-structures are abstract features of the code, and role mappings are directly derived from the pattern definition and the analyzed system, so the dataset encoding maintains the semantics exposed by the two parts of the combination.

7.3 Evolution of the methodology

The classification methodology described up to now is a basic version. During the development and real implementation of the solution, many improvements were made to it. The remainder of this section describes the different improvements.

7.3.1 Micro-structures representation

The basic usage of micro-structures for the creation of the dataset of role mappings was to represent each micro-structure as a *single* boolean value telling if that micro-structure is present in that particular class. Micro-structures, as explained in Chapter 5, are *binary* relationships (or facts) among classes or other code entities. The fact of being unary is seen as a particular case where the source and destination of the micro-structure are the same. Some micro-structures are unary by definition, e.g. AbstractClass, FinalClass, Immutable: all the ones describing a simple property of a class. Being more precise with respect to the creation of the dataset (as explained up to now) every cell is a boolean value telling if a particular class in a role is the source of a particular micro-structure. This modeling loses the information about the destination of a micro-structure. An important property of micro-structures to remember is that, in general, there is no limit to the number of different micro-structures of the same kind contained in a class. For example, the Delegate EDP represents a method call existing between two methods belonging to different classes and with different signatures. It is clear that this kind of relationship is very common.

To enhance the representation of the role mappings in the dataset, the information about the destination of each micro-structure was included: A new dimension was added to the definition of a single feature, representing the destination role of a particular micro-structure. Every feature is a boolean value defined as $R_i MS_j R_k$. A cell tells if in a mapping the class in the i-th role is the source of a micro-structure j having as destination the class in the k role of the same mapping. This representation better describes the mapping, fully exploiting the available information. Another improvement was done adding a default *Other* role to each mapping, representing every other class of the system (not represented by the current mapping), which is used as either source or destination (but not both of them) in the feature building described above. This solution allows the clusterer to have an approximated idea of the relationships among the represented mappings and the rest of the system.

7.3.2 Choice of the micro-structures

The improvement introduced in the previous subsection contributed to the creation of a plethora of features in the role mapping dataset representation. A deeper analysis of the available micro-structures resulted in a more accurate selection of the micro-structures to use. Table 7.4 contains the list of the micro-structures selected for the machine learning process, with additional simple properties added for the machine learning process. The content of Table 7.4 is called *feature setup* in MARPLE. The interpretation of the columns of the table is:

Category: it is a categorization of the micro-structures: everything other than Micro Patterns and Elemental Design Patterns are Design Pattern Clues. Clues are divided in several categories, and some category is not present in the original catalogue.

Cardinality: it admits two values (B: Binary, U: Unary). When it assumes the "Unary" value, the behaviour explained in Subsection 7.3.1 changes and reverts to the original one. In other words only one feature $R_iMS_jR_i$ is produced for each role i and unary feature j.

Del Self?: stands for "Delete self?", and can be applied only to micro-structures having "Binary" cardinality. When "Del Self?" = "Yes", the micro-structure will not produce a self link (where source is equal to destination). The consequence is that only features like $R_iMS_jR_k$, with $i \neq k$ will be produced for micro-structure j having "Del Self?" = "Yes".

Table 7.4: Micro-structures selection

Category	**Name**	**Cardinality**	**Del Self?**
Basic Attribute Information	OtherInstanceReference	B	No
	OtherStaticReference	B	No
	PrivateInstanceReference	B	No
	PrivateStaticReference	B	No
	ProtectedInstanceReference	B	No
	ProtectedStaticReference	B	No
Basic Class Relationship	AbstractMethodInvoked	B	No
	ClassInherited	B	Yes
	InterfaceInherited	B	Yes
	SameClass	B	Yes
Basic Method Information	ControlledParameter	B	No
	InheritanceThisParameter	B	No
	PrivateConstructor	U	
	ProtectedConstructor	U	
Basic Returned Elements Information	CloneReturned	U	
	DifferentHierarchyObjectReturned	B	Yes
	SameHierarchyObjectReturned	B	Yes
Basic Type Information	FinalClass	U	
Behavioural Clue	AbstractCycleTerminationCall	B	No

Table 7.4: Micro-structures selection

Category	Name	Cardinality	Del Self?
Behavioural Clue	AbstractCyclicCall	B	No
	ManyThisParameterCallTarget	U	
Elemental Design Pattern	AbstractInterface	U	
	Conglomeration	B	No
	Delegate	B	Yes
	DelegatedConglomeration	B	No
	DelegateInFamily	B	Yes
	DelegateInLimitedFamily	B	Yes
	ExtendMethod	B	Yes
	Recursion	B	No
	Redirect	B	Yes
	RedirectInFamily	B	Yes
	RedirectInLimitedFamily	B	Yes
	RedirectRecursion	B	No
	Retrieve	B	No
	RevertMethod	B	Yes
Information Clue	AbstractType	U	
	CloneableImplemented	U	
	ConcreteProductGetter	U	
	ConcreteProductsReturned	U	
	ControlledException	U	
	ControlledInstantiation	B	No
	DirectReturnedObject	B	No
	EmptyConcreteProductGetter	U	
	FactoryParameter	U	
	MultipleReturnedObject	B	No
	MultipleReturns	B	No
	ReceivesParameter	B	No
	SingleReturnedObject	B	No
	VoidReturn	U	
Micro Pattern	AugmentedType	U	
	Box	U	
	Canopy	U	
	CobolLike	U	
	CommonState	U	
	CompoundBox	U	
	DataManager	U	
	Designator	U	
	Extender	U	
	FunctionObject	U	
	FunctionPointer	U	

Table 7.4: Micro-structures selection

Category	Name	Cardinality	Del Self?
Micro Pattern	Immutable	U	
	Implementor	U	
	Joiner	U	
	Outline	U	
	Overrider	U	
	Pool	U	
	PseudoClass	U	
	PureType	U	
	Record	U	
	RestrictedCreation	U	
	Sampler	U	
	Sink	U	
	Stateless	U	
	StateMachine	U	
	Taxonomy	U	
	Trait	U	
Structural Clues	AdapterMethod	U	
	AllMethodsInvoked	B	No
	ExtendedInheritance	B	Yes
	InstanceInAbstractReferred	B	No
	ProductReturns	B	No

The feature setup is a more precise characterization of what is needed in the role mapping dataset, and contributes to a more compact and manageable feature space, without removing important information. The choice of the micro-structures was made by selecting the ones with a clear definition, removing the ones that were not completely proved and tested or that were not considered to be useful in a design pattern detection task. In fact, many other kinds of micro-structures are available in MARPLE, e.g. for the detection of code smells and anti-patterns. In addition some new micro-structure was created, to better characterize some aspect of a software systems that were not addressed by the existing ones. The new micro-structures were designed to have some useful property when used as features. For example, the six "*Reference" micro-structures in the "Basic Attribute Information" category represent attributes of a class as links from the class to the type of the attribute. Their characterization is the combination of two criteria {Private, Protected, Other} and {Static, Instance} that characterize the modifiers of the attribute for visibility and the presence of absence of the **static** keyword. The characterization is total: each attribute can report only one of the six micro-structures. This kind of property allows to safely group the features by summarizing over one or both the criteria, reducing the number of features. The choice of grouping features can be done to enhance the machine learning performance for example when a set of features is too sparse to be significant or if the feature space is still to large to be computed. Grouping is only a planned feature, and it will be implemented in future work. The feature setup will be integrated with new information that will allow to automatically choose a set of features to group, e.g. when some programmable

constraints on the performance values and the feature space will be satisfied.

7.3.3 Single level patterns

The last important enhancement of the machine learning process is the handling of design patterns whose definitions are composed of only one level. Examples of this kind of patterns are Singleton and Adapter, whose merge rule diagrams are reported below.

The problem with single-level patterns is that, applying the dataset generation process explained in this chapter to them, the output dataset, containing the representation of the instances using clusters as features, has a peculiar form. In fact, instances of design patterns with a single level are composed of only one mapping, producing a classifier dataset having only one true value per row. In fact, if each instance contains exactly one mapping, and the mappings are clustered in k clusters, each vector describing an instance will have k cells, one having value "true" (the one corresponding to the cluster containing the only mapping), and the other $k-1$ will have value "false".

A dataset in that form reduces the supervised classification problem to a choice of the best set of clusters representing the design pattern, which is an over-simplification, but a solution is available to improve this condition. In fact, if a single mapping is present in this kind of patterns, it is possible to directly use the role mapping dataset for the supervised classification, by adding the class label to each mapping, because there is only one of both for each instance.

A slightly different approach was chosen, to avoid different implementations of similar processes: a bypass clusterer. When a single-level design pattern is analyzed, the applied clusterer simply produces a classifier dataset having the same number of features of the input dataset, using this criterion: cluster i contains a mapping if the value of the i-th feature of the mapping is "true". In other words, it produces a copy of the input dataset, with different feature names. This solution allows the process to remain the same, adding only the choice of the right clusterer for the pattern.

The only change to the methodology, is that the clustering phase shifts from a regular hard clustering task to a *soft clustering* [53, 149, 35] one. Soft clustering occurs when it is possible to assign an input vector to more than one cluster, creating a multi-categorization, like tagging on the web. Soft clustering is a generalization of the regular clustering, so the change in the methodology was simple and safe.

7.4 User experience

The MARPLE tool supports an iterative and incremental design pattern discovery and evaluation methodology. The user creates a "MARPLE Developer Project", selecting which projects of the same workspace he wants to analyze. Then in the new project he creates a "MARPLE Developer File", which is an xmi file containing the setup of the analysis. The editor shows a red box, representing the Information Detector. By right-clicking on the box and selecting "Run" from the menu the Information Detector starts and collects all the metrics and micro-structures contained in the analyzed project. Saving the analysis file triggers the save of all collected information. This behaviour is consistent for all the elaboration steps.

Now MARPLE is ready for design pattern detection. By clicking "Add Elaboration Chain" in the Eclipse toolbar, two new boxes appear in the analysis editor, representing the Joiner (green) and the Classifier (blue) modules. A right-click on the outer yellow box lets to choose

the pattern to detect, by selecting the “Change chain properties” menu item. Once selected a pattern and confirmed, MARPLE is in a state like the one shown in Figure 7.5.

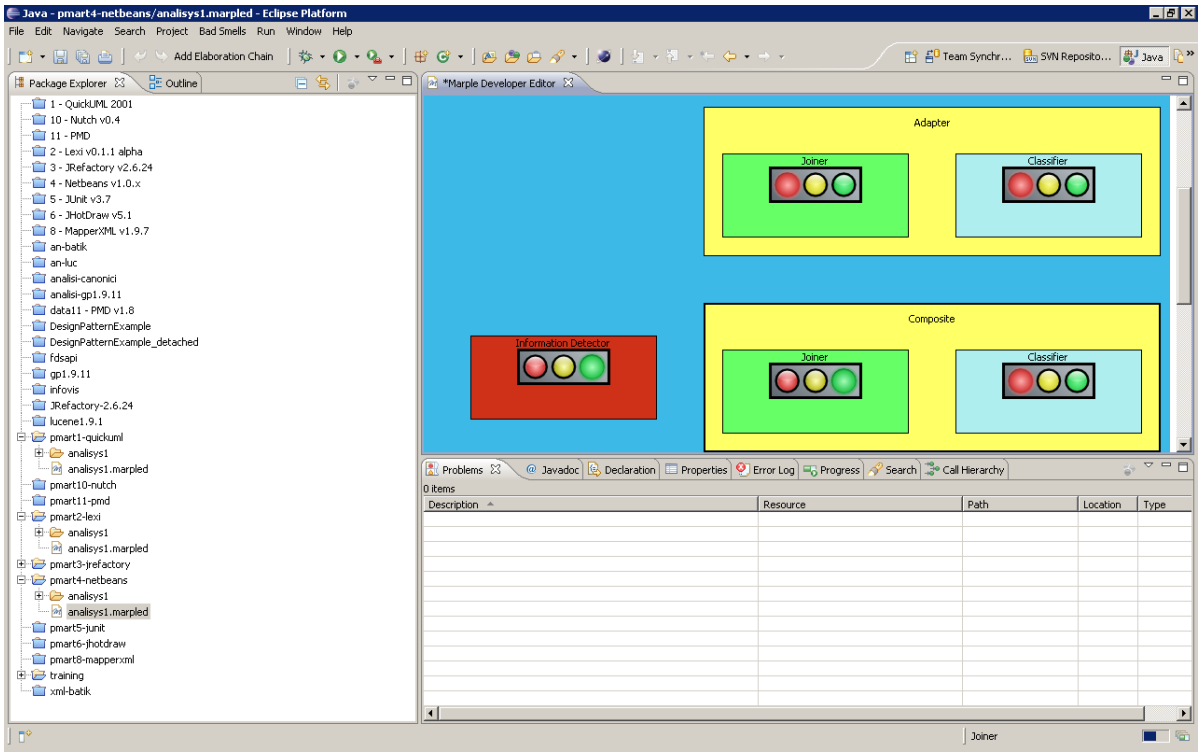


Figure 7.5: MARPLE configured for pattern detection

The detection of design patterns candidates can be triggered by selecting the “Run” item in the contextual menu of the Joiner module. Then, selecting the “Show results” menu item a table view opens with the list of the detected design pattern candidates. For each pattern the names of the classes with role in the root level is shown, together with a combo box for the evaluation of the pattern and a column for the confidence value. Selecting a row and clicking the buttons “Graph” and “Joiner Model” in the “DP Instance Selection” view, two different graphical views of the selected pattern instance are shown, the first representing the classes as nodes in a graph (where the edges are the micro-structures), and the second following the graphical representation of the merge rule diagram (see Section 6.3). An example result is shown in Figure 7.6.

After having inspected some instances, it is possible to evaluate them as “CORRECT” or “INCORRECT” using the combo box. After a number of instances have been evaluated, it is possible to submit them to the Classifier module. First, the “Copy instances to training” contextual menu item must be selected, and then the “Run training” one. The splitting of the two commands is caused by the fact that the instances used from the training are copied in a separate “training” project, which can be shared across many different analyses for different projects. The training project allows summing the instances coming from different systems to have a bigger training set. After the “Run training” command is finished, the clusterer and classifier algorithms have been trained and persisted to file, in the training project.

At this point it is possible to select the “Run” menu item on the Classifier module, to fill the “Confidence” column in the “DP Instance Selection” view. An example of the evaluation of the *Adapter* design pattern is shown in Figure 7.7.

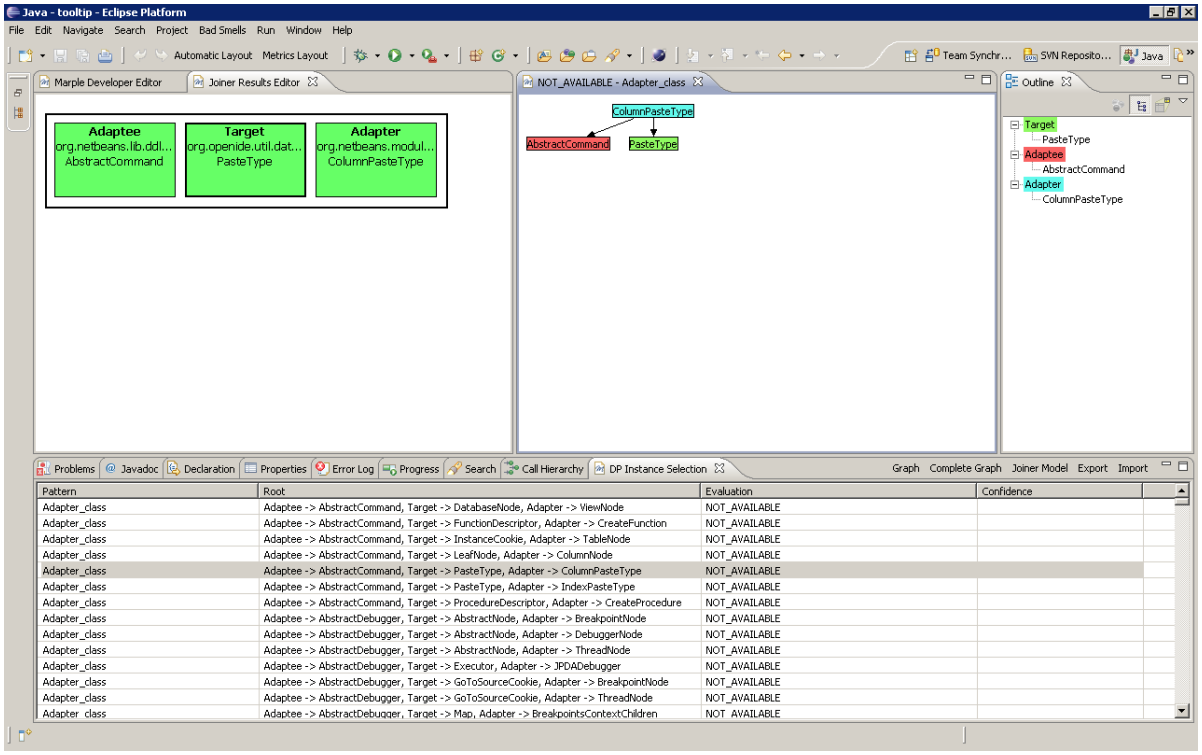


Figure 7.6: MARPLE showing a pattern candidate

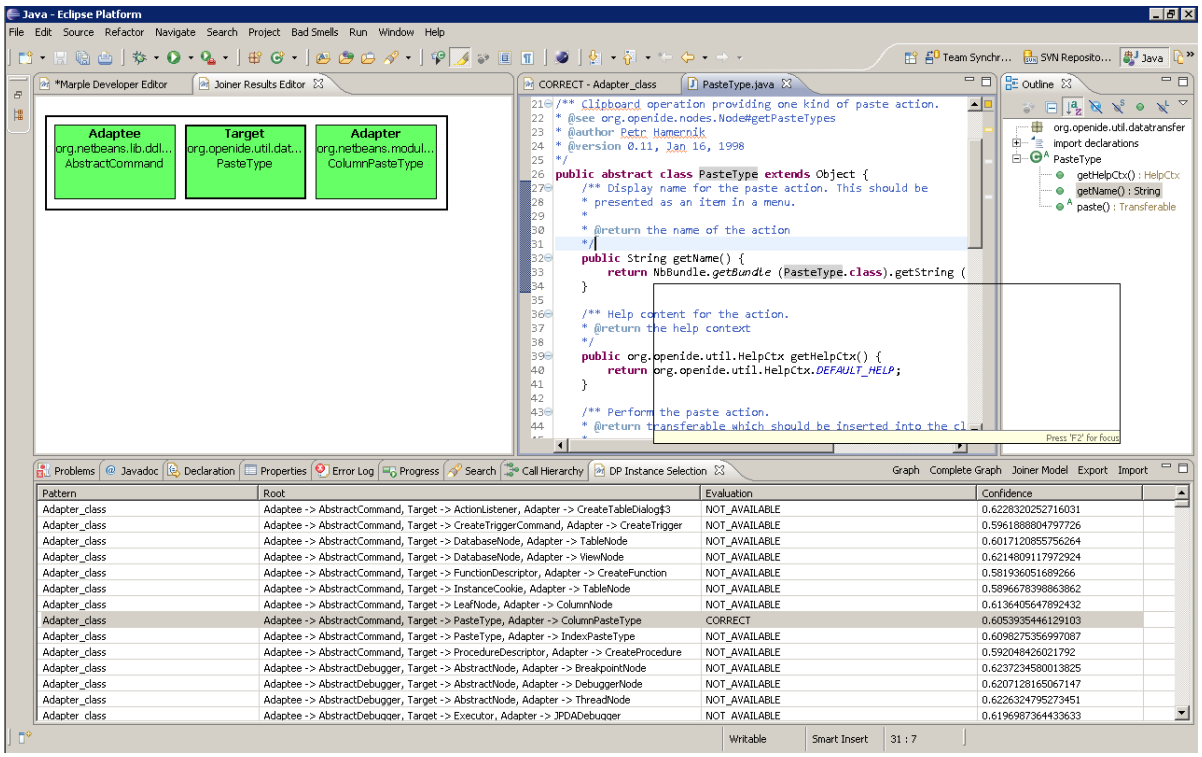


Figure 7.7: MARPLE showing an evaluated pattern instance

7.5 Conclusion

This chapter described the machine learning methodology applied to the problem of design pattern detection. The methodology is focused on the *modeling* of the input and output formats of the learning algorithms employed. Machine learning algorithms traditionally work on datasets or matrices composed of vectors, which describe the *domain* of the problem. Domain modeling is important, sometimes crucial, in object-oriented software solutions: machine learning software solutions are not different in this sense. It is important to describe the domain to the algorithms in the best possible manner, to hope to have some kind of sensible results. Machine learning algorithms performances can be influenced by the input representation: Many techniques to improve their performance are based on the different kind of modification of the input format, like the use of kernel functions [166] in Support Vector Machines (SVM) [50] to change the feature space, or the selection of the features to use during the learning [90, 46, 48]. For this reason most of the effort was in the direction of dataset and feature modeling.

Another important feature of the methodology is that it does not rely on a particular algorithm or set of algorithms, but it just focuses on the general formulation of supervised and unsupervised classification problem. No concrete technology or algorithm was mentioned in this chapter with the purpose of highlighting this aspect. The fact of being "algorithm-agnostic" brings a nice modularity in the implementation of the analysis system, allowing free plugging of different algorithms without the need of rewriting some kind of adaptation code.

Chapter 8

Experimentations with MARPLE-DPD

The Classifier module was tested against the detection of five design patterns: *Singleton*, *Adapter*, *Composite*, *Decorator*, *Factory Method*. The detection rule for the five patterns are specified in Section 6.3, while the ones for all the other design patterns are available in Appendix A. This chapter explains the performed experiments and reports the obtained results.

8.1 Experiments

The experiments on the five design patterns were conducted applying a set of clustering and classification algorithms. The choice of the algorithms was made looking to the ones available for the Weka [196, 91] framework.

8.1.1 Algorithms

The choice of the algorithms to test resulted in the following list:

- ZeroR is a simple classification rule that chooses always the dominant class; it is useful to include it in the tests because it provides a baseline telling how much the problem is unbalanced.
- OneR [100] is a classification rule that chooses the attribute giving the minimum-error prediction for the classification. This is another kind of baseline useful to measure the difficulty of the problem.
- NaiveBayes [109] is the simplest bayesian network available. It makes strong assumptions on the input: features are considered as independent.
- JRip [49] is a rule learner. Its advantage is to be able to produce simple propositional logic rules for the classification, which are understandable to humans and simply translatable in logic programming.
- RandomForest [41] is a classifier that build a forest of random decision trees, each one using a subset of the input features.
- J48 [156] is an implementation of the C4.5 decision tree. It has the advantage of producing human-understandable rules for the classification of new instances.

- SMO [152, 113, 94] is an implementation of John Platt's sequential minimal optimization algorithm for training a support vector classifier. In the experimentation only the RBF (Radial Basis Function) kernel is used in combination with this classifier.
- LibSVM [63, 44] is another support vector machine (SVM) library, available to Weka using an external adapter. Two SVM variants are experimented (C-SVC, ν-SVC), in combination with four different kernels (Linear, Polynomial, RBF, Sigmoid).
- SimpleKMeans [23] is an implementation of the k means algorithm. It was exploited in two variants: with the Euclidean and Manhattan distances.
- CLOPE [202] is a clustering algorithm for transactional data. Its advantages are being very fast and designed for nominal attributes datasets.
- SelfOrganizingMap [120] is a clusterer that implements Kohonen's Self-Organizing Map[1] algorithm for unsupervised clustering. Self Organizing Maps are a special kind of competitive networks.
- LVQ [121] is a clusterer that implements the Learning Vector Quantization algorithm for unsupervised clustering.
- Cobweb [71, 75] is an hierarchical clusterer implementing the Cobweb and Classit clustering algorithms.

8.1.2 Projects

For each pattern, a set of pattern instances was extracted using the Joiner module, and then manually classified with the support of the user interface of the Classifier module, integrated with the Eclipse IDE. The set of projects used for the gathering of the design pattern instances was composed of a project containing example pattern instances gathered on the web, and the projects used for the PMARt [83] dataset. The summary of the experimented systems and some demographic metrics about them are shown in Table 8.1.

Table 8.1: Projects for the experimentations

Project	CUs	Packages	Types	Methods	Attributes	TLOC
DesignPatternExample	1060	235	1749	4710	1786	32313
1 - QuickUML 2001	156	11	230	1082	421	9233
2 - Lexi v0.1.1 alpha	24	6	100	677	229	7101
3 - JRefactory v2.6.24	569	49	578	4883	902	79732
4 - Netbeans v1.0.x	2444	184	6278	28568	7611	317542
5 - JUnit v3.7	78	10	104	648	138	4956
6 - JHotDraw v5.1	155	11	174	1316	331	8876
8 - MapperXML v1.9.7	217	25	257	2120	691	14928
10 - Nutch v0.4	165	19	335	1854	1309	23579
11 - PMD v1.8	446	35	519	3665	1463	41554

CUs: Number of Compilation Units — TLOC: Total number of Lines of Code

[1] A more tested SOM package is available in MATLAB. Some tests were conducted with the MATLAB GUI, and a adaptation module was developed to call MATLAB from Java. Unfortunately, a non-documented incompatibility did not allow the Java-MATLAB bridge to work. It appears from the Mathworks support forum that it is not possible to train a neural network from Java.

Each Joiner rule was originally designed and tested against the project containing the example patterns. The idea behind the approach is to have the Joiner extracting (possibly) all the pattern instances contained in a software system, having virtually 100% recall. To achieve this goal, each rule (the ones in Chapter 6 and in Appendix A) was tuned to be able to catch *all* the pattern instances contained in the DesignPatternExampleproject. Then, the five rules of the experimented design patterns were enhanced during the experimentations. In fact, every pattern contained in the PMARt dataset was checked to be present in the results obtained by the Joiner, and when an instance was missed, the rule was analyzed and modified to include the missing instance (without losing the others). Another kind of enhancement was adding more selective constraints to avoid the explosion of the number of results.

Some problems rose during the comparison with the PMARt dataset. The instances of the five tested patterns contained in PMARt revealed to be only partially correct. In particular, 14 instances out of 61; the 14 instances in PMARt contain 26 classes having key roles in their patterns, raising the number of wrong instances in the MARPLE definition to 26. The error contained in the dataset are of different kinds: for example in 4 - Netbeans v1.0.xorg.netbeans.modules.form.FormAdapter is reported as playing the Adapter role in the *Adapter* pattern, but it is an empty class; another example is net.sourceforge.pmd.ast.JavaParserConstants in 11 - PMD v1.8, which is reported as a Product for a *Factory Method*, but it is an interface only defining static constants, which has no meaning to instantiate. Other errors concerns, e.g., the wrong assignment of roles to some class in the pattern instance. The corrections will be discussed with the authors of PMARt after the submission if this thesis.

8.1.3 Patterns

The patterns to test were chosen by looking in the literature for the ones reported to be more frequent. The estimation was made analyzing the results coming from PMARt, the Design Pattern Detection Tool from Tsantalis et al. [194, 195] and design pattern detection results from Rasool et al. [157, 175]. Table 8.2 reports the amount of instances found and evaluated for each experimented pattern. The *Adapter* and *Factory Method* patterns have a number of candidates with a "+" suffix, because the number of candidates is not complete. In fact, those two patterns have been experimented only with the projects from "DesignPatternExample" to "4 - Netbeans v1.0.x", because during the evaluation of instances in "4 - Netbeans v1.0.x", in both cases, the number of evaluations grew over 1000, which I considered the threshold to decide to stop the manual evaluation. The decision of stopping to evaluate instances is motivated by the great amount of time employed in the evaluation task: the validation of a single pattern instance can take from 15 seconds to 5 minutes, depending on the complexity of instance itself. The 1000 threshold was set to reach a significant dataset for the patterns that made it possible.

Table 8.2: Summary of detected pattern instances

Pattern	Candidates	Evaluated	Correct	Incorrect
Singleton	154	154	58	96
Adapter	5861+	1221	618	603
Composite	128	128	30	98
Decorator	250	247	93	154
Factory Method	2546+	1044	562	482

The evaluations were then exploited for the search of the best classification setup. *Singleton* and *Adapter* are single-level patterns, so only classification algorithms were tested, while for the other patterns also clustering algorithms were included in the tests.

8.1.4 Parameter optimization

The search for the best parameter of a machine learning algorithm is a long, tedious, and error-prone (when results are recorded manually) task. Each algorithm has a set of parameters, each one having its domain, which can be of different types (i.e. continuous, discrete, boolean, nominal) and the entire set of parameters of a single algorithm is a potentially huge space to explore. The traditional way of exploring the space is a grid search[2], which means that every parameter must be discretized, by defining a discretization criterion for continuous parameters and assigning a discrete number to boolean and nominal ones. Then for each possible combination of parameters the algorithm must be tested and its performance values (e.g. confusion matrix, area under ROC, precision, recall) must be recorded. The set of performance values must be then analyzed and the best parameter set must be searched, defining one or more comparison criteria. The approach is defined to be sure to have a complete description of the parameter space with respect to the performance values. The problem with this methodology is that, as already introduced, it is time-consuming.

A faster evaluation was needed, and an approach exploiting genetic algorithms was taken to try to reach the objective. Genetic algorithms [54, 127, 78] are an approach for the solution of optimization problems, inspired from the biological research and that allows sub-optimal solutions. Its general formulation is simple to adapt to different problems, and many implementation are available for research purposes. In particular I exploited JGAP[3], which is able to abstract from the algorithm details, providing a simple way of extending the framework just adding a new FitnessFunction. In general, optimization problems are characterized by the search for the best set of parameters to pass to a fitness function: when the maximum (or minimum for minimization problems) value is reached, the parameters are the optimal solution to the problem. The same kind of approach was already tested in the literature [163].

Performance estimators

In my context, I defined a fitness function able to perform ten-fold cross validation[33, 178] over the evaluated pattern instances, using different performance indexes [26] as fitness values: accuracy, f-measure[197], area under ROC [40].

Accuracy Accuracy is one of the simplest performance measure for classification tasks: it is the percentage of correctly classified instances, in the positive and negative class. It is usually never reported alone, because it has several drawbacks, in particular when the positive and negative classes are unbalanced: a typical issue in classification tasks is having a little number of positive instances and a huge number of negative instances. In such cases, a naive classifier always giving the negative answers would have high accuracy. As the most important class is usually the positive one, it is clear that accuracy is not very useful as-is. Other kind of performance indicators exist. In fact, the second indicator employed is f-measure.

F-measure It is defined (in the basic version) as the harmonic mean of precision and recall, which are other two performance values. Precision is the part of the positive-classified instances which is really positive, while recall is the part of really-positive instances classified as positive. The first measures the ability to correctly choose a positive instance, the second the ability to extract positive instances among the others. The two measures tend to be in contrast and the goal is usually to find the best trade-off value of the two. F-measure is a way of having a single number combining the two measures.

[2] like suggested, e.g., in `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`

[3] `http://jgap.sourceforge.net/`

Area under ROC The last indicator is the area under ROC (Receiver Operating Characteristic). The ROC is a plot of true positive rate against false positive rate as the discrimination threshold of the classifier is varied. The area under ROC grows near to value 1 when the discrimination performs better, while bad classification brings values near to 0.5. The area under ROC has a stricter relation with the ability of ranking the results [67], because of the way the curve is built.

All the described performance values are to be considered useful only in combination with each other and in relation to the domain. Moreover, research in machine learning and information retrieval is evolving, and some indicators are being criticized [92]. The approach taken in this experimentation is to rely on available performance indicators, exploiting the ones able to give a single score value and comparing the models from different perspectives.

8.2 Results

For each experimented design pattern, the classifiers' setups having the best performance results are reported and discussed, together with the performance values. The experiments with the Classifier module are made on the output on the Joiner. Every direct or indirect performance value (including *recall*) is calculated on that particular dataset. In other words, the Classifier module performance is estimated under the hypothesis that the Joiner succeeded in extracting every pattern instance contained in the analyzed systems.

8.2.1 Singleton

Singleton is a single-level pattern, so only classifier algorithms were experimented. Table 8.3 reports the best results for each classifier type (the best value of the column is in bold font). When the same classifier is reported in different variants it is because some options lead to major differences in the algorithms being applied, e.g. the pruning setting in J48 and the SVM type and kernel in LibSVM. The numbers near the performance values are the identifiers of the configuration able to produce it; they will be used in Table 8.5 and Table 8.6 to uniquely identify a particular classifier setup. The values reported in Table 8.3 are the ones achieved during the parameter optimization procedure. The table shows performance values that can be considered quite good, because in many cases the values go beyond 0.9. The accuracy value near to 0.6 of the ZeroR classifier can be used as a simple measure of the balance between the classes, which is good because it is near to 0.5. At this level it is interesting to notice that the best performances were achieved by three different algorithm types.

Table 8.4 gives a more detailed view of the distribution of the retrieved and evaluated instances among the different projects. The values are referred to the instances as they came from the Joiner, and how they were manually evaluated. No learning was involved in this phase.

The classifier configurations whose performances are reported in Table 8.3 were then used to build an experiment in the Weka Experimenter module, which allows to test different classifiers against the same data and to compare their performance, calculating the standard deviations of the performance values and the statistical significance of the comparisons. In fact, to correctly compare different classifiers (with respect to one performance value), ranking the values is not enough, but a standard test must be applied with a significance value. The tests were conducted with the default 0.05 significance value. Table 8.5 contains the performance values, calculated using the Weka experimenter, for the classifiers defined in Table 8.3, accompanied by their standard deviations. For each classifier its reference number and class name are reported. The values are not the same (for the respective classifier setup) as the ones shown in Table 8.3, because the experimenter by default repeats each cross validation 10 times with different dataset

Table 8.3: Best performance results for the *Singleton* design pattern

Classifier	accuracy		f-measure		area under ROC	
ZeroR	0.61	(1)	0.00	(1)	0.48	(1)
OneR	0.87	(2)	0.83	(2)	0.86	(2)
NaiveBayes	0.81	(3)	0.73	(3)	0.89	(3)
JRip	0.88	(5)	0.85	(6)	0.88	(4)
RandomForest	**0.93**	(7)	0.90	(9)	**0.97**	(8)
J48 Unpruned	0.87	(10)	0.82	(12)	0.91	(11)
J48 Reduced Error Pruning	0.88	(13)	**0.91**	(15)	0.85	(14)
J48 Pruned	0.88	(16)	0.84	(18)	0.91	(17)
SMO RBF	0.90	(19)	0.86	(21)	0.94	(20)
C-SVC Linear	0.83	(22)	0.77	(24)	0.87	(23)
C-SVC Polynomial	0.91	(25)	0.88	(27)	0.94	(26)
C-SVC RBF	0.92	(28)	0.89	(30)	0.95	(29)
C-SVC Sigmoid	0.86	(31)	0.88	(33)	0.94	(32)
ν-SVC Linear	0.91	(34)	0.88	(36)	0.93	(35)
ν-SVC Polynomial	0.90	(37)	0.87	(39)	0.94	(38)
ν-SVC RBF	0.91	(40)	0.89	(42)	0.95	(41)
ν-SVC Sigmoid	0.90	(43)	0.79	(45)	0.92	(44)

Table 8.4: *Singleton*: evaluated instances summary

Project	Candidates	Evaluated	Correct	Incorrect
DesignPatternExample	18	18	18	0
1 - QuickUML 2001	2	2	1	1
2 - Lexi v0.1.1 alpha	2	2	2	0
3 - JRefactory v2.6.24	10	10	6	4
4 - Netbeans v1.0.x	99	99	22	77
5 - JUnit v3.7	0	0	0	0
6 - JHotDraw v5.1	1	1	1	0
8 - MapperXML v1.9.7	4	4	3	1
10 - Nutch v0.4	14	14	2	12
11 - PMD v1.8	4	4	3	1
Total	154	154	58	96

randomizations. The shown values are the average on the 10 different runs.

To be able to tell which classifier is the best among the tested ones, and with a certain significance, it is possible to apply a ranking test in the Weka experimenter. Table 8.6 contains the ranking for the accuracy, f-measure and area under ROC of all the classifiers. The ranking test ranks the different classifiers by counting how many times each classifier significantly wins or loses against the other ones. The Wins−Losses difference gives the ranking score, telling which classifier is the best. From the tables it is simple to understand that there is no single winner, but different algorithms perform best for different purposes.

One of the most noticeable facts is that accuracy and f-measure give a similar ranking to the classifiers. In fact, classifiers (28), (41), and (42) occupy the top 3 positions. All the three classifiers are support vector machines using a RBF kernel, using LibSVM and two different SVM types. The top three positions for the area under ROC are given to the three RandomForest classifiers, instead, followed by another support vector machine with RBF kernel and from the (41) classifier. The overall results suggest that support vector machines with RBF kernel can be considered the best trade-off choice for the *Singleton* design pattern. The fact these good performance values are achieved on a relatively small dataset (less than two hundred instances) can be an indicator of the fact this pattern is simple to discriminate because the micro-structures are able to describe well its features, or that the dataset is biased in some way, and it would need more instances.

Table 8.5: *Singleton*: experiments summary. Significance level: 0.05

	Classifier	accuracy	stddev	F-measure	stddev	auc	stddev
1	rules.ZeroR	0.61	0.02	0.00	0.00	0.50	0.00
2	rules.OneR	0.87	0.08	0.82	0.11	0.86	0.09
3	bayes.NaiveBayes	0.81	0.09	0.72	0.16	0.89	0.08
4	rules.JRip	0.88	0.08	0.85	0.10	0.88	0.09
5	rules.JRip	0.88	0.07	0.85	0.10	0.88	0.08
6	rules.JRip	0.87	0.08	0.84	0.10	0.88	0.08
7	trees.RandomForest	0.89	0.07	0.85	0.10	0.97	0.04
8	trees.RandomForest	0.89	0.07	0.85	0.11	0.97	0.03
9	trees.RandomForest	0.89	0.08	0.85	0.11	0.96	0.04
10	trees.J48	0.86	0.08	0.80	0.12	0.90	0.08
11	trees.J48	0.86	0.08	0.82	0.10	0.91	0.07
12	trees.J48	0.86	0.07	0.80	0.12	0.91	0.07
13	trees.J48	0.87	0.08	0.82	0.10	0.87	0.08
14	trees.J48	0.87	0.08	0.82	0.10	0.87	0.09
15	trees.J48	0.84	0.09	0.78	0.14	0.85	0.10
16	trees.J48	0.89	0.08	0.87	0.10	0.89	0.08
17	trees.J48	0.86	0.08	0.80	0.12	0.90	0.08
18	trees.J48	0.87	0.08	0.83	0.12	0.88	0.08
19	functions.SMO	0.89	0.08	0.84	0.12	0.95	0.05
20	functions.SMO	0.88	0.07	0.83	0.12	0.95	0.05
21	functions.SMO	0.89	0.08	0.85	0.12	0.95	0.05
22	functions.LibSVM	0.84	0.09	0.78	0.13	0.91	0.08
23	functions.LibSVM	0.84	0.09	0.78	0.15	0.90	0.08
24	functions.LibSVM	0.84	0.09	0.77	0.14	0.91	0.08
25	functions.LibSVM	0.89	0.07	0.85	0.11	0.88	0.09
26	functions.LibSVM	0.89	0.08	0.85	0.12	0.94	0.06
27	functions.LibSVM	0.88	0.08	0.84	0.12	0.87	0.09
28	functions.LibSVM	0.90	0.07	0.86	0.10	0.95	0.05
29	functions.LibSVM	0.90	0.07	0.86	0.11	0.95	0.06
30	functions.LibSVM	0.89	0.07	0.85	0.10	0.95	0.05
31	functions.LibSVM	0.83	0.08	0.73	0.15	0.79	0.10
32	functions.LibSVM	0.88	0.08	0.82	0.13	0.95	0.05
33	functions.LibSVM	0.87	0.07	0.81	0.12	0.94	0.05
34	functions.LibSVM	0.88	0.07	0.82	0.12	0.94	0.05
35	functions.LibSVM	0.88	0.08	0.82	0.12	0.95	0.05
36	functions.LibSVM	0.88	0.07	0.82	0.12	0.94	0.05
37	functions.LibSVM	0.89	0.07	0.85	0.11	0.94	0.06
38	functions.LibSVM	0.86	0.08	0.80	0.13	0.95	0.05
39	functions.LibSVM	0.89	0.07	0.85	0.11	0.94	0.06
40	functions.LibSVM	0.89	0.07	0.85	0.11	0.95	0.06
41	functions.LibSVM	0.90	0.08	0.86	0.11	0.95	0.05
42	functions.LibSVM	0.90	0.07	0.86	0.11	0.95	0.05
43	functions.LibSVM	0.86	0.08	0.79	0.13	0.83	0.09
44	functions.LibSVM	0.84	0.08	0.77	0.14	0.92	0.07
45	functions.LibSVM	0.82	0.09	0.69	0.19	0.77	0.11
	Average	0.87		0.80		0.90	

Table 8.6: *Singleton* Classifier Ranking. W−L: Wins−Losses, W: Wins, L:Losses

Accuracy				f-measure				area under ROC			
Classifier	W−L	W	L	Classifier	W−L	W	L	Classifier	W−L	W	L
(42)	9	9	0	(42)	8	8	0	(7)	25	25	0
(28)	6	6	0	(41)	8	8	0	(9)	24	24	0
(41)	5	5	0	(28)	8	8	0	(8)	24	24	0
(40)	5	5	0	(30)	6	6	0	(21)	20	20	0
(37)	5	5	0	(29)	6	6	0	(41)	19	19	0
(29)	5	5	0	(25)	6	6	0	(19)	19	19	0
(39)	4	4	0	(40)	5	5	0	(32)	17	17	0
(30)	4	4	0	(16)	5	5	0	(30)	17	17	0
(26)	4	4	0	(39)	4	4	0	(42)	16	16	0
(25)	4	4	0	(37)	4	4	0	(40)	16	16	0
(16)	4	4	0	(26)	4	4	0	(38)	16	16	0
(8)	4	4	0	(21)	4	4	0	(36)	16	16	0
(7)	4	4	0	(20)	4	4	0	(35)	16	16	0
(21)	3	3	0	(19)	4	4	0	(34)	16	16	0
(20)	3	3	0	(9)	4	4	0	(29)	16	16	0
(19)	3	3	0	(8)	4	4	0	(28)	16	16	0
(9)	3	3	0	(7)	4	4	0	(20)	16	16	0
(36)	2	2	0	(6)	4	4	0	(39)	14	14	0
(35)	2	2	0	(5)	4	4	0	(37)	14	14	0
(43)	1	1	0	(4)	4	4	0	(26)	14	14	0
(38)	1	1	0	(36)	3	3	0	(33)	13	14	1
(34)	1	1	0	(35)	3	3	0	(11)	3	6	3
(33)	1	1	0	(34)	3	3	0	(44)	1	4	3
(32)	1	1	0	(27)	3	3	0	(12)	1	5	4
(27)	1	1	0	(11)	3	3	0	(24)	0	4	4
(18)	1	1	0	(38)	2	2	0	(22)	0	4	4
(17)	1	1	0	(33)	2	2	0	(17)	0	5	5
(14)	1	1	0	(32)	2	2	0	(10)	0	5	5
(13)	1	1	0	(18)	2	2	0	(23)	-4	4	8
(12)	1	1	0	(14)	2	2	0	(16)	-14	3	17
(11)	1	1	0	(13)	2	2	0	(3)	-14	3	17
(10)	1	1	0	(2)	2	2	0	(27)	-18	3	21
(6)	1	1	0	(17)	1	1	0	(25)	-18	3	21
(5)	1	1	0	(12)	1	1	0	(18)	-18	3	21
(4)	1	1	0	(10)	1	1	0	(13)	-18	3	21
(2)	1	1	0	(23)	0	1	1	(6)	-18	3	21
(23)	0	1	1	(15)	0	1	1	(5)	-18	3	21
(24)	-1	1	2	(43)	-1	1	2	(4)	-18	3	21
(22)	-1	1	2	(24)	-3	1	4	(14)	-19	2	21
(15)	-1	1	2	(44)	-5	1	6	(15)	-21	1	22
(44)	-4	1	5	(22)	-5	1	6	(2)	-23	2	25
(31)	-11	1	12	(31)	-19	1	20	(43)	-28	1	29
(45)	-16	1	17	(3)	-24	1	25	(31)	-37	1	38
(3)	-18	1	19	(45)	-31	1	32	(45)	-39	1	40
(1)	-44	0	44	(1)	-44	0	44	(1)	-44	0	44

8.2.2 Adapter

The *Adapter* design pattern had the same experimental setup of *Singleton*. Table 8.7 contains the best results obtained during the parameter optimization, and it is interesting to see that the best performance were achieved always by support vector machines with a RBF kernel type, even if from two different kinds of algorithm.

Table 8.7: Best performance results for the *Adapter* design pattern

Classifier	accuracy		f-measure		area under ROC	
ZeroR	0.53	(1)	0.00	(1)	0.50	(1)
OneR	0.68	(2)	0.66	(2)	0.68	(2)
NaiveBayes	0.70	(3)	0.70	(3)	0.78	(3)
JRip	0.81	(4)	0.77	(5)	0.82	(6)
RandomForest	0.85	(7)	0.84	(8)	0.92	(9)
J48 Unpruned	0.80	(10)	0.79	(11)	0.86	(12)
J48 Reduced Error Pruning	0.79	(13)	0.79	(14)	0.84	(15)
J48 Pruned	0.80	(16)	0.79	(17)	0.86	(18)
SMO RBF	0.85	(19)	0.84	(20)	0.92	(21)
C-SVC Linear	0.80	(22)	0.79	(23)	0.85	(24)
C-SVC Polynomial	0.84	(25)	0.82	(26)	0.89	(27)
C-SVC RBF	**0.86**	(28)	**0.85**	(29)	0.92	(30)
C-SVC Sigmoid	0.79	(31)	0.69	(32)	0.84	(33)
ν-SVC Linear	0.80	(34)	0.79	(35)	0.85	(36)
ν-SVC Polynomial	0.84	(37)	0.82	(38)	0.91	(39)
ν-SVC RBF	**0.86**	(40)	0.84	(41)	**0.93**	(42)
ν-SVC Sigmoid	0.75	(43)	0.64	(44)	0.82	(45)

The evaluation of the *Adapter* candidates stopped during the analysis of 4 - Netbeans v1.0.x, because the number of evaluated patterns reached 1221. Table 8.8 suggests that *Adapter* is widely present as a pattern, which is a confirmation for the considerations expressed in Section 6.3.2.

Table 8.8: *Adapter*: evaluated instances summary

Project	Candidates	Evaluated	Correct	Incorrect
DesignPatternExample	212	212	128	84
1 - QuickUML 2001	75	69	13	56
2 - Lexi v0.1.1 alpha	65	65	44	21
3 - JRefactory v2.6.24	508	500	261	239
4 - Netbeans v1.0.x	5003	375	172	203
5 - JUnit v3.7	-	-	-	-
6 - JHotDraw v5.1	-	-	-	-
8 - MapperXML v1.9.7	-	-	-	-
10 - Nutch v0.4	-	-	-	-
11 - PMD v1.8	-	-	-	-
Total	5861+	1221	618	603

The usage of the Weka Experimenter of the best classifier configurations for the *Adapter* is

summarized in Table 8.9. The table shows data that are consistent with the ones shown for the *Singleton* pattern. In fact, the best-fitting classifiers are support vector machine and random forests. They are also among the ones with the lower standard deviation values. The overall shown performance is over 0.8 for the best setup, which can be considered good with respect to the 0.53 accuracy of ZeroR and ~0.65 performance of OneR.

A deeper comparison of the classifiers is available in Table 8.10, which shows the ranking of the classifiers for the *Adapter pattern*. Classifiers number {(28),(29),(30),(40),(42)} are in the top five positions for all the three performance values. Even the difference in terms of Wins−Losses is low (0 for the area under ROC). With no surprise, the classifiers are support vector machines with RBF kernels. RandomForest classifiers are lower in ranking, but the difference Wins−Losses score is not much lower than the top positions. In fact, the first ten positions of the rank never lose a comparison, and the first 18 have a positive Wins−Losses balance. The 18th position is the limit of significantly good classifiers. The classifiers in the first 18 positions are occupied by support vector machines and random forests. That 18th position line separates the classifiers that are significantly better than the others, and very distant in terms of ranking.

The results for the *Adapter* and *Singleton* design patterns suggest that the classification strategy adopted for single-level design patterns is reasonable and produces sensible results. The performance values are high in both patterns and quite stable also when different randomizations of the dataset are used for the cross validation. In fact, the peak values shown in Table 8.7 are not much higher than the average values produced by the Experimenter and shown in Table 8.9, where the standard deviations are under 0.05 in most cases.

Table 8.9: *Adapter*: experiments summary. Significance level: 0.05

	Classifier	accuracy	stddev	fmeasure	stddev	auc	stddev
1	rules.ZeroR	0.53	0.00	0.00	0.00	0.50	0.00
2	rules.OneR	0.68	0.04	0.65	0.05	0.68	0.04
3	bayes.NaiveBayes	0.70	0.04	0.70	0.05	0.78	0.05
4	rules.JRip	0.77	0.04	0.73	0.05	0.76	0.04
5	rules.JRip	0.77	0.04	0.73	0.05	0.76	0.04
6	rules.JRip	0.77	0.04	0.73	0.05	0.76	0.04
7	trees.RandomForest	0.83	0.04	0.82	0.04	0.91	0.03
8	trees.RandomForest	0.83	0.04	0.82	0.04	0.92	0.03
9	trees.RandomForest	0.83	0.04	0.82	0.04	0.92	0.03
10	trees.J48	0.77	0.04	0.75	0.05	0.78	0.05
11	trees.J48	0.76	0.05	0.75	0.05	0.81	0.05
12	trees.J48	0.77	0.04	0.75	0.05	0.83	0.04
13	trees.J48	0.76	0.05	0.75	0.05	0.81	0.05
14	trees.J48	0.76	0.05	0.75	0.05	0.81	0.05
15	trees.J48	0.75	0.05	0.73	0.06	0.80	0.05
16	trees.J48	0.76	0.05	0.74	0.05	0.80	0.05
17	trees.J48	0.77	0.04	0.75	0.05	0.80	0.05
18	trees.J48	0.76	0.05	0.75	0.05	0.83	0.04
19	functions.SMO	0.83	0.04	0.83	0.04	0.91	0.03
20	functions.SMO	0.83	0.04	0.82	0.04	0.83	0.04
21	functions.SMO	0.82	0.04	0.82	0.04	0.91	0.03
22	functions.LibSVM	0.78	0.04	0.76	0.05	0.77	0.04
23	functions.LibSVM	0.78	0.04	0.76	0.05	0.78	0.04
24	functions.LibSVM	0.78	0.04	0.77	0.05	0.83	0.04
25	functions.LibSVM	0.82	0.04	0.81	0.04	0.88	0.03
26	functions.LibSVM	0.82	0.04	0.81	0.04	0.88	0.03
27	functions.LibSVM	0.81	0.04	0.80	0.04	0.88	0.03
28	functions.LibSVM	0.84	0.04	0.83	0.04	0.91	0.03
29	functions.LibSVM	0.84	0.04	0.83	0.04	0.91	0.03
30	functions.LibSVM	0.84	0.04	0.83	0.04	0.91	0.03
31	functions.LibSVM	0.77	0.04	0.75	0.05	0.83	0.04
32	functions.LibSVM	0.63	0.05	0.67	0.04	0.71	0.05
33	functions.LibSVM	0.76	0.04	0.74	0.05	0.83	0.04
34	functions.LibSVM	0.78	0.04	0.77	0.04	0.78	0.04
35	functions.LibSVM	0.78	0.04	0.77	0.05	0.78	0.04
36	functions.LibSVM	0.77	0.04	0.75	0.04	0.84	0.04
37	functions.LibSVM	0.82	0.04	0.81	0.04	0.89	0.03
38	functions.LibSVM	0.82	0.04	0.81	0.04	0.82	0.04
39	functions.LibSVM	0.82	0.04	0.81	0.04	0.89	0.03
40	functions.LibSVM	0.84	0.04	0.83	0.04	0.91	0.03
41	functions.LibSVM	0.83	0.04	0.82	0.04	0.83	0.04
42	functions.LibSVM	0.84	0.04	0.83	0.04	0.91	0.03
43	functions.LibSVM	0.67	0.14	0.63	0.16	0.66	0.14
44	functions.LibSVM	0.47	0.00	0.64	0.00	0.50	0.00
45	functions.LibSVM	0.74	0.04	0.72	0.05	0.80	0.05
	Average	0.77		0.75		0.82	

Table 8.10: *Adapter* Classifier Ranking. W−L: Wins−Losses, W: Wins, L:Losses

Accuracy				f-measure				area under ROC			
Classifier	W−L	W	L	Classifier	W−L	W	L	Classifier	W−L	W	L
(28)	31	31	0	(42)	32	32	0	(42)	35	35	0
(40)	29	29	0	(29)	32	32	0	(40)	35	35	0
(42)	28	28	0	(28)	32	32	0	(30)	35	35	0
(30)	28	28	0	(30)	31	31	0	(29)	35	35	0
(29)	28	28	0	(40)	30	30	0	(28)	35	35	0
(41)	27	27	0	(41)	28	28	0	(21)	35	35	0
(38)	27	27	0	(21)	28	28	0	(19)	35	35	0
(37)	27	27	0	(20)	28	28	0	(9)	35	35	0
(21)	27	27	0	(19)	28	28	0	(8)	35	35	0
(20)	27	27	0	(37)	27	27	0	(7)	35	35	0
(19)	27	27	0	(9)	27	27	0	(39)	23	33	10
(9)	27	27	0	(8)	27	27	0	(37)	23	33	10
(8)	27	27	0	(7)	27	27	0	(26)	19	31	12
(7)	27	27	0	(39)	23	27	4	(25)	19	31	12
(26)	26	27	1	(26)	23	27	4	(27)	16	30	14
(25)	26	27	1	(25)	23	27	4	(36)	3	18	15
(39)	25	27	2	(38)	22	27	5	(31)	3	18	15
(27)	22	27	5	(27)	18	27	9	(18)	3	18	15
(35)	-10	8	18	(36)	-11	7	18	(12)	3	18	15
(34)	-10	8	18	(35)	-11	7	18	(24)	2	17	15
(24)	-10	8	18	(34)	-11	7	18	(41)	1	16	15
(36)	-11	7	18	(33)	-11	7	18	(20)	1	16	15
(31)	-11	7	18	(31)	-11	7	18	(38)	-1	14	15
(23)	-11	7	18	(24)	-11	7	18	(33)	-2	15	17
(22)	-11	7	18	(23)	-11	7	18	(14)	-7	8	15
(33)	-12	6	18	(22)	-11	7	18	(13)	-7	8	15
(17)	-12	6	18	(17)	-11	7	18	(11)	-7	8	15
(12)	-12	6	18	(18)	-12	6	18	(15)	-12	5	17
(10)	-12	6	18	(14)	-12	6	18	(45)	-15	8	23
(6)	-12	6	18	(13)	-12	6	18	(16)	-15	5	20
(5)	-12	6	18	(12)	-12	6	18	(17)	-17	5	22
(4)	-12	6	18	(11)	-12	6	18	(35)	-19	5	24
(18)	-13	5	18	(10)	-12	6	18	(34)	-19	5	24
(16)	-13	5	18	(16)	-13	5	18	(23)	-19	5	24
(14)	-13	5	18	(15)	-14	4	18	(22)	-19	5	24
(13)	-13	5	18	(6)	-14	4	18	(10)	-19	5	24
(11)	-13	5	18	(5)	-14	4	18	(3)	-19	5	24
(15)	-16	5	21	(4)	-14	4	18	(6)	-23	5	28
(45)	-21	5	26	(45)	-23	4	27	(5)	-23	5	28
(43)	-29	2	31	(3)	-31	3	34	(4)	-23	5	28
(3)	-36	3	39	(43)	-32	1	33	(32)	-37	3	40
(2)	-36	3	39	(32)	-37	2	39	(43)	-38	2	40
(32)	-39	2	41	(2)	-39	1	40	(2)	-39	2	41
(1)	-42	1	43	(44)	-40	1	41	(44)	-43	0	43
(44)	-44	0	44	(1)	-44	0	44	(1)	-43	0	43

8.2.3 Composite

The experimental setup for the *Composite* (and all the next patterns) is a bit different. First, *Composite* is a multi-level pattern, so it needs a clustering algorithm for the detection. Second, less clusterers and classifiers were employed for the experiments than the ones described in Subsection 8.1.1, making the experiments less time and resource-demanding. In Subsection 8.3.4 the issues related to clusterers are discussed. The reduction of the number of employed classifiers is given only by time constraints for the production of the results. Only the faster and better performing (on average) were kept. In particular all the LibSVM with kernels different from RBF were removed, and also the SMO classifier. The rationale is that, among the SVMs, the RBF is the kernel with better performance and lower computation time. SMO with the RBF kernel was removed because it performs similar to LibSVM with the RBF kernel.

The best results achieved for the *Composite* design pattern are reported in Table 8.11. The performances are by far lower than the ones reported in Table 8.3 and Table 8.7 for the *Singleton* and *Adapter* design patterns, respectively. It is not possible to give an idea of the significance of the results, but some considerations can be formulated. First, the maximum (even if modest) performance values are scored by the support vector machines classifiers. Second, there is no clue for deducing which is the best clusterer among the two, even calculating the mean and standard deviation of the performance values for the two cases.

Table 8.11: Best performance results for the *Composite* design pattern

Clusterer	Classifier	accuracy		f-measure		area under ROC	
SimpleKMeans	ZeroR	0.75	(1)	0.00	(1)	0.39	(1)
SimpleKMeans	OneR	0.75	(2)	0.11	(2)	0.50	(2)
SimpleKMeans	NaiveBayes	0.77	(3)	0.38	(3)	0.63	(3)
SimpleKMeans	JRip	0.75	(4)	0.35	(5)	0.48	(6)
SimpleKMeans	RandomForest	0.75	(7)	0.45	(8)	0.61	(9)
SimpleKMeans	J48 Unpruned	0.75	(10)	0.27	(11)	0.59	(12)
SimpleKMeans	J48 Reduced Error Pruning	0.77	(13)	0.25	(14)	0.51	(15)
SimpleKMeans	J48 Pruned	0.75	(16)	0.00	(17)	0.51	(18)
SimpleKMeans	C-SVC RBF	0.79	(19)	0.36	(20)	**0.88**	(21)
SimpleKMeans	ν-SVC RBF	**0.81**	(22)	0.55	(23)	0.67	(24)
CLOPE	ZeroR	0.75	(25)	0.00	(25)	0.39	(25)
CLOPE	OneR	0.75	(26)	0.00	(26)	0.50	(26)
CLOPE	NaiveBayes	0.75	(27)	0.12	(27)	0.52	(27)
CLOPE	JRip	**0.81**	(28)	0.42	(29)	0.60	(30)
CLOPE	RandomForest	**0.81**	(31)	0.38	(32)	0.65	(33)
CLOPE	J48 Unpruned	**0.81**	(34)	0.42	(35)	0.53	(36)
CLOPE	J48 Reduced Error Pruning	**0.81**	(37)	0.25	(38)	0.56	(39)
CLOPE	J48 Pruned	0.75	(40)	0.42	(41)	0.58	(42)
CLOPE	C-SVC RBF	**0.81**	(43)	0.40	(44)	0.61	(45)
CLOPE	ν-SVC RBF	0.77	(46)	**0.56**	(47)	0.72	(48)

The summary of the per-project evaluated instance is available in Table 8.12. The table shows that all the available systems were analyzed, and their pattern candidates evaluated, because the total number of instances was less than 1000. In fact, only 128 instances were found

in all the systems analyzed. The not very high amount of pattern instances can be motivated by the diffusion of the pattern, which is high but not pervasive, and from the fact that *Composite* instances tend to be composed of many classes, and therefore of many role mappings. The size of the instances has the direct consequence of lowering the total number of the instances.

Table 8.12: *Composite*: evaluated instances summary

Project	Candidates	Evaluated	Correct	Incorrect
DesignPatternExample	27	27	15	12
1 - QuickUML 2001	5	5	1	4
2 - Lexi v0.1.1 alpha	1	1	0	1
3 - JRefactory v2.6.24	2	2	0	2
4 - Netbeans v1.0.x	71	71	9	62
5 - JUnit v3.7	1	1	1	0
6 - JHotDraw v5.1	5	5	2	3
8 - MapperXML v1.9.7	5	5	0	5
10 - Nutch v0.4	10	10	2	8
11 - PMD v1.8	1	1	0	1
Total	128	128	30	98

One of the motivations of the low performances on this pattern could be the insufficient information contained in the dataset. Future work will try to demonstrate this hypothesis.

The last major difference with the results of the first two design patterns is that no results are available from the Weka Experimenter: it is not possible to evaluate the significance level of the produced results. The reason is two-fold:

- First, Weka is able to run only classification experiments (or other one-type only experiments), while the needed procedure is too complex. In fact, a single run requires the application of a clusterer chained with a classifier. Even writing a new classifier for Weka able to hide the complexity of the process to the Experimenter, a problem remains regarding the cross validation: the input of the clusterer is a dataset without the class labels, which is an information that stays outside the process until the supervised classification phase starts. Without the prior knowledge about the class labeling, the Experimenter is not able to record the analytical results for its estimations.

- Second, even producing a dataset containing the summary data from all the experiments conducted during the optimization phase, the Experimenter is not prepared to handle all the produced data, and needs a huge amount of memory and time. In addition, the results would not be comparable, and the significance levels and standard deviations would have less meaning, because the data available from the optimization algorithm is the summary of an entire cross validation, while the Experimenter is designed to handle the results coming from all the single foldings of a cross validation, repeated a certain number of times on different randomizations of the same dataset.

In Section 9.1 some future works related to these issues are described.

8.2.4 Decorator

The experiment regarding the *Decorator* design pattern has the same setup of the one on the *Composite* design pattern, for the same causes.

The best performance values obtained are shown in Table 8.13. The scenario is quite different from the one available in Table 8.11 for the *Composite* design pattern. The first significant difference is the splitting of the values between the two clusterers: SimpleKMeans beats CLOPE in most cases. Another difference is that the f-measure values for the *Decorator* are almost double of the ones in the *Composite*, and area under ROC values are significantly higher. At a first impression it seems that the *Decorator* dataset had a better classification outcome than the one for the *Composite*. One peculiarity of the results is that all the best performance values are scored by RandomForest classifiers. Random forests had good results also on the *Singleton* and *Adapter* patterns, but they never totally won over support vector machines. SVMs are just a bit lower in performance, and a suspect comes that, if a significance analysis was possible, the support vector machines could be considered equal or superior to the random forests, in the same way it happened for the first two patterns.

Table 8.13: Best performance results for the *Decorator* design pattern

Clusterer	Classifier	accuracy		f-measure		area under ROC	
SimpleKMeans	ZeroR	0.58	(1)	0.00	(1)	0.49	(1)
SimpleKMeans	OneR	0.70	(2)	0.56	(2)	0.65	(2)
SimpleKMeans	NaiveBayes	0.77	(3)	0.74	(3)	0.76	(3)
SimpleKMeans	JRip	0.80	(4)	0.76	(5)	0.76	(6)
SimpleKMeans	RandomForest	**0.82**	(7)	**0.77**	(8)	**0.82**	(9)
SimpleKMeans	J48 Unpruned	0.77	(10)	0.75	(11)	0.74	(12)
SimpleKMeans	J48 Reduced Error Pruning	0.77	(13)	0.73	(14)	0.77	(15)
SimpleKMeans	J48 Pruned	0.80	(16)	0.76	(17)	0.75	(18)
SimpleKMeans	C-SVC RBF	0.80	(19)	0.75	(20)	**0.82**	(21)
SimpleKMeans	ν-SVC RBF	0.80	(22)	0.76	(23)	0.81	(24)
CLOPE	ZeroR	0.58	(25)	0.00	(25)	0.49	(25)
CLOPE	OneR	0.66	(26)	0.59	(26)	0.64	(26)
CLOPE	NaiveBayes	0.70	(27)	0.67	(27)	0.73	(27)
CLOPE	JRip	0.71	(28)	0.65	(29)	0.68	(30)
CLOPE	RandomForest	0.73	(31)	0.72	(32)	0.74	(33)
CLOPE	J48 Unpruned	0.73	(34)	0.66	(35)	0.74	(36)
CLOPE	J48 Reduced Error Pruning	0.72	(37)	0.68	(38)	0.74	(39)
CLOPE	J48 Pruned	0.73	(40)	0.65	(41)	0.73	(42)
CLOPE	C-SVC RBF	0.72	(43)	0.66	(44)	0.72	(45)
CLOPE	ν-SVC RBF	0.74	(46)	0.70	(47)	0.76	(48)

The evaluated instances for the *Decorator* design pattern are more than the ones for the *Composite*, which are 128. In fact, the dataset is composed of 247 instances, 93 correct and 154 incorrect. The dataset is therefore also better balanced, having 38% of correct instances and 62% of incorrect instances, while the percentage for the *Composite* is respectively 23% and 77%.

The better performance gained on the *Decorator* (with respect to the *Composite*) can be justified also with the larger size of the dataset, which is almost double, and the better balance between correct and incorrect instances that favours a better learning for the classifiers.

Table 8.14: *Decorator*: evaluated instances summary

Project	Candidates	Evaluated	Correct	Incorrect
DesignPatternExample	31	31	10	21
1 - QuickUML 2001	16	16	2	14
2 - Lexi v0.1.1 alpha	6	6	0	6
3 - JRefactory v2.6.24	12	12	1	11
4 - Netbeans v1.0.x	131	128	59	69
5 - JUnit v3.7	7	7	1	6
6 - JHotDraw v5.1	14	14	5	9
8 - MapperXML v1.9.7	16	16	8	8
10 - Nutch v0.4	11	11	6	5
11 - PMD v1.8	6	6	1	5
Total	250	247	93	154

8.2.5 Factory Method

The last experimented pattern is the *Factory Method*, which has the same experimental setup of the last two analyzed patterns.

In Table 8.15 the best results achieved for the *Factory Method* are reported. This results report another enhancement over the ones reported in Table 8.13 for the *Decorator* design pattern. The f-measure is over 0.8 in most cases, and the area under ROC reaches the maximum of 0.87. Accuracy values are comparable to the ones in for the *Decorator*, near to 0.8 in most cases. Another time, SimpleKMeans clearly beats CLOPE: results obtained through the CLOPE clusterer on the *Factory Method* are lower than the same experiments applied to the *Decorator*. There is a confirmation instead regarding the classifier algorithms: random forests have the maximum performance for all the indicators, and support vector machines are equal or a few behind. The real surprise in the table are the performances of NaiveBayes. In fact, while for all the other patterns it has an average or bad performance, here it scores performances very near to the top ones. The causes of these good scores are unknown, but some hypotheses can be formulated. The simplest one is that on this dataset the SimpleKMeans is able to group the mappings in a way that simplifies a lot the job of the classifiers, leading to good average results and the outstanding values of the NaiveBayes, which is a simple[4] (but sometimes very effective) technique. A confirmation of this hypothesis is that, for example, for the *Composite* the NaiveBayes has performances comparable to the average ones, and very near to the ones of the ZeroR and OneR. These values suggest that the performed clustering did not help the classifiers to choose the correct instances. In the *Decorator*'s results, instead the values of ZeroR, OneR and NaiveBayes progressively grow, suggesting that as the complexity of the classifier augments, better results can be achieved because more complicated relationships among the features (the clusters) can be calculated. The best values are better than the ones for the NaiveBayes, but not so much. Finally, in the *Factory Method* there is the same performance growing, but the NaiveBayes has similar performances to the top ones. This last observation suggests that the extracted clusters are meaningful enough without, the need to combine their values in complicated ways, like more advanced classifiers are able to do.

The number of instances found in the target systems for the *Factory Method* design pattern is huge compared with the *Composite* and *Decorator* patterns. One of the causes is the large

[4]The naïve bayes classification technique is the simplest bayesian network. It is composed of one level, and assumes the input features are statistically independent, which is a strong assumption

Table 8.15: Best performance results for the *Factory Method* design pattern

Clusterer	Classifier	accuracy		f-measure		area under ROC	
SimpleKMeans	ZeroR	0.52	(1)	0.69	(1)	0.50	(1)
SimpleKMeans	OneR	0.70	(2)	0.73	(2)	0.70	(2)
SimpleKMeans	NaiveBayes	0.81	(3)	0.82	(3)	**0.87**	(3)
SimpleKMeans	JRip	0.76	(4)	0.79	(5)	0.77	(6)
SimpleKMeans	RandomForest	**0.82**	(7)	**0.83**	(8)	**0.87**	(9)
SimpleKMeans	J48 Unpruned	0.80	(10)	0.81	(11)	0.86	(12)
SimpleKMeans	J48 Reduced Error Pruning	0.78	(13)	0.81	(14)	0.85	(15)
SimpleKMeans	J48 Pruned	0.79	(16)	0.81	(17)	0.85	(18)
SimpleKMeans	C-SVC RBF	**0.82**	(19)	**0.83**	(20)	0.84	(21)
SimpleKMeans	ν-SVC RBF	0.80	(22)	0.82	(23)	**0.87**	(24)
CLOPE	ZeroR	0.52	(25)	0.69	(25)	0.50	(25)
CLOPE	OneR	0.54	(26)	0.69	(26)	0.53	(26)
CLOPE	NaiveBayes	0.55	(27)	0.69	(27)	0.55	(27)
CLOPE	JRip	0.54	(28)	0.69	(29)	0.52	(30)
CLOPE	RandomForest	0.55	(31)	0.69	(32)	0.57	(33)
CLOPE	J48 Unpruned	0.55	(34)	0.69	(35)	0.54	(36)
CLOPE	J48 Reduced Error Pruning	0.55	(37)	0.69	(38)	0.55	(39)
CLOPE	J48 Pruned	0.55	(40)	0.69	(41)	0.54	(42)
CLOPE	C-SVC RBF	0.56	(43)	0.69	(44)	0.54	(45)
CLOPE	ν-SVC RBF	0.56	(46)	0.69	(47)	0.58	(48)

diffusion of this pattern, which simply prescribes an indirection in the responsibility of the creation of a new object, and another one is that the merge rule groups *Factory Method* instances using the Creator, ConcreteCreator and Product roles in the root level. The rationale is detailed in Section 6.3.1, and it is based on the definition and intent of the pattern itself. The sum of these causes led to 1044 pattern instances evaluated out of 2546 found ones. Also the balance is very good: 562 correct instances out of 482 wrong ones means that the 54% of the found instances were correct.

Table 8.16: *Factory Method*: evaluated instances summary

Project	Candidates	Evaluated	Correct	Incorrect
DesignPatternExample	269	269	218	51
1 - QuickUML 2001	55	49	26	23
2 - Lexi v0.1.1 alpha	18	18	13	5
3 - JRefactory v2.6.24	159	159	108	51
4 - Netbeans v1.0.x	2045	549	197	352
5 - JUnit v3.7	-	-	-	-
6 - JHotDraw v5.1	-	-	-	-
8 - MapperXML v1.9.7	-	-	-	-
10 - Nutch v0.4	-	-	-	-
11 - PMD v1.8	-	-	-	-
Total	2546	1044	562	482

As already outlined for the last two patterns, the size and composition of the dataset is important for the creation of the classification models. The one for the *Factory Method* is the larger dataset for the multi-level patterns and it is the better balanced, very near to the 1:1 ratio of correct and incorrect instances. This can be one of the causes of the performance values, which are better then the previous two ones.

8.3 Threats to validity and Limitations

8.3.1 Design pattern definitions

The entire goal of this thesis is to provide a way of training the tool to understand what a pattern instance is and what is not. The machine learning approach was taken following the conjecture that it is not possible to formulate the precise formal definition of a pattern using only its implementation details, because the definition of a pattern is taught to designers through high level concepts and examples, and often designers learn patterns in different ways. This experiment was conducted by training classifiers using a labeling given from only one person, i.e. the author of this thesis. This condition influences (it must!) the results, in unknown directions. The interpretation of the results given in this chapter must be therefore interpreted in this sense *"is this particular setup able to learn the definition of pattern x learned by this particular person?"*. A better experiment would be to have training sets produced by different people, or better from an agreed and shared dataset, like the one we are trying to build through our DPB [15] project. The different personal interpretation of the design pattern definitions, in fact, reduces the reliability of the results of all the existing approaches in the literature, as every researcher defines the detection rules for design patterns following his interpretation of the definition.

8.3.2 Granularity

Currently the granularity of the micro-structures employed for the modeling of the analyzed system is at class level. This means that every micro-structure is recorded into the model as a link from a Type to another one or to itself. Micro-structures (represented by the BasicElement class into the model) are supported by the model as links from one CodeEntity to another one. CodeEntity is an abstract class representing every kind of clearly recognizable and referable piece of code, i.e. classes, attributes, methods, and also procedures and global variables (referring to non object-oriented code).

The motivation of this choice is purely historical. MARPLE-DPD (see Chapter 4) is a project existing since 2008 and has been evolved to the current state. The module containing the Abstract Syntax Tree visitors catching the micro-structures was designed in order to support only class-level information granularity, while lower level granularity was supported using optional attributes. As the goal of this thesis is to demonstrate that data mining techniques applied to the pattern detection task are a sensible solution, the choice was to keep the existing (proved and tested) setup, and to focus on the development of the machine learning module.

The consequence of the choice is that the current implementation is expected to have some kind of sub-optimal performances, especially regarding precision. In fact, placing information also at the method and attribute level could allow specifying detection rules able to express that the same method (having a role in the pattern) must satisfy a set of constraints, also related to some other role played by other classes/methods/attributes in the same pattern. The same applies to the Classifier module: more precise information can help the algorithm to have a precise description of the input, and to behave accordingly. I mentioned that this will impact on precision in particular, because the current implementation, specifying rules that must match patterns using more abstract information, must use less strict constraints, which statistically include more results and thus more correct pattern instances.

The work for the immediate future will be to extend the approach to information placed to a more appropriate granularity level, in order to exploit all the information that can be extracted from the source code.

8.3.3 Libraries

The analysis of the connection from an analyzed system to its libraries is one of the major limitations of the approach. Most real-life software systems use lots of libraries to reach their goal. This is normal and it is what software reuse is about. The problem for design pattern detection is that many patterns are useful in order to extend frameworks (small or big) implemented into libraries. And frameworks use plenty of design patterns. The consequence is that in order to extract all the real instances of patterns existing in a system we should be able to represent an integrated model of the system and all its libraries. Unfortunately some issues arise, making the solution harder to implement. First, it is possible, especially in industry, that the source code of libraries is not available. Second, loading the model of every library connected to the system would mean handling a really huge model, whose significance would be low, as we expect a software system to use only a fraction of the classes of the libraries it relies on. The usage of such an integrated model like that would have many obvious performance consequences, in terms of both detection time and memory consumption. Even if the performance issues were solved, another modeling problem would raise: detecting patterns into a model including the analyzed system and all its libraries would extract also patterns which are exclusively contained in the libraries, which are of no interest to the user. Moreover a pattern half-implemented in a library is more related to the library, from the design point of view, than to the client system, but knowing about the connection between the two is useful information for program comprehension. The

ideal library handling, in the context of design pattern detection, would be to create a repository of single library analyzed models, which can be connected to the analyzed system model (like the libraries are connected to the system), and to exploit the joint information without the need to build the library model every time and already knowing the instances contained in the library. This kind of implementation needs a degree of design and scalability that is out of the scope of this thesis, but it is still the goal I think a reverse engineering tool should aim to.

In the actual implementation, MARPLE does not handle the content of libraries, with the exception of inheritance, which is retrieved using the information provided by Eclipse using the compiler bindings. That kind of information is simple to retrieve and it is available also when the source code of the library is not available. Future work will aim to at least mitigate the issues coming from libraries and perhaps provide a degree of configuration of the desired behaviour.

8.3.4 Time and computational resources

The initial experimental setup was larger than the one reported in this chapter. In particular, more clustering algorithms were tested other than CLOPE and SimpleKMeans, i.e. SelfOrganizingMaps, LVQ, Cobweb. The achieved results are not reported because for different reasons those algorithms were not applicable. SelfOrganizingMap is too slow, at least one order of magnitude, compared to, e.g., SimpleKMeans. It takes 4 hours to learn the *Composite* dataset, for example. In addition, an undocumented problem does not allow the clusterer to handle nominal attribute, even if they are accepted without returning errors. The same applied for LVQ, which does not share the same computation time, however. In both cases the clustering is null (instances are all clustered in the same cluster) if the dataset is made by nominal attributes, and only a little better when nominal attributes are transformed to a numeric (0,1) form. Finally, Cobweb does not fit into memory for most of the inputs, even allocating 3500MB of Java heap.

The remaining clustering algorithms, namely CLOPE and SimpleKMeans, are fast and have a small memory footprint, but they are also simpler as clustering technologies. For these reasons, newer and better results will be hopefully achieved by testing other clustering algorithm implementations, as outlined in Section 9.1.

8.4 Conclusion

This chapter described the experimental setup for the Classifier module, and the results obtained over five different design patterns.

Despite the experienced difficulties, especially regarding the clustering algorithms, some overall considerations can be formulated.

First, it seems that, in the average case, the kinds of classification algorithm having the best performance are the support vector machines, when paired with a RBF kernel. This is not a real surprise, as it is one of the most common kernels for SVMs and it is the default one, for example, in LibSVM. The second classifier was RandomForest, which is less stable in the performances, but reaches higher area under ROC values, better also than SVMs.

Second, as the size of the dataset grows, the performance values increase also on multi-level patterns, and a difference between the SimpleKMeans and the CLOPE clusterers becomes more evident. In fact, in the results for the *Decorator*, and more in the ones for the *Factory Method*, the clustering performed by the SimpleKMeans produces datasets that are classified with better performances. A peculiarity of the last experiment, the one on the *Factory Method* pattern is that the NaiveBayes classifier, which is not as advanced as, e.g., as support vector machines, as performance values very near to the top ones, scored by RandomForest and LibSVM. It will be

interesting, in the future, to test new clustering algorithms and to augment the smaller datasets, to understand if better performances can be achieved.

Another important validation of the approach is to compare its results with the results coming from other tools. A first attempt to provide this kind of validation is reported in Appendix C.

Chapter 9

Conclusions and Future Developments

Many approaches have been developed by the research community for design pattern detection. This thesis described an approach that applies data mining techniques to the design pattern detection problem, to solve the issues related to the informal definition given to design patterns, which often leads to slightly different interpretations by different developers.

The approach is implemented in the MARPLE tool, building the MARPLE-DPD module. MARPLE-DPD allows a user to detect patterns, exploiting existing rules or new ones, and to train some clusterer and classifier algorithms to learn what a good pattern instance is and what is not. The training consists of submitting to the algorithms a set of example pattern instances, which have been labeled as correct or incorrect, to let them build an internal model able to perform their evaluations. New pattern instances can then be automatically evaluated without having been inspected by the user. The automatic evaluation gives a confidence value to each pattern instance, which measures how much an instance is compliant with the set of examples submitted in the training phase.

In this thesis the design pattern detection area has been analyzed from different perspectives. A review of the different approaches for design pattern detection highlighted that only few existing approaches try to overcome the limitations of exact matching techniques. By exploiting some kind of soft computing or machine learning techniques, which are designed to approximate the optimal solution of decision problems, it is possible to address the design pattern definition informality issue. The four reviewed approaches applying inexact matching exploit different techniques to solve the problem: machine learning, similarity scoring, and constraint relaxation. The machine learning approach most similar to the one described in this thesis is the one from Ferenc et al. [69], then extended by Fülöp [72]. In fact, it is the only approach using supervised classification to filter out wrong instances from a set of candidates. The apparent major limitation of that approach is that patterns are represented by features that mask their structure, while the approach introduced in this thesis employs lower level features, i.e. micro-structures. Similarity scoring does not involve the subjective view of the developer into the detection, while constraint relaxation is not able to learn from the choices made by the user when deciding to relax or not the rule. The other machine learning approach, which did not use exact matching for the extraction of patterns, was limited to represent one role at a time in the dataset, cutting the possibility to show the relationship among different the roles to the classifier algorithm. Given the requirements the usage of a classification as a filter is a better solution.

The modeling aspect has a great relevance in reverse engineering, so a review of the available models for reverse engineering was performed. It turned out that, despite different good solutions are available, no one appears to be going to become a de facto standard. The only available

model for design pattern detection results, DPDX [119], was reported, discussed and compared to the model existing in MARPLE-DPD for the same purpose. DPDX has a different goal than the model employed in MARPLE-DPD. In fact, DPDX is wider and more complete, and is designed for the exchange of information among research groups, while the MARPLE-DPD model has the only goal of representing the structure of design pattern instances. One of the information available in DPDX and not in MARPLE-DPD, for example is the justification of the role assignments. Finally, DPDX lacks a meta-meta-model definition able to express the grammar of pattern definitions; this limitation can be problematic for users writing new pattern definitions and wanting to test the compliance of the definition with the model.

One of the central points of the model integrated in MARPLE for the representation of an analyzed system is the concept of micro-structure. A micro-structure is a fact regarding one or two classes, having the property of being mechanically recognizable from the source code. This property makes micro-structures very useful for the abstraction of software systems. MARPLE supports three different families of micro-structures: elemental design patterns, micro patterns and design pattern clue. Each of these have been discussed and analyzed.

Micro-structures are also the base for the pattern detection facilities in MARPLE-DPD. In fact, the Joiner module exploits micro-structures as edges in the graph it uses to represent the software system; the nodes are represented by the classes. Patterns are extracted by matching query rules against the graph, and then grouping the extracted role-class mappings in tree-wise structures representing pattern instances. The grouping (called "merge" in the Joiner) procedure is one of the most important parts of the entire approach, because it allows to define the roles identifying a pattern instance and their dependencies. The Joiner rules for all the design patterns of the GoF book have been defined, and the rules for the five design pattern which had the deeper experiments are discussed. In particular, many issues regarding the definition of the patterns are highlighted, and some proposals to overcome these issues are made. For example, the *Adapter* design pattern was found to be pervasive in software systems, when allowing all the possible interpretations available in the literature and in the analyzed system. A proposal for a different and formal interpretation of the *Adapter* design pattern has been then introduced, able to express the amount of adaptation performed by each class. The same rationale can be applied also to the *Decorator* and the other patterns defining an object indirection protocol. The rules defined for the Joiner were written and tested to maximize the number of detected patterns, admitting to retrieve many false positives. This criterion is based on the fact that the classification phase will filter out false positives, but it cannot extract correct instances from the system by itself. The Classifier module assumes the Joiner to have 100% recall.

The classification approach was then described, and the modeling aspect of the supervised classification problem was discussed. The proposed solution is to split the process in two phases, the first exploiting clustering and the second supervised classification. The clustering phase transforms a dataset composed of the role-class mappings in a dataset representing the instances. The first dataset uses a combination of roles and micro-structures as features, while the second uses the information coming from the clustering phase to represent the pattern instances. The advantage of this approach is that it is able to represent pattern instances composed of an arbitrary number of classes as feature vectors of fixed size, which can be consumed by data mining algorithms. A major enhancement to the process is the special handling of patterns modeled with a single level. This kinds of pattern instances are composed of only one mapping. This property enables the direct usage of supervised classification on the first kind of dataset, allowing the classification algorithm to build its model on a representation that is more close to the system.

The classification process was tested through a set of experiments on five design patterns: *Singleton*, *Adapter*, *Composite*, *Decorator* and *Factory Method*. The patterns were extracted

from a set of systems composed of the ones contained in the PMARt dataset plus one system containing example pattern instances. Many clustering and classification techniques, limiting to the ones having an implementation in the Weka framework, have been experimented on the five design patterns, with different results. More precisely, when clustering techniques are applied, the performance of the approach is lower than when applying only classification techniques. In the latter case, instead good performance values were achieved. The lower performances of the clustering process (than the direct classification one) can be caused by different factors. To have a confirmation of the performance degrade it will be necessary to be able to apply some of the clustering algorithms that were not runnable in the time limits, due to their implementation and to the limitations of computational resources. The best performances were obtained on the *Singleton* design pattern, which is a single-level pattern. The performances for the *Adapter* are only a little lower: *Singleton* scores ~0.9 of f-measure and ~1.0 of area under ROC, and *Adapter* ~0.8 and ~0.9, respectively, which are both good results. The best result on multi-level patterns is the one obtained on the *Factory Method*, which reached values similar to the ones of the *Adapter*, but ~0.05 lower. The performances for the *Decorator* are slightly lower and the ones for the *Composite* are not good. One of the outcomes of the experiments on these three patterns is that there is an apparent increase in the performances when the size of the dataset grows; moreover, as the dataset grows, also the difference between the two clusterer algorithms becomes relevant, with SimpleKMeans achieving better performances than CLOPE. The hypothesis of the correlation of the size of the dataset and the performances should be tested with the approach of learning curves [151].

Some problems raised through the manual evaluation and labeling of the results coming from the Joiner module. In particular, many of the instances reported in the PMARt dataset were not considered correct ones, and many others were not present in the dataset. If the second problem is normal in a manually validated dataset, the first one is more serious, tackling the validity of the dataset itself.

Finally, the described approach demonstrated that an approach for the detection of design patterns exploiting data mining techniques can be successfully applied, despite the encountered technical limitations.

9.1 Future work

Many future works can be planned, and many were advised or described in the content of the thesis. Next paragraphs will summary a selection of the most relevant future developments.

Micro-structures catalog The micro-structures catalog is not meant to be complete or perfect. Many new kinds of information can be specified, and others may be removed if they are not enough significant. A review of the catalog contents will be performed and new experiments will be done using the newer available information. Another task related to the catalog will be to test (empirically) the significance of the found micro-structures, in the same or similar way Gil and Maman did for micro patterns [76].

Micro-structures granularity The current implementation of the Micro Structure Detector reports micro-structures at the class level, i.e. information given by micro-structures is reported referring to classes. Some of this information should be reported at method or attribute level, leading to a more precise description of the underlying system. A more precise description will allow writing better Joiner rules, and it should imply better performances of the Classifier module.

Software architecture reconstruction integration MARPLE has architecture reconstruction facilities able to show graphs representing a software system, exploiting the micro-structures. As the model is enriched with the design pattern instances, the views will benefit from the integration of that information, which will augment the value of both the pattern detection and the architecture reconstruction modules.

Design pattern instances pruning The pattern instances retrieved by the Joiner are modeled as trees of levels. The trees have the property of allowing the deletion of intermediate nodes without losing their correctness. The only limit is that, after the pruning, at least one class must play every role of the pattern. In future work, the user interface of MARPLE will allow the users delete levels from the pattern instances, permitting the cleaning of partially correct ones.

Patterns interconnections Design patterns often exploited in combination with other ones. The representation of the connection among patterns will be useful to enhance the detection, and will have some application also in the SAR module: for example, when two patterns are connected and they have complementary goals, they can be shown as a single pattern or cluster of patterns.

Variants or alternatives handling In some cases the same pattern has different structural alternatives, where the number and type of roles are not equal. If the different alternatives share the same intent, it will be possible to code the relationship in the pattern definitions, allowing the user interface to show the different alternatives under the same name, enhancing the user perception of the results.

Libraries The handling of the connection among systems and libraries is one of the planned future works. The plan is to create an analysis repository for each library, and to recall it when a library is used by the analyzed system. The approach will permit to reuse existing analysis and to discover new usages of the patterns implemented in the libraries.

New patterns One of the most obvious future works is to extend the experiments to all the other design pattern, which are reported in Appendix A. The new experiments will benefit from the experience gained with the experiments reported in this thesis, and errors due to algorithm implementations can be avoided. The results on all the patterns will allow better comparison with other results coming from different tools, exploiting for example the comparison functionalities of our Design Pattern Benchmark platform [15].

Clustering algorithm Given the results obtained through the clustering algorithms, one of the most important future developments is to test better clustering algorithm implementations, adapting them to the Weka interface if necessary. An interesting technique to try will be Self Organizing Maps[1].

Dataset representation A new experiment to perform will be to represent also metrics in the datasets, and to represent micro-structures using their frequency values, absolute or relative, or to try more sophisticated frequency measures, like the TF-IDF [162] one.

[1]The implementation used for the experimentations gave bad results. It needs more testing at least.

Feature grouping The current feature setup is not able to express the possibility of grouping different features to create more general ones. The grouping feature can be useful if the dataset representation that uses the basic features is too sparse, and can enhance both the time and results performances of the classification phase.

Experimenter plugin The Weka Experimenter module is a great source of information about the significance of the performance values for classification algorithms. It allows to have a better informed view of the real reliability of the trained models and to sensibly rank different models. Unfortunately, only classification is supported as model scheme. A future work will be to provide a plugin for the Weka Experimenter to allow the evaluation of the full clusterer-classifier trained models, enabling the statistical analysis of the performances for the entire approach. One possibility will be to implement a meta-classifier that will mask the complete tool chain; another one will be to plug new weka.experiment. ResultListener or weka.experiment.ResultProducer classes to allow this functionality.

If this kind of work will be too complicated, because it changes the way the Experimenter is designed, another possibility would be to modify the optimization algorithm to produce a fully-compliant Experimenter result file. During the experiments a partially compliant exporter was developed, but it produced too large and unmanageable files.

Weka enhancement Many of the issues related to Weka during the experiments, especially in the Experimenter module, were related to observable design or implementation flaws in the code (which is open-source). Weka is becoming very popular in the research, and a contribution from the software engineering perspective could help to enhance a good initiative like it is, that brings data mining techniques to Java systems in a free and open manner.

Full automation The effort in the last two points is in the direction of achieving *full automation* in the experiments, removing the issues related to the manual results collection. The only human-driven phases should be the evaluation of instances and the interpretation of the experimental results.

Extensive comparison To better validate the approach, an extensive comparison with other tools, repositories and techniques will be performed, exploiting also the DPB [65] platform. The comparison will be performed against a number of system and instances that will make it statistically significant. A preliminary comparison against only one system is available in Appendix C.

Appendix A

Joiner rules for non-experimented patterns

A.1 Creational Design Patterns

A.1.1 Abstract Factory

Match rule The match rule for the *Abstract Factory* design pattern addresses only the basic concepts regarding the pattern definition. The constraints are related to the two hierarchies of factories and products, to the abstractness of AbstractFactory and AbstractProduct, and to the declarations of the returned products by the factories. Every other constraint will be tested by the classifier module, because *Abstract Factory* is a generic pattern allowing many different variants, like factory methods or prototypes as concrete factories.

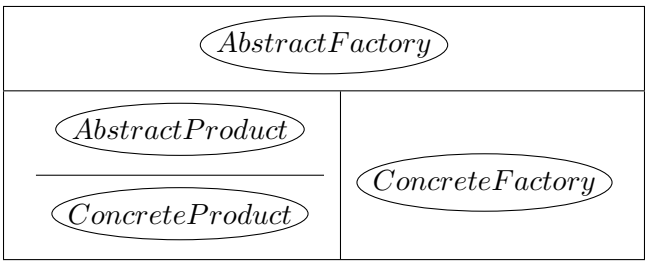
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?AbstractFactory ?ConcreteFactory ?AbstractProduct ?ConcreteProduct
WHERE {
?AbstractFactory
  BE:AbstractType ?AbstractFactory ;
  BE:ProductReturns ?AbstractProduct .
?ConcreteFactory
  BE:ExtendedInheritance ?AbstractFactory ;
  BE:ProductReturns ?AbstractProduct .
?AbstractProduct BE:AbstractType ?AbstractProduct .
?ConcreteProduct BE:ExtendedInheritance ?AbstractProduct .
NOT EXISTS {?AbstractFactory BE:SameClass ?AbstractProduct .}
}
```

Merge rule The merge rule describes an *Abstract Factory* as an AbstractFactory class having some AbstractProducts and some ConreteFactory classes realizing it. Each AbstractProducts is realized by some ConcreteProduct.

```
<pattern name="Abstract Factory">
  <role name="AbstractFactory" />
  <sublevel>
    <role name="ConcreteFactory" />
  </sublevel>
  <sublevel>
    <role name="AbstractProduct" />
    <sublevel>
```

```
      <role name="ConcreteProduct" />
    </sublevel>
  </sublevel>
</pattern>
```

Merge rule diagram



A.1.2 Builder

Match rule The match rule for the *Builder* design pattern focuses on two elements Director and Builder must be linked my method calls, but they must be unrelated classes, and the ConcreteBuilder realizing the Builder must be able to return the Product.

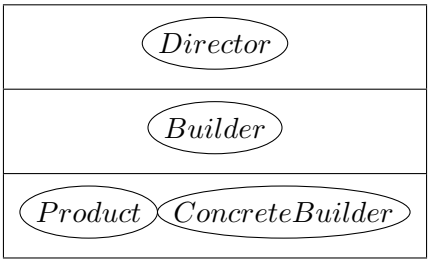
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Director ?Builder ?Product ?ConcreteBuilder
WHERE {
{{?Director BE:Delegate ?Builder}
UNION {?Director BE:Redirect ?Builder}
UNION {?Director BE:DelegateInLimitedFamily ?Builder}
UNION {?Director BE:RedirectInLimitedFamily ?Builder}}.

?ConcreteBuilder
  BE:ExtendedInheritance ?Builder ;
  BE:ProductReturns ?Product .

NOT EXISTS { ?Director BE:CreateObject ?ConcreteBuilder . }
NOT EXISTS { ?Director BE:ExtendedInheritance ?Builder . }
NOT EXISTS { ?Builder BE:ExtendedInheritance ?Director . }
NOT EXISTS { ?Builder BE:SameClass ?Director . }
NOT EXISTS { ?Product BE:ExtendedInheritance ?Director . }
NOT EXISTS { ?Product BE:ExtendedInheritance ?ConcreteBuilder . }
NOT EXISTS { ?Product BE:ExtendedInheritance ?Builder . }
NOT EXISTS { ?Product BE:SameClass ?Director . }
NOT EXISTS { ?Product BE:SameClass ?ConcreteBuilder . }
NOT EXISTS { ?Product BE:SameClass ?Builder . }
NOT EXISTS { ?ConcreteBuilder BE:Stateless ?ConcreteBuilder . }
}
```

Merge rule The merge rule tells that a *Builder* pattern instance is identified by its Director, which manages potentially more than one Builder. Each Builder is realized by some ConcreteBuilder, each one able to produce one Product kind.

```
<pattern name="Builder">
  <role name="Director" />
  <sublevel>
```

```
    <role name="Builder" />
    <sublevel>
      <role name="Product" />
      <role name="ConcreteBuilder" />
    </sublevel>
  </sublevel>
</pattern>
```

Merge rule diagram



A.1.3 Prototype

Match rule The match rule for the *Prototype* pattern specifies two main variants: classes creating instances of theirselves and returning them declared as a superclass, and classes that implement the java.lang.Cloneable interface, following the Java framework.

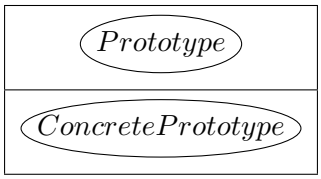
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Prototype ?ConcretePrototype
WHERE {
{{?Prototype BE:ProductReturns ?Prototype .
?ConcretePrototype BE:ExtendedInheritance ?Prototype ;
  BE:CreateObject ?ConcretePrototype.
{
    {?ConcretePrototype BE:ProductReturns ?ConcretePrototype.}
    UNION
    {?ConcretePrototype BE:ProductReturns ?Prototype}
}.
}UNION{
?Prototype BE:CloneableImplemented ?Prototype .
{
    {?ConcretePrototype BE:ExtendedInheritance ?Prototype.}
    UNION
    {?ConcretePrototype BE:SameClass ?Prototype.
    OPTIONAL {?x BE:ExtendedInheritance ?Prototype.}
    FILTER(!bound(?x))
    }
}.
}}.
}
```

Merge rule In the *Prototype* pattern there are only two roles, one child of the other. The ConcretePrototype is in a sublevel in order to group all the ConcretePrototypes for each *Prototype*.

```
<pattern name="Prototype">
  <role name="Prototype" />
```

```
  <sublevel>
    <role name="ConcretePrototype" />
  </sublevel>
</pattern>
```

Merge rule diagram



A.2 Structural Design Patterns

A.2.1 Bridge

The match rule for the *Bridge* design pattern looks for two separate hierarchies, connected through a reference from Abstraction or RefinedAbstraction to Implementor and wants the RefinedAbstraction to use the Implementor.

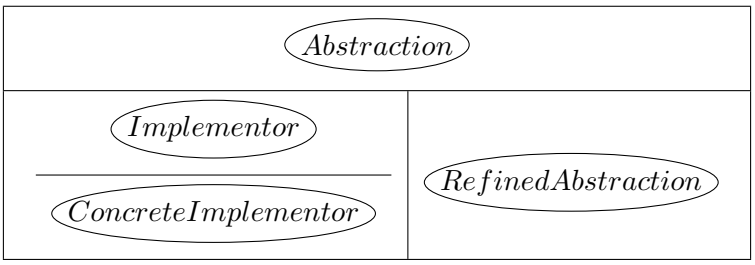
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Abstraction ?Implementor ?ConcreteImplementor ?RefinedAbstraction
WHERE {
?ConcreteImplementor BE:ExtendedInheritance ?Implementor.
?ConcreteImplementor BE:OverridingMethod" ?Implementor.

?RefinedAbstraction BE:ExtendedInheritance ?Abstraction.

{
{?Abstraction BE:OtherStaticReference ?Implementor.}
UNION{?Abstraction BE:PrivateInstanceReference ?Implementor.}
UNION{?Abstraction BE:PrivateStaticReference ?Implementor.}
UNION{?Abstraction BE:OtherInstanceReference ?Implementor.}
UNION{?Abstraction BE:ProtectedInstanceReference ?Implementor.}
UNION{?Abstraction BE:ProtectedStaticReference ?Implementor.}
UNION{?RefinedAbstraction BE:OtherStaticReference ?Implementor.}
UNION{?RefinedAbstraction BE:PrivateInstanceReference ?Implementor.}
UNION{?RefinedAbstraction BE:PrivateStaticReference ?Implementor.}
UNION{?RefinedAbstraction BE:OtherInstanceReference ?Implementor.}
UNION{?RefinedAbstraction BE:ProtectedInstanceReference ?Implementor.}
UNION{?RefinedAbstraction BE:ProtectedStaticReference ?Implementor.}
}.

{
{?RefinedAbstraction BE:Conglomeration ?Implementor.}
UNION{?RefinedAbstraction BE:Delegate ?Implementor.}
UNION{?RefinedAbstraction BE:DelegatedConglomeration ?Implementor.}
UNION{?RefinedAbstraction BE:DelegateInFamily ?Implementor.}
UNION{?RefinedAbstraction BE:DelegateInLimitedFamily ?Implementor.}
UNION{?RefinedAbstraction BE:ExtendMethod ?Implementor.}
UNION{?RefinedAbstraction BE:Recursion ?Implementor.}
UNION{?RefinedAbstraction BE:Redirect ?Implementor.}
```

```
UNION{?RefinedAbstraction␣BE:RedirectInFamily␣?Implementor.}
UNION{?RefinedAbstraction␣BE:RedirectInLimitedFamily␣?Implementor.}
UNION{?RefinedAbstraction␣BE:RedirectRecursion␣?Implementor.}
UNION{?RefinedAbstraction␣BE:RevertMethod␣?Implementor.}
}.

NOT␣EXISTS{?Abstraction␣BE:SameClass␣?Implementor.}
NOT␣EXISTS{?Abstraction␣BE:SameClass␣?ConcreteImplementor.}
NOT␣EXISTS{?Implementor␣BE:SameClass␣?RefinedAbstraction.}
NOT␣EXISTS{?ConcreteImplementor␣BE:SameClass␣?RefinedAbstraction.}
NOT␣EXISTS{?RefinedAbstraction␣␣BE:ExtendedInheritance␣?Implementor.}
NOT␣EXISTS{?RefinedAbstraction␣␣BE:SameClass␣?Implementor.}
NOT␣EXISTS{?RefinedAbstraction␣␣BE:SameClass␣?ConcreteImplementor.}
}
```

Merge rule The merge rule tells that the *Bridge* is identified by its Abstraction class, which manages a number of Implementors and has some realization as RefinedAbstraction. Each Implementor has some realization.

```
<pattern name="Bridge">
  <role name="Abstraction" />
  <sublevel>
    <role name="Implementor" />
    <sublevel>
      <role name="ConcreteImplementor" />
    </sublevel>
  </sublevel>
  <sublevel>
    <role name="RefinedAbstraction" />
  </sublevel>
</pattern>
```

Merge rule diagram



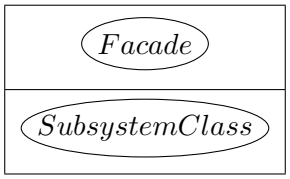
A.2.2 Facade

```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT  ?Facade ?SubsystemClass
WHERE {
{{ ?Facade  BE:Delegate  ?SubsystemClass  }
UNION { ?Facade  BE:Redirect  ?SubsystemClass  }
UNION { ?Facade  BE:Conglomeration  ?SubsystemClass  }
UNION { ?Facade  BE:Recursion  ?SubsystemClass  }
```

```
UNION { ?Facade  BE:RevertMethod  ?SubsystemClass  }
UNION { ?Facade  BE:ExtendMethod  ?SubsystemClass  }
UNION { ?Facade  BE:DelegatedConglomeration  ?SubsystemClass  }
UNION { ?Facade  BE:RedirectRecursion  ?SubsystemClass  }
UNION { ?Facade  BE:DelegateInFamily  ?SubsystemClass  }
UNION { ?Facade  BE:RedirectInFamily  ?SubsystemClass  }
UNION { ?Facade  BE:DelegateInLimitedFamily  ?SubsystemClass  }
UNION { ?Facade  BE:RedirectInLimitedFamily  ?SubsystemClass  }}.

NOT EXISTS { ?Facade  BE:ExtendedInheritance  ?SubsystemClass . }
NOT EXISTS { ?SubsystemClass  BE:ExtendedInheritance  ?Facade . }

NOT EXISTS { ?SubsystemClass  BE:Delegate  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:Redirect  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:Conglomeration  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:Recursion  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:RevertMethod  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:ExtendMethod  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:DelegatedConglomeration  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:RedirectRecursion  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:DelegateInFamily  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:RedirectInFamily  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:DelegateInLimitedFamily  ?Facade . }
NOT EXISTS { ?SubsystemClass  BE:RedirectInLimitedFamily  ?Facade . }
}
```

```
<pattern name="Facade">
  <role name="Facade" />
  <sublevel>
    <role name="SubsystemClass" />
  </sublevel>
</pattern>
```

Merge rule diagram

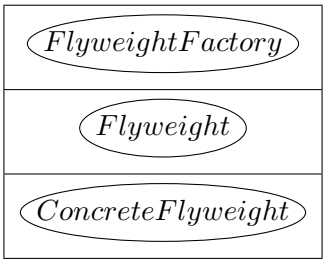


A.2.3 Flyweight

Complete rule The rule for the *Flyweight* pattern is completely specified using the XML syntax. Only little constraints are specified: the FlyweightFactory must create at least a ConcreteFlyweight instance and there must exist an inheritance of Flyweight and ConcreteFlyweight roles. Both the roles may not make method calls to FlyweightFactory, i.e. they cannot know about the existence of a factory managing them. The structure of the roles tells that the FlyweightFactory identifies the pattern, and it can manage many Flyweight types, which can be realized by many ConcreteFlyweight types.

```
<pattern name="Flyweight">
  <role name="FlyweightFactory">
    <be name="CreateObject" to="ConcreteFlyweight" />
  </role>
  <sublevel>
    <role name="Flyweight">
      <be name="ExtendedInheritance" to="FlyweightFactory" negation="true" />
      <be name="Delegate" to="FlyweightFactory" negation="true" />
    </role>
    <sublevel>
      <role name="ConcreteFlyweight">
        <be name="ExtendedInheritance" to="Flyweight" />
        <be name="Delegate" to="FlyweightFactory" negation="true" />
      </role>
    </sublevel>
  </sublevel>
</pattern>
```

Merge rule diagram

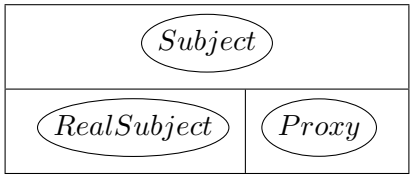


A.2.4 Proxy

The rule for the *Proxy* pattern is specified using the XML syntax. The rule defines the inheritance among Subject and {RealSubject, Proxy} roles. Proxy must keep a reference to the Subject or RealSubject and make a call to RealSubject. The pattern is identified by the Subject, which can be realized by many RealSubject and Proxy instances.

```
<pattern name="Proxy">
  <role name="Subject">
    <be name="Delegate" to="Proxy" negation="true" />
  </role>
  <sublevel>
    <role name="RealSubject">
      <be name="ExtendedInheritance" to="Subject" />
      <be name="ExtendedInheritance" to="Proxy" negation="true" />
    </role>
  </sublevel>
  <sublevel>
    <role name="Proxy">
      <be name="ExtendedInheritance" to="Subject" />
      <be name="SameHierarchyObject" to="Proxy" />
      <be name="RedirectInLimitedFamily" to="RealSubject" />
    </role>
  </sublevel>
</pattern>
```

Merge rule diagram



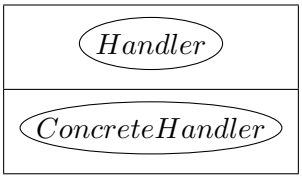
A.3 Behavioral Design Patterns

A.3.1 Chain of Responsibility

The rule for the *Chain of Responsibility* pattern is very simple. It defines the inheritance tree of Handler and ConcreteHandler and that the ConcreteHandler must make a call to the Handler, which must have at least an abstract method. The way the ConcreteHandler reaches the Handler is not defined. The structure of the pattern is simple: for each Handler, many ConcreteHandler may exist.

```
<pattern name="ChainOfResponsibility">
  <role name="Handler">
    <be name="AbstractInterface" to="Handler" />
  </role>
  <sublevel>
    <role name="ConcreteHandler">
      <be name="ExtendedInheritance" to="Handler" />
      <be name="RedirectInFamily" to="Handler" />
    </role>
  </sublevel>
</pattern>
```

Merge rule diagram



A.3.2 Command

Match rule In the *Command* pattern the match rule cares about the ordering of the method calls among the roles, and the division between Commands (abstract and concrete) and the other roles. In fact, Commands may not extend Receivers or Invokers, and vice versa. In order to have a decent pattern implementation the Receiver must not call the ConcreteCommand, otherwise it means that the internal command layer is known to the implementation of the system.

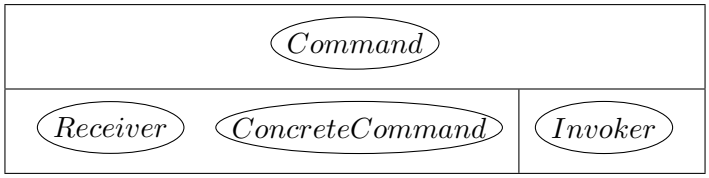
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Command ?Receiver ?ConcreteCommand ?Invoker
WHERE {
```

```
?Command BE:AbstractInterface ?Command .
?ConcreteCommand BE:ExtendedInheritance ?Command .
?Invoker BE:AbstractMethodInvoked ?Command .
{
      { ?ConcreteCommand BE:Delegate ?Receiver }
UNION { ?ConcreteCommand BE:Redirect ?Receiver }
UNION { ?ConcreteCommand BE:Conglomeration ?Receiver }
UNION { ?ConcreteCommand BE:Recursion ?Receiver }
UNION { ?ConcreteCommand BE:RevertMethod ?Receiver }
UNION { ?ConcreteCommand BE:ExtendMethod ?Receiver }
UNION { ?ConcreteCommand BE:DelegatedConglomeration ?Receiver }
UNION { ?ConcreteCommand BE:RedirectRecursion ?Receiver }
UNION { ?ConcreteCommand BE:DelegateInFamily ?Receiver }
UNION { ?ConcreteCommand BE:RedirectInFamily ?Receiver }
UNION { ?ConcreteCommand BE:DelegateInLimitedFamily ?Receiver }
UNION { ?ConcreteCommand BE:RedirectInLimitedFamily ?Receiver }
}.
NOT EXISTS { ?Command BE:ExtendedInheritance ?Receiver . }
NOT EXISTS { ?Command BE:ExtendedInheritance ?Invoker . }

NOT EXISTS { ?Receiver BE:Delegate ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:Redirect ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:Conglomeration ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:Recursion ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:RevertMethod ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:ExtendMethod ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:DelegatedConglomeration ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:RedirectRecursion ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:DelegateInFamily ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:RedirectInFamily ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:DelegateInLimitedFamily ?ConcreteCommand . }
NOT EXISTS { ?Receiver BE:RedirectInLimitedFamily ?ConcreteCommand . }

NOT EXISTS { ?Invoker BE:ExtendedInheritance ?Command . }
NOT EXISTS { ?Invoker BE:ExtendedInheritance ?ConcreteCommand . }
}
```

Merge rule The *Command* pattern encapsulates actions into objects, exposing only a generic interface and permitting the choice of the real action to do at runtime. Of consequence, each Command may be used by many Invokers with many implementations (unknown to the invoker) that forward to specific Receivers.

```
<pattern name="Command">
  <role name="Command" />
  <sublevel>
    <role name="Receiver" />
    <role name="ConcreteCommand" />
  </sublevel>
  <sublevel>
    <role name="Invoker" />
  </sublevel>
</pattern>
```

Merge rule diagram

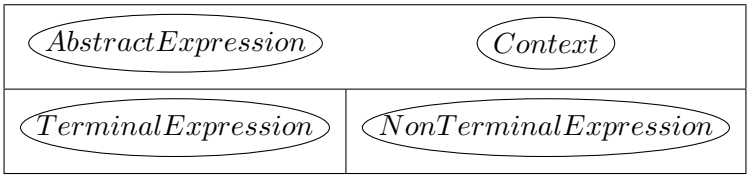


A.3.3 Interpreter

The rule for the *Interpreter* design pattern defines first the inheritance tree of the expressions and forbids the Context to be in the same inheritance tree of the expressions. The expressions must receive the Context as a method parameter. The rule tells that there is one Context for each AbstractExpression, and many TerminalExpression and NonTerminalExpression for each AbstractExpression.

```
<pattern name="Interpreter">
  <role name="AbstractExpression">
    <be name="ExtendedInheritance" to="Context" negation="true" />
    <be name="SameClass" to="Context" negation="true" />
    <be name="ReceivesParameter" to="Context" />
  </role>
  <role name="Context">
    <be name="ExtendedInheritance" to="AbstractExpression" negation="true" />
  </role>
  <sublevel>
    <role name="TerminalExpression">
      <be name="ExtendedInheritance" to="AbstractExpression" />
      <be name="ExtendedInheritance" to="NonTerminalExpression"
        negation="true" />
      <be name="ReceivesParameter" to="Context" />
      <be name="RedirectInFamily" to="AbstractExpression" negation="true" />
    </role>
  </sublevel>
  <sublevel>
    <role name="NonTerminalExpression">
      <be name="ExtendedInheritance" to="AbstractExpression" />
      <be name="RedirectInFamily" to="AbstractExpression" />
      <be name="ReceivesParameter" to="Context" />
    </role>
  </sublevel>
</pattern>
```

Merge rule diagram



A.3.4 Iterator

Match rule The match rule for the *Iterator* design pattern defines two inheritances of aggregators and iterators, which must be disconnected from one another, and looks for Iterators

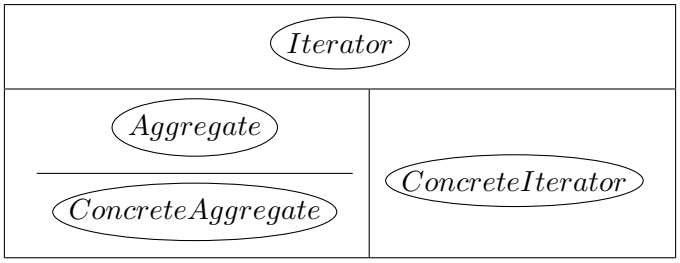
which define all methods with no parameters (the StateMachine clause). The ConcreteIterator must call the ConcreteAggregate to get the next value or to test the stop condition.

```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT  ?Iterator ?Aggregate ?ConcreteAggregate ?ConcreteIterator
WHERE {

?Iterator BE:StateMachine ?Iterator.
?Aggregate BE:ProductReturns ?Iterator.
?ConcreteAggregate BE:ExtendedInheritance ?Aggregate.
?ConcreteIterator BE:ExtendedInheritance ?Iterator.

{{?ConcreteIterator BE:Conglomeration ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:Delegate ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:DelegateInFamily ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:DelegateInLimitedFamily ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:DelegatedConglomeration ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:ExtendMethod ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:Recursion ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:Redirect ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:RedirectInFamily ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:RedirectInLimitedFamily ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:RedirectRecursion ?ConcreteAggregate.}
UNION{?ConcreteIterator BE:RevertMethod ?ConcreteAggregate.}}.

NOT EXISTS {?Iterator BE:ExtendedInheritance  ?Aggregate.}.
NOT EXISTS {?Aggregate BE:ExtendedInheritance  ?Iterator.}.
}
```

Merge rule The merge rule defines that an Iterator can manage different type of Aggregates, which can be realized by more than one ConcreteAggregate. Each Iterator can be realized by many ConcreteIterator types, not necessarily one for each ConcreteAggregate.

```
<pattern name="Iterator">
  <role name="Iterator" />
  <sublevel>
    <role name="Aggregate" />
    <sublevel>
      <role name="ConcreteAggregate" />
    </sublevel>
  </sublevel>
  <sublevel>
    <role name="ConcreteIterator" />
  </sublevel>
</pattern>
```

Merge rule diagram

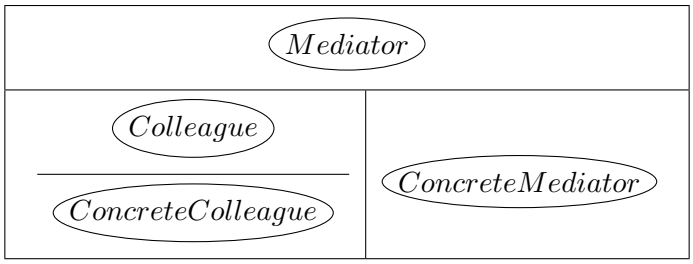


A.3.5 Mediator

Complete rule The *Mediator* detection rule is defined using the XML syntax. Not many constraints are defined. Basically the inheritance trees of mediators and colleagues are defined, and the crossing calls among mediators and colleagues are required. The Mediator role identifies the pattern, which can manage many Colleague types. Mediator and Colleague types can be realized by many ConcreteMediator and ConcreteColleague ones.

```
<pattern name="Mediator">
  <role name="Mediator">
    <be name="ExtendedInheritance" to="Colleague" negation="true"/>
  </role>
  <sublevel>
    <role name="Colleague">
    </role>
    <sublevel>
      <role name="ConcreteColleague">
        <be name="ExtendedInheritance" to="Colleague" />
        <be name="Delegate" to="Mediator" />
      </role>
    </sublevel>
  </sublevel>
  <sublevel>
    <role name="ConcreteMediator">
      <be name="ExtendedInheritance" to="Mediator" />
      <be name="Delegate" to="ConcreteColleague" />
    </role>
  </sublevel>
</pdef:pattern>
```

Merge rule diagram

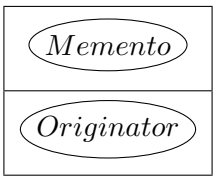


A.3.6 Memento

Complete rule *Memento* is a pattern defined by a rule telling that Memento and Originator must by unrelated and that the Originator is a factory for the Memento, which is not Stateless and does not propagate calls to other classes. This is a single-level pattern; every instance is identified by the Memento-Originator pair.

```
<pattern name="Memento">
  <role name="Memento">
    <be name="ExtendedInheritance" to="Originator" negation="true" />
    <be name="Sink" to="Memento" />
    <be name="Stateless" to="Memento" negation="true"/>
  </role>
  <role name="Originator">
    <be name="ExtendedInheritance" to="Memento" negation="true" />
    <be name="ProductReturns" to="Memento" />
    <be name="CreateObject" to="Memento" />
  </role>
</pattern>
```

Merge rule diagram



A.3.7 Observer

Match rule The match rule for the *Observer* pattern allows the abstract and concrete roles to collapse, while the Subject and Observer may not be covered by the same class. The other constraints specify that the ConcreteObserver makes the notification and that the Subject exposes a way to register the Observers.

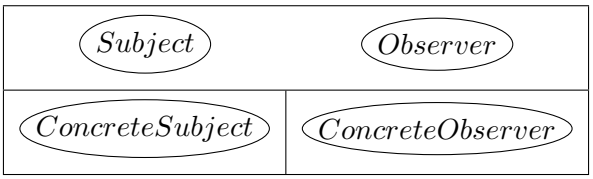
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT  ?Subject ?Observer ?ConcreteSubject ?ConcreteObserver
WHERE {
?Subject BE:ReceivesParameter  ?Observer .

{{?ConcreteObserver BE:ExtendedInheritance ?Observer.}
UNION { ?ConcreteObserver BE:SameClass ?Observer.
    OPTIONAL { ?ConcreteObserver BE:ExtendedInheritance ?x.
        FILTER(?x != ?Observer) }
    FILTER(!bound(?x))
}}
{{?ConcreteSubject BE:ExtendedInheritance ?Subject.}
UNION {
    ?ConcreteSubject BE:SameClass ?Subject.
    OPTIONAL { ?x BE:ReceivesParameter  ?Observer.
        ?ConcreteSubject BE:ExtendedInheritance ?x.
        FILTER(?x != ?Subject)
    }
    FILTER(!bound(?x))
}}
{     { ?ConcreteSubject BE:Delegate ?Observer}
UNION { ?ConcreteSubject BE:Redirect ?Observer}
UNION { ?ConcreteSubject BE:Conglomeration ?Observer}
UNION { ?ConcreteSubject BE:Recursion ?Observer}
UNION { ?ConcreteSubject BE:RevertMethod ?Observer}
UNION { ?ConcreteSubject BE:ExtendMethod ?Observer}
```

```
UNION { ?ConcreteSubject BE:DelegatedConglomeration ?Observer}
UNION { ?ConcreteSubject BE:RedirectRecursion ?Observer}
UNION { ?ConcreteSubject BE:DelegateInFamily ?Observer}
UNION { ?ConcreteSubject BE:RedirectInFamily ?Observer}
UNION { ?ConcreteSubject BE:DelegateInLimitedFamily ?Observer}
UNION { ?ConcreteSubject BE:RedirectInLimitedFamily ?Observer}}

NOT EXISTS { ?Subject BE:ExtendedInheritance ?Observer . }
NOT EXISTS { ?Observer BE:ExtendedInheritance ?Subject . }
NOT EXISTS { ?Subject BE:SameClass ?Observer . }
NOT EXISTS { ?ConcreteObserver BE:Interface ?ConcreteObserver . }}
```

Merge rule Each instance of the *Observer* pattern is identified by a couple of Subject and Observer roles. Each of them can have a concrete implementation.

```
<pattern name="Observer">
  <role name="Subject" />
  <role name="Observer" />
  <sublevel>
    <role name="ConcreteSubject" />
  </sublevel>
  <sublevel>
    <role name="ConcreteObserver" />
  </sublevel>
</pattern>
```

Merge rule diagram



A.3.8 State

Match rule The match rule for the *State* pattern describes a Context class that may not be immutable and that calls methods belonging to a State class (which is not in its hierarchy), which is abstract and has at least one concrete implementation.

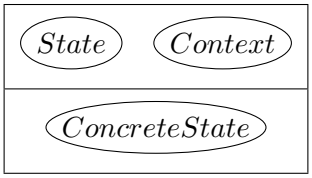
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?State ?Context ?ConcreteState
WHERE {
?ConcreteState BE:ExtendedInheritance ?State.

{{ ?Context BE:Delegate ?State}
UNION { ?Context BE:Redirect ?State}
UNION { ?Context BE:DelegateInLimitedFamily ?State}
UNION { ?Context BE:RedirectInLimitedFamily ?State}}

OPTIONAL {?x BE:CreateObject ?State.} FILTER(!bound(?x))

NOT EXISTS { ?Context BE:ExtendedInheritance ?State .}
NOT EXISTS { ?Context BE:Immutable ?Context .}
```

}

Merge rule One *State* pattern instance is identified by a State class and its Context. There can be many concrete implementations of the State.

```
<pattern name="State">
  <role name="State" />
  <role name="Context" />
  <sublevel>
    <role name="ConcreteState">
      <be name="ExtendedInheritance" to="State" />
    </role>
  </sublevel>
</pattern>
```

Merge rule diagram



A.3.9 Strategy

The *Strategy* pattern is a very general one. Its detection rule is very similar to the one of the *State* pattern, but many other patterns can be seen as specializations of *Strategy*: *Builder* and *Factory Method*, for example define two different way of implementing a strategy for object instantiation, and *Visitor* defines a strategy for handling different kind of elements of a data structure.

Match rule In the match rule the *Strategy* pattern is defined as an inheritance of strategies, whose abstract class is used by a Context. The Context must keep a reference to the Strategy or receive it as a parameter, and it has to call it.

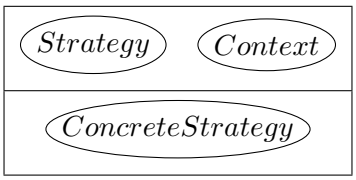
```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT ?Strategy ?Context ?ConcreteStrategy
WHERE {
?ConcreteStrategy BE:ExtendedInheritance ?Strategy.

{{ ?Context BE:ReceivesParameter ?Strategy}
UNION { ?Context BE:PrivateInstanceReference ?Strategy}
UNION { ?Context BE:ProtectedInstanceReference ?Strategy}}

{{ ?Context BE:Delegate ?Strategy}
UNION { ?Context BE:Redirect ?Strategy}
UNION { ?Context BE:DelegateInLimitedFamily ?Strategy}
UNION { ?Context BE:RedirectInLimitedFamily ?Strategy}}

OPTIONAL {?x BE:CreateObject ?Strategy.} FILTER(!bound(?x))

NOT EXISTS { ?Context BE:ExtendedInheritance ?Strategy .}
```

```
NOT EXISTS { ?Strategy BE:Delegate ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:Redirect ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:Conglomeration ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:Recursion ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:RevertMethod ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:ExtendMethod ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:DelegatedConglomeration ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:RedirectRecursion ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:DelegateInFamily ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:RedirectInFamily ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:DelegateInLimitedFamily ?ConcreteStrategy}
NOT EXISTS { ?Strategy BE:RedirectInLimitedFamily ?ConcreteStrategy}
}
```

Merge rule The merge rule for the *Strategy* pattern is basically the same as the one for the *State* pattern. Each pattern instance is identified by a couple {Strategy, Context}; many Concrete Strategy instances are allowed for each pattern instance.

```
<pattern name="Strategy">
  <role name="Strategy" />
  <role name="Context" />
  <sublevel>
    <role name="ConcreteStrategy" />
  </sublevel>
</pattern>
```

Merge rule diagram

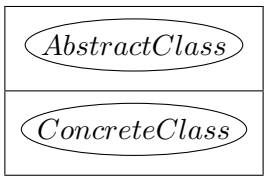


A.3.10 Template Method

Complete rule The *Template Method* pattern is detected using a rule that needs no more expressiveness than the XML syntax can provide. It is a very simple rule that specifies classes declaring an abstract method and calling it, and their subclasses. It is possible to have many Concrete Classes for each *AbstractClass*.

```
<pattern name="TemplateMethod">
  <role name="AbstractClass">
    <be name="AbstractMethodInvoked" to="AbstractClass" />
    <be name="AbstractInterface" to="AbstractClass" />
  </role>
  <sublevel>
    <role name="ConcreteClass">
      <be name="ExtendedInheritance" to="AbstractClass" />
    </role>
  </sublevel>
</pattern>
```

Merge rule diagram



A.3.11 Visitor

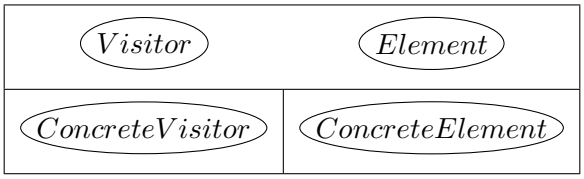
Match rule In the *Visitor* pattern the match rule defines the two inheritances and their relations. ConcreteElements pass theirselves to the Visitors and Visitors accept Elements as paramters. The abstract and concrete roles are allowed to collapse, but at least the two abstract roles must not be related one to the other. In addition, Concrete Visitors must have an implementation.

```
PREFIX BE:      <http://essere.disco.unimib.it/marple/BEs#>

SELECT  ?Visitor ?Element ?ConcreteVisitor ?ConcreteElement
WHERE {
?Visitor BE:InheritanceThisParameter ?ConcreteElement.
?Element BE:ReceivesParameter ?Visitor.

{{?ConcreteElement BE:ExtendedInheritance ?Element.}
UNION { ?ConcreteElement BE:SameClass ?Element.}}

{{?ConcreteVisitor BE:ExtendedInheritance ?Visitor.}
UNION { ?ConcreteVisitor BE:SameClass ?Visitor. }}

NOT EXISTS { ?Visitor BE:ExtendedInheritance  ?Element . }
NOT EXISTS { ?Element BE:ExtendedInheritance  ?Visitor . }
NOT EXISTS { ?Visitor BE:SameClass  ?Element . }
NOT EXISTS { ?ConcreteVisitor BE:Interface  ?ConcreteVisitor . }
NOT EXISTS { ?ConcreteVisitor BE:PseudoClass  ?ConcreteVisitor . }

OPTIONAL {
    ?ConcreteElement BE:SameClass ?Element; BE:ExtendedInheritance ?e.
    ?e BE:ReceivesParameter ?Visitor.
}
FILTER(!bound(?e) || ?e = ?Element)

OPTIONAL {
    ?ConcreteVisitor BE:SameClass ?Visitor; BE:ExtendedInheritance ?v.
    ?v BE:InheritanceThisParameter ?ConcreteElement.
}
FILTER(!bound(?v) || ?v = ?Visitor)

OPTIONAL {
    ?ConcreteVisitor BE:SameClass ?Visitor.
    ?cv BE:ExtendedInheritance ?Visitor.
}
FILTER(!bound(?cv) || ?cv = ?ConcreteVisitor)
}
```

Merge rule A *Visitor* pattern instance is defined by a Visitor and its related Element. Their concrete implementations are each one if a different sublevel.

```
<pattern name="Visitor">
  <role name="Visitor" />
  <role name="Element" />
  <sublevel>
    <role name="ConcreteVisitor" />
  </sublevel>
  <sublevel>
    <role name="ConcreteElement" />
  </sublevel>
</pattern>
```

Merge rule diagram



Appendix B

Setup parameters for the experimented algorithms

This appendix contains the setup of the parameters for each experiment reported in Chapter 8. The parameter names are taken from the parameters of the respective algorithm implementations found in Weka. When an algorithm has a major variant it was considered as a different algorithm. The name of the variant should be sufficient to setup the respective algorithm using the dedicated parameters, which are not reported here for brevity.

B.1 Setup of the genetic algorithm parameters

This section contains the list of the parameters exploited by the genetic algorithm for their optimization. For each parameter the range of tested values is reported. Boolean values have no range. When a boolean value has literals in the range cell, it means it was employed as a selector for a nominal parameter. The parameters are reported in Tables B.1, B.2, B.3, B.4, B.5, B.6 and B.7.

Table B.1: SimpleKMeans: experiment parameters

Parameter	Type	Min/False	Max/True
DISTANCE_FUNCTION	Boolean	ManhattanDistance	EuclideanDistance
MAX_ITERATIONS	Integer	10	1000
NUM_CLUSTERS	Integer	2	n. pattern instances
PRESERVE_INSTANCE_ORDER	Boolean		

Table B.2: CLOPE: experiment parameters

Parameter	Type	Min/False	Max/True
REPULSION	Double	0.0001	5

Table B.3: JRip: experiment parameters

Parameter	Type	Min/False	Max/True
FOLDS	Integer	2	20
MIN_NO	Double	1	10
OPTIMIZATIONS	Integer	1	10
USE_PRUNING	Boolean		

Table B.4: SMO: experiment parameters

Parameter	Type	Min/False	Max/True
COMPLEXITY	Double	0	1
RBF_GAMMA	Double	0	1
BUILD_LOGISTIC	Boolean		
FILTER	Integer	0	2
	0:SMO.FILTER_NONE, 2:SMO.FILTER_STANDARDIZE		1:SMO.FILTER_NORMALIZE,

Table B.5: RandomForest: experiment parameters

Parameter	Type	Min/False	Max/True
MAX_DEPTH	Integer	1	100
NUM_FEATURES	Integer	1	100
NUM_TREES	Integer	1	100

Table B.6: J48: experiment parameters

Parameter	Type	Min/False	Max/True
MIN_NUM_OBJ	Integer	2	10
USE_LAPLACE	Boolean		
NUM_FOLDS	Integer	2	10
SUBTREE_RAISING	Boolean		
CONFIDENCE_FACTOR	Double	0.000001	0.499999

Table B.7: LibSVM: experiment parameters

Parameter	Type	Min/False	Max/True
EPS	Double	0.000001	0.5
NORMALIZE	Boolean		
PROBABILTY_ESTIMATES	Boolean		
SHRINKING	Boolean		
NU	Double	0.001	1.9 * min(t, f) / (t + f)
	t,f: number of correct/incorrect instances		
GAMMA	Double	0.000001	1
COEF0	Double	0	1
DEGREE	Integer	1	10
COST	Double	1	1000

B.2 Parameter values for the best result setups

This section contains the values assigned to the parameters in the setups that gave the best performance results for each clusterer/classifier/indicator combination.

B.2.1 Singleton

Table B.8: Singleton: JRip parameter setup

fitness	Area under ROC	Accuracy	F-measure
fitnessValue	0.876667869	88.23529412	0.85
FOLDS	14	17	10
MIN_NO	2.491927135	2.194347838	3.382071059
OPTIMIZATIONS	6	6	2
USE_PRUNING	1	1	1

Table B.9: Singleton: SMO parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	89.54248366	0.936350523	0.86440678
COMPLEXITY	0.907272432	0.390406503	0.878478287
RBF_GAMMA	0.048917945	0.079104816	0.053803617
BUILD_LOGISTIC	1	1	1
FILTER	0	0	0

Table B.10: Singleton: RandomForest parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	92.81045752	0.97295348	0.902654867
MAX_DEPTH	75	6	8
NUM_FEATURES	23	21	32
NUM_TREES	27	29	21

Table B.11: Singleton: J48 Unpruned parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	86.92810458	0.906869816	0.824561404
MIN_NUM_OBJ	6	2	5
USE_LAPLACE	0	1	1

Table B.12: Singleton: J48 reduced error pruning parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	88.23529412	0.85	0.909574468
MIN_NUM_OBJ	2	2	2
NUM_FOLDS	9	9	2
SUBTREE_RAISING	1	1	1
USE_LAPLACE	1	0	1

Table B.13: Singleton: J48 pruned parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	88.23529412	0.906689506	0.836065574
CONFIDENCE_FACTOR	0.009631137	0.489802895	0.015933832
MIN_NUM_OBJ	2	6	4
SUBTREE_RAISING	1	1	1
USE_LAPLACE	1	0	1

Table B.14: Singleton: LibSVM ν-SVC Linear parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	90.8496732	0.928056257	0.877192982
EPS	0.344078746	0.495402263	0.194587296
NORMALIZE	1	0	1
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	0	0	0
NU	0.342955452	0.314646335	0.335400779

Table B.15: Singleton: LibSVM ν-SVC Sigmoid parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	90.19607843	0.918139199	0.788461538
EPS	0.113284358	0.222587555	0.147295419
NORMALIZE	1	1	1
PROBABILTY_ESTIMATES	0	1	0
SHRINKING	1	1	0
NU	0.424088139	0.598580091	0.711933062
GAMMA	0.008328026	0.010606261	0.035134906
COEF0	0.116485554	0.0101632	0.549330163

Table B.16: Singleton: LibSVM ν-SVC RBF parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	90.8496732	0.94536603	0.884955752
EPS	0.100644678	0.42076115	0.120477979
NORMALIZE	0	0	1
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	0	0	1
NU	0.104058888	0.466017838	0.242945741
GAMMA	0.213819335	0.137143097	0.157237791

Table B.17: Singleton: LibSVM ν-SVC Polynomial parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	90.19607843	0.942661378	0.869565217
EPS	0.012350442	0.002045813	0.384346545
NORMALIZE	1	0	0
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	1	1	1
NU	0.036450364	0.45456103	0.080471697
GAMMA	0.489635291	0.512654237	0.878147677
COEF0	0.555937306	0.623301385	0.928545162
DEGREE	7	8	8

Table B.18: Singleton: LibSVM C-SVC Polynomial parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	85.62091503	0.934998197	0.882882883
EPS	0.473158238	0.335832258	0.057769974
NORMALIZE	1	1	0
PROBABILTY_ESTIMATES	0	1	1
SHRINKING	0	0	0
COST	4.759183142	160.5730193	64.99786261
GAMMA	0.031108782	0.003717292	0.004188825
COEF0	0.184433818	0.90200825	0.929203933

Table B.19: Singleton: LibSVM C-SVC RBF parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	91.50326797	0.947169131	0.884955752
EPS	0.187421331	0.475207425	0.291966499
NORMALIZE	1	0	1
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	0	1	0
COST	182.3670168	652.3441583	352.3539085
GAMMA	0.152371477	0.199882514	0.115169846

Table B.20: Singleton: LibSVM C-SVC Polynomial parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	90.8496732	0.936891453	0.879310345
EPS	0.33341545	0.447516262	0.443612346
NORMALIZE	0	0	0
PROBABILTY_ESTIMATES	0	1	0
SHRINKING	0	0	0
COST	920.691888	878.7101755	316.7532615
GAMMA	0.077797416	0.503991969	0.27360741
COEF0	0.956183195	0.758533359	0.934314589
DEGREE	9	8	6

Table B.21: Singleton: LibSVM C-SVC Linear parameter setup

fitness	Accuracy	Area under ROC	F-measure
fitnessValue	83.00653595	0.869094843	0.767857143
EPS	0.021746244	0.452934558	0.286795006
NORMALIZE	0	0	0
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	0	1	1
COST	690.7413843	481.1982605	213.5932008

B.2.2 Adapter

Table B.22: Adapter: SMO parameter setup

fitness type	Area under ROC	F-measure	Accuracy
fitnessvalue	0.922075924	0.838137472	84.59893048
Complexity	0.865905488	0.865985504	0.983172214
RBF Gamma	0.147751954	0.121408508	0.077572363
Build Log. Mod.	1	0	1
filter type	None	Normalize	Normalize

Table B.23: Adapter: J48 pruned parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	79.57219251	0.859284297	0.785634119
unpruned	0	0	0
reduced error pruning	0	0	0
confidence factor	0.199527046	0.471610036	0.220464716
min num obj	3	3	2
subtree raising	0	0	1
use laplace	0	1	0

Table B.24: Adapter: J48 unpruned parameter setup

fitness type	Area under ROC	F-measure	Accuracy
fitnessvalue	0.857816978	0.786740331	79.67914439
unpruned	1	1	1
reduced error pruning	0	0	0
min num obj	2	4	2
subtree raising	1	1	1
use laplace	1	0	0

Table B.25: Adapter: J48 reduced error pruning parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	78.71657754	0.838337681	0.787553648
unpruned	0	0	0
reduced error pruning	1	1	1
min num obj	2	6	2
subtree raising	1	0	0
use laplace	1	1	1
num folds	4	6	4

Table B.26: Adapter: RandomForest parameter setup

fitness type	F-measure	Area under ROC	Accuracy
fitnessvalue	0.83699422	0.922000147	84.9197861
max depth	63	30	78
num features	30	22	41
num trees	86	69	73

Table B.27: Adapter: LibSVM C-SVC Linear parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	79.78609626	0.848618102	0.788636364
COST	20.40604817	3.090249402	10.04844513
EPS	0.361713603	0.262171802	0.354427706
GAMMA			
NORMALIZE	0	0	0
PROBABILTY_ESTIMATES	0	1	0
SHRINKING	0	0	0

Table B.28: Adapter: LibSVM C-SVC Polynomial parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	83.95721925	0.88955838	0.824742268
COST	230.8110157	753.4009324	139.9882963
EPS	0.046882418	0.314884205	0.023950759
GAMMA	0.021235224	0.228908609	0.037430045
NORMALIZE	1	0	0
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	0	1	0
COEF0	0.874965071	0.785273948	0.273116537
DEGREE	10	5	9

Table B.29: Adapter: LibSVM C-SVC RBF parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	85.88235294	0.92256503	0.84787472
COST	43.51200118	255.5979976	643.4954656
EPS	0.454684996	0.00219059	0.413506262
GAMMA	0.084317679	0.135001489	0.126313835
NORMALIZE	1	1	0
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	1	0	0

Table B.30: Adapter: LibSVM C-SVC Sigmoid parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	78.71657754	0.83744443	0.686849574
COST	888.7931318	136.49770200376494	999.0044255
EPS	0.450397687	0.070988244	0.17289205
GAMMA	0.000331892	0.001674441	0.023477247
NORMALIZE	0	0	0
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	0	0	0
COEF0	0.537172567	0.14348617	0.275626783

Table B.31: Adapter: LibSVM ν-SVC Linear parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	80.42780749	0.85102919	0.792710706
EPS	0.434901047	0.044215302	0.130299238
NORMALIZE	0	1	1
PROBABILTY_ESTIMATES	0	1	0
SHRINKING	0	0	0
NU	0.28004794775497777	0.450111654	0.272521773

Table B.32: Adapter: LibSVM ν-SVC Polynomial parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	84.17112299	0.909163972	0.823798627
EPS	0.119565788	0.047143104	0.269064174
GAMMA	0.073863324	0.021901786	0.90431511
NORMALIZE	0	1	0
PROBABILTY_ESTIMATES	1	1	0
SHRINKING	0	0	0
COEF0	0.146193618	0.302682177	0.997158129
DEGREE	9	10	6
NU	0.131077112	0.263294293	0.156632596

Table B.33: Adapter: LibSVM ν-SVC RBF parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	85.56149733	0.925995665	0.840354767
EPS	0.017374013	0.018321126	0.170673699
GAMMA	0.07774256	0.094701366	0.151420292
NORMALIZE	1	1	0
PROBABILTY_ESTIMATES	1	1	0
SHRINKING	1	1	0
NU	0.37252526586781476	0.117159219	0.404754854

Table B.34: Adapter: LibSVM ν-SVC Sigmoid parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	74.54545455	0.820603553	0.639010189
EPS	0.04753725	0.017132685	0.153713693
GAMMA	0.003498945	0.00296521	0.123433326
NORMALIZE	0	1	0
PROBABILTY_ESTIMATES	0	1	1
SHRINKING	0	0	1
COEF0	0.337645091	0.23301761	0.455315401
NU	0.48029852	0.487152425	0.050874644

Table B.35: Adapter: JRip parameter setup

fitness type	Accuracy	Area under ROC	F-measure
fitnessvalue	80.53475936	0.815611452	0.770780856
FOLDS	19	9	6
MIN_NO	4.734980748	4.930796817	4.447837397
OPTIMIZATIONS	1	5	5
USE_PRUNING	0	0	0

B.2.3 Composite

SimpleKMeans

Table B.36: Composite: SimpleKMeans - OneR parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.43859649	0.111111111	0.500830565
DISTANCE_FUNCTION	1	0	1
MAX_ITERATIONS	434	518	495
NUM_CLUSTERS	31	13	13
PRESERVE_INSTANCE_ORDER	0	0	1

Table B.37: Composite: SimpleKMeans - NaiveBayes parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	77.19298246	0.380952381	0.626245847
DISTANCE_FUNCTION	1	1	1
MAX_ITERATIONS	406	460	247
NUM_CLUSTERS	46	46	46
PRESERVE_INSTANCE_ORDER	1	1	0

Table B.38: Composite: SimpleKMeans - JRip parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.43859649	0.347826087	0.47923588
DISTANCE_FUNCTION	0	1	0
MAX_ITERATIONS	493	131	555
NUM_CLUSTERS	20	16	57
PRESERVE_INSTANCE_ORDER	0	1	0
FOLDS	20	2	4
MIN_NO	8.429803315	3.380404131	1.527368557
OPTIMIZATIONS	1	4	7
USE_PRUNING	0	0	0

Table B.39: Composite: SimpleKMeans - RandomForest parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.43859649	0.454545455	0.61461794
DISTANCE_FUNCTION	1	1	0
MAX_ITERATIONS	66	937	383
NUM_CLUSTERS	3	33	15
PRESERVE_INSTANCE_ORDER	1	1	1
MAX_DEPTH	89	9	7
NUM_FEATURES	94	3	39
NUM_TREES	9	1	6

Table B.40: Composite: SimpleKMeans - J48 unpruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.43859649	0.272727273	0.592192691
DISTANCE_FUNCTION	0	0	0
MAX_ITERATIONS	837	589	613
NUM_CLUSTERS	34	17	31
PRESERVE_INSTANCE_ORDER	1	0	0
MIN_NUM_OBJ	5	5	2
USE_LAPLACE	0	1	0

Table B.41: Composite: SimpleKMeans - J48 reduced error pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	77.19298246	0.25	0.512458472
DISTANCE_FUNCTION	1	1	0
MAX_ITERATIONS	311	569	380
NUM_CLUSTERS	16	35	24
PRESERVE_INSTANCE_ORDER	1	0	0
MIN_NUM_OBJ	4	2	2
USE_LAPLACE	1	1	1
NUM_FOLDS	8	10	7
SUBTREE_RAISING	0	1	1

Table B.42: Composite: SimpleKMeans - J48 pruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.43859649	0	0.509136213
DISTANCE_FUNCTION	1	1	0
MAX_ITERATIONS	387	73	488
NUM_CLUSTERS	43	22	57
PRESERVE_INSTANCE_ORDER	1	0	1
MIN_NUM_OBJ	3	3	2
USE_LAPLACE	0	0	1
SUBTREE_RAISING	0	1	1
CONFIDENCE_FACTOR	0.077144984	0.442125797	0.490801397

Table B.43: Composite: SimpleKMeans - LibSVM C-SVC RBF parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	78.94736842	0.363636364	0.882890365
DISTANCE_FUNCTION	1	1	1
MAX_ITERATIONS	671	739	206
NUM_CLUSTERS	5	23	20
PRESERVE_INSTANCE_ORDER	0	1	0
EPS	0.080501581	0.300619908	0.355347522
NORMALIZE	0	0	1
PROBABILTY_ESTIMATES	1	0	1
SHRINKING	1	1	1
GAMMA	0.172696899	0.731010389	0.047630468
COST	129.7843365	890.6592276	35.88081429

Table B.44: Composite: SimpleKMeans - LibSVM ν-SVC RBF parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.70175439	0.551724138	0.67192691
DISTANCE_FUNCTION	1	0	0
MAX_ITERATIONS	948	970	185
NUM_CLUSTERS	19	17	46
PRESERVE_INSTANCE_ORDER	0	1	1
EPS	0.227419701	0.319148425	0.156936556
NORMALIZE	1	0	1
PROBABILTY_ESTIMATES	1	0	1
SHRINKING	1	1	0
NU	0.271947753	0.334323643	0.087636061
GAMMA	0.409835808	0.203621544	0.790240729

CLOPE

Table B.45: Composite: CLOPE - OneR parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.43859649	0	0.5
REPULSION	0.183870953	1.874012942	2.024577041

Table B.46: Composite: CLOPE - NaiveBayes parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.43859649	0.117647059	0.522425249
REPULSION	2.418779301	4.460473835	4.993424821

Table B.47: Composite: CLOPE - JRip parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.70175439	0.421052632	0.59717608
REPULSION	4.009555786	3.987781208	3.641830125
FOLDS	11	18	15
MIN_NO	2.579433314	1.335208694	1.931853551
OPTIMIZATIONS	8	8	1
USE_PRUNING	0	0	1

Table B.48: Composite: CLOPE - RandomForest parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.70175439	0.380952381	0.647009967
REPULSION	3.709822053	3.856774369	0.824341586
MAX_DEPTH	2	99	96
NUM_FEATURES	76	8	42
NUM_TREES	99	16	1

Table B.49: Composite: CLOPE - J48 unpruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.70175439	0.421052632	0.533222591
REPULSION	3.867457495	3.847633716	3.77623696
MIN_NUM_OBJ	3	3	2
USE_LAPLACE	0	0	0

Table B.50: Composite: CLOPE - J48 reduced error pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.70175439	0.25	0.55730897
REPULSION	3.900886157	3.93265919	3.903386009
MIN_NUM_OBJ	2	3	3
NUM_FOLDS	8	8	3
SUBTREE_RAISING	0	0	0
USE_LAPLACE	0	0	0

Table B.51: Composite: CLOPE - J48 pruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.43859649	0.421052632	0.57641196
REPULSION	2.176694351	3.859931202	3.792497218
MIN_NUM_OBJ	9	2	4
SUBTREE_RAISING	0	1	1
USE_LAPLACE	1	1	0
CONFIDENCE_FACTOR	0.335749462	0.429897668	0.399593715

Table B.52: Composite: CLOPE - LibSVM C-SVC RBF parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.70175439	0.4	0.611295681
REPULSION	3.658938766	3.843321227	3.900085972
EPS	0.413262897	0.316562312	0.023123313
NORMALIZE	1	0	1
PROBABILTY_ESTIMATES	1	0	1
SHRINKING	0	0	1

Table B.53: Composite: CLOPE - LibSVM ν-SVC RBF parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	77.19298246	0.555555556	0.717607973
REPULSION	2.733112047	0.685348493	4.520115091
EPS	0.087404586	0.03277159	0.33927191
NORMALIZE	0	1	1
PROBABILTY_ESTIMATES	1	0	0
SHRINKING	0	1	1
NU	0.428307441	0.349824712	0.208937114
GAMMA	0.743461332	0.033228898	0.550830164

B.2.4 Decorator

SimpleKMeans

Table B.54: Decorator: SimpleKMeans - OneR parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	69.71830986	0.557692308	0.650609756
DISTANCE_FUNCTION	0	1	1
MAX_ITERATIONS	846	555	371
NUM_CLUSTERS	104	34	108
PRESERVE_INSTANCE_ORDER	0	0	1

Table B.55: Decorator: SimpleKMeans - NaiveBayes parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	77.46478873	0.737704918	0.761890244
DISTANCE_FUNCTION	0	1	1
MAX_ITERATIONS	855	48	390
NUM_CLUSTERS	136	136	123
PRESERVE_INSTANCE_ORDER	1	1	0

Table B.56: Decorator: SimpleKMeans - JRip parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	79.57746479	0.756302521	0.755081301
DISTANCE_FUNCTION	0	1	0
MAX_ITERATIONS	747	400	735
NUM_CLUSTERS	93	92	104
PRESERVE_INSTANCE_ORDER	0	1	0
FOLDS	3	19	10
MIN_NO	1.631138361	1.181172757	1.441909535
OPTIMIZATIONS	4	6	1
USE_PRUNING	0	0	1

Table B.57: Decorator: SimpleKMeans - RandomForest parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	81.69014085	0.774193548	0.819004065
DISTANCE_FUNCTION	0	1	1
MAX_ITERATIONS	953	504	994
NUM_CLUSTERS	129	123	123
PRESERVE_INSTANCE_ORDER	1	1	1
MAX_DEPTH	49	53	13
NUM_FEATURES	45	27	18
NUM_TREES	6	15	82

Table B.58: Decorator: SimpleKMeans - J48 unpruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	77.46478873	0.747826087	0.742886179
DISTANCE_FUNCTION	0	0	0
MAX_ITERATIONS	903	79	760
NUM_CLUSTERS	131	136	115
PRESERVE_INSTANCE_ORDER	0	0	1
MIN_NUM_OBJ	4	4	3
USE_LAPLACE	1	0	0

Table B.59: Decorator: SimpleKMeans - J48 reduced error pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	77.46478873	0.732142857	0.765752033
DISTANCE_FUNCTION	1	0	0
MAX_ITERATIONS	259	582	508
NUM_CLUSTERS	112	95	133
PRESERVE_INSTANCE_ORDER	0	1	1
MIN_NUM_OBJ	2	2	2
USE_LAPLACE	1	0	0
NUM_FOLDS	9	10	5
SUBTREE_RAISING	0	0	1

Table B.60: Decorator: SimpleKMeans - J48 pruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.28169014	0.75862069	0.753760163
DISTANCE_FUNCTION	1	0	1
MAX_ITERATIONS	551	720	670
NUM_CLUSTERS	136	136	141
PRESERVE_INSTANCE_ORDER	1	0	1
MIN_NUM_OBJ	2	2	4
USE_LAPLACE	1	0	0
SUBTREE_RAISING	1	1	0
CONFIDENCE_FACTOR	0.137163929	0.167650337	0.350560531

Table B.61: Decorator: SimpleKMeans - LibSVM C-SVC RBF parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	79.57746479	0.747826087	0.815752033
DISTANCE_FUNCTION	1	1	0
MAX_ITERATIONS	863	210	873
NUM_CLUSTERS	95	130	127
PRESERVE_INSTANCE_ORDER	1	1	0
EPS	0.081406929	0.056190332	0.377847611
NORMALIZE	1	0	1
PROBABILTY_ESTIMATES	1	0	1
SHRINKING	1	0	0
GAMMA	0.002206704	0.658422613	0.127348768
COST	860.2031727	210.9170396	296.6834907

Table B.62: Decorator: SimpleKMeans - LibSVM ν-SVC RBF parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.28169014	0.75862069	0.809349593
DISTANCE_FUNCTION	1	0	0
MAX_ITERATIONS	281	446	28
NUM_CLUSTERS	124	122	132
PRESERVE_INSTANCE_ORDER	1	1	1
EPS	0.132975006	0.475033107	0.076417029
NORMALIZE	0	0	0
PROBABILTY_ESTIMATES	0	0	1
SHRINKING	1	1	1
NU	0.514773445	0.509070688	0.741886437
GAMMA	0.406221173	0.550577284	0.570703549

CLOPE

Table B.63: Decorator: CLOPE - OneR parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	66.1971831	0.591549296	0.644715447
REPULSION	4.785043641	4.570730394	4.714897055

Table B.64: Decorator: CLOPE - NaiveBayes parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	69.71830986	0.666666667	0.727235772
REPULSION	4.747312231	4.809130751	4.741317658

Table B.65: Decorator: CLOPE - Jrip parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	71.12676056	0.648648649	0.681808943
REPULSION	4.709496849	4.684522026	4.789454749
FOLDS	6	3	5
MIN_NO	5.201339273	2.697944142	3.946578448
OPTIMIZATIONS	10	10	5
USE_PRUNING	1	1	1

Table B.66: Decorator: CLOPE - RandomForest parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	73.23943662	0.724832215	0.744308943
REPULSION	4.840795752	4.921676173	4.917802019
MAX_DEPTH	8	3	51
NUM_FEATURES	3	25	19
NUM_TREES	29	99	25

Table B.67: Decorator: CLOPE - J48 unpruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	72.53521127	0.661538462	0.73546748
REPULSION	4.899147674	4.805396176	4.896935622
MIN_NUM_OBJ	4	10	3
USE_LAPLACE	1	1	1

Table B.68: Decorator: CLOPE - J48 reduced error pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	71.83098592	0.67768595	0.74054878
REPULSION	4.900482596	4.895846622	4.632937405
MIN_NUM_OBJ	2	2	9
NUM_FOLDS	4	4	3
SUBTREE_RAISING	0	0	0
USE_LAPLACE	1	1	0

Table B.69: Decorator: CLOPE - J48 pruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	72.53521127	0.649122807	0.725914634
REPULSION	4.898652991	4.462867544	4.93429999
MIN_NUM_OBJ	3	10	2
SUBTREE_RAISING	0	0	0
USE_LAPLACE	1	0	0
CONFIDENCE_FACTOR	0.3083529	0.493976558	0.477118428

Table B.70: Decorator: CLOPE - LibSVM C-SVC RBF parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	71.83098592	0.661290323	0.723373984
REPULSION	4.900345284	4.838658879	4.917109203
EPS	0.032546474	0.474632229	0.448192638
NORMALIZE	0	1	1
PROBABILTY_ESTIMATES	0	0	1
SHRINKING	1	0	0
GAMMA	0.22872371	0.197780493	0.100424045
COST	411.3189911	727.1400025	8.456614514

Table B.71: Decorator: CLOPE - LibSVM ν-SVC RBF parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	73.94366197	0.704	0.764329268
REPULSION	4.86697326	4.855369018	4.859859604
EPS	0.491801915	0.22313508	0.106376777
NORMALIZE	0	0	0
PROBABILTY_ESTIMATES	0	0	1
SHRINKING	0	1	0
NU	0.613261861	0.386209277	0.614024318
GAMMA	0.444388438	0.51232777	0.668408264

B.2.5 Factory Method

SimpleKMeans

Table B.72: Factory Method: SimpleKMeans - OneR parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	70.03121748	0.734920635	0.697658169
DISTANCE_FUNCTION	0	0	0
MAX_ITERATIONS	264	909	855
NUM_CLUSTERS	3	4	3
PRESERVE_INSTANCE_ORDER	1	1	0

Table B.73: Factory Method: SimpleKMeans - NaiveBayes parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.54110302	0.82146161	0.870658976
DISTANCE_FUNCTION	1	1	1
MAX_ITERATIONS	559	632	683
NUM_CLUSTERS	122	163	336
PRESERVE_INSTANCE_ORDER	1	0	0

Table B.74: Factory Method: SimpleKMeans - JRip parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	75.6503642	0.793879388	0.774114869
DISTANCE_FUNCTION	1	1	0
MAX_ITERATIONS	172	309	43
NUM_CLUSTERS	61	52	24
PRESERVE_INSTANCE_ORDER	0	0	1
FOLDS	16	4	12
MIN_NO	4.696965029	3.052700485	2.998763359
OPTIMIZATIONS	4	10	8
USE_PRUNING	0	1	0

Table B.75: Factory Method: SimpleKMeans - RandomForest parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	81.89386056	0.833955224	0.873158781
DISTANCE_FUNCTION	1	0	1
MAX_ITERATIONS	677	960	807
NUM_CLUSTERS	123	162	143
PRESERVE_INSTANCE_ORDER	1	0	1
MAX_DEPTH	75	67	100
NUM_FEATURES	77	4	2
NUM_TREES	27	51	19

Table B.76: Factory Method: SimpleKMeans - J48 unpruned parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.12486993	0.808300395	0.864496263
DISTANCE_FUNCTION	1	0	1
MAX_ITERATIONS	531	72	164
NUM_CLUSTERS	123	120	144
PRESERVE_INSTANCE_ORDER	0	0	0
MIN_NUM_OBJ	2	2	2
USE_LAPLACE	0	1	0

Table B.77: Factory Method: SimpleKMeans - J48 reduced error pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	77.73152966	0.805217391	0.853709346
DISTANCE_FUNCTION	0	0	1
MAX_ITERATIONS	228	268	163
NUM_CLUSTERS	111	115	163
PRESERVE_INSTANCE_ORDER	1	1	1
MIN_NUM_OBJ	2	2	2
USE_LAPLACE	0	1	0
NUM_FOLDS	9	8	7
SUBTREE_RAISING	1	0	1

Table B.78: Factory Method: SimpleKMeans - J48 pruned pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	79.18834547	0.814814815	0.849929259
DISTANCE_FUNCTION	0	1	1
MAX_ITERATIONS	920	108	822
NUM_CLUSTERS	120	165	141
PRESERVE_INSTANCE_ORDER	0	1	0
MIN_NUM_OBJ	2		2
USE_LAPLACE	0		0
SUBTREE_RAISING	1		0
CONFIDENCE_FACTOR	0.431117159	0.021105213	0.473508094

Table B.79: Factory Method: SimpleKMeans - LibSVM C-SVC RBF pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	81.78980229	0.826904055	0.839502556
DISTANCE_FUNCTION	0	1	0
MAX_ITERATIONS	162	120	28
NUM_CLUSTERS	123	123	500
PRESERVE_INSTANCE_ORDER	0	1	1
EPS	0.17957423	0.257847013	0.285146007
NORMALIZE	1	1	0
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	1	0	0
GAMMA	0.433900861	0.995915645	0.154952215
COST	68.85739085	158.413506	175.758742

Table B.80: Factory Method: SimpleKMeans - LibSVM ν-SVC RBF pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	80.2289282	0.823747681	0.865162444
DISTANCE_FUNCTION	1	0	0
MAX_ITERATIONS	571	925	155
NUM_CLUSTERS	334	164	336
PRESERVE_INSTANCE_ORDER	1	1	0
EPS	0.254983279	0.264379569	0.156750503
NORMALIZE	1	0	0
PROBABILTY_ESTIMATES	0	1	1
SHRINKING	1	0	0
NU	0.562181189	0.67549282	0.699786976
GAMMA	0.377257812	0.27181635	0.132671811

CLOPE

Table B.81: Factory Method: CLOPE - OneR pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	54.11030177	0.689607708	0.525766216
REPULSION	4.975233877	4.494942346	4.996605249

Table B.82: Factory Method: CLOPE - NaiveBayes pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	55.1508845	0.686261107	0.550141482
REPULSION	4.974190008	0.180940135	4.947164564

Table B.83: Factory Method: CLOPE - JRip pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	54.31841831	0.689226519	0.518206043
REPULSION	4.967227909	4.419208949	4.966745238
FOLDS	9	9	5
MIN_NO	3.351089895	1.232149997	5.917575367
OPTIMIZATIONS	7	5	10
USE_PRUNING	0	1	0

Table B.84: Factory Method: CLOPE - RandomForest pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	55.46305931	0.692200557	0.566832886
REPULSION	4.956635607	4.646575307	4.896194005
MAX_DEPTH	8	36	10
NUM_FEATURES	5	72	18
NUM_TREES	11	58	3

Table B.85: Factory Method: CLOPE - J48 unpruned pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	55.04682622	0.69123783	0.541012421
REPULSION	4.95711195	4.655361008	4.782930144
MIN_NUM_OBJ	4	3	4
USE_LAPLACE	0	0	0

Table B.86: Factory Method: CLOPE - J48 reduced error pruning pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	54.52653486	0.692520776	0.548474946
REPULSION	4.971951846	4.492369234	4.989492797
MIN_NUM_OBJ	2	6	3
NUM_FOLDS	7	6	3
SUBTREE_RAISING	1	1	0
USE_LAPLACE	1	0	0

Table B.87: Factory Method: CLOPE - J48 pruned pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	55.25494277	0.69123783	0.542451111
REPULSION	4.948825773	4.648482649	4.780161516
MIN_NUM_OBJ	3	3	4
SUBTREE_RAISING	1	1	0
USE_LAPLACE	0	1	0
CONFIDENCE_FACTOR	0.481916334	0.45165662	0.181158498

Table B.88: Factory Method: CLOPE - LibSVM C-SVC RBF pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	55.67117586	0.694019471	0.536939388
REPULSION	4.958204031	4.654364389	4.976853476
EPS	0.195764181	0.493704237	0.046550532
NORMALIZE	0	0	1
PROBABILTY_ESTIMATES	1	0	0
SHRINKING	0	1	1
GAMMA	0.709541592	0.814339376	0.760618206
COST	968.5425882	949.7913411	539.5664491

Table B.89: Factory Method: CLOPE - LibSVM ν-SVC RBF pruning parameter setup

fitnessType	Accuracy	F-measure	Area under ROC
fitnessValue	56.39958377	0.692041522	0.579789773
REPULSION	4.89562368	4.445868389	4.993482293
EPS	0.37228905	0.439423838	0.492907525
NORMALIZE	1	0	0
PROBABILTY_ESTIMATES	1	1	1
SHRINKING	1	1	1
NU	0.594281712	0.901827114	0.552202024
GAMMA	0.369782724	0.981972706	0.358000756

Appendix C

Comparison with other tools

The Classifier module was compared to other available results produced by different design pattern detection tools. The comparison is made against a single project, namely "6 - JHotDraw v5.1", which is contained in PMARt. The compared results come from different tools or sources:

- MARPLE Joiner module;
- MARPLE Classifier module;
- PMARt [83];
- DPD-Tool [195] (using the results loaded into DPB [65]);
- Web of Patterns [59] (using the results loaded into DPB [65]);
- DPJF [37] (results exported by the authors of the tool).

Every reported pattern instance was also manually evaluated.

The set of analyzed instances is the union of the instances reported as *correct* by each tool, minus the Joiner module, because its target is not to be a full detector, but only a step in a computation chain. The name of the instances in the tables are symbolic, and they represent an index in the dataset (in the case of PMARt) or a sequence number paired with the name of a representative class in the pattern instance. The results for every tool are reported as boolean values, where true (T) means the tool considers the instance as correct; cells containing "-" values represent cases where the value is not available, i.e. when the tool does not support the pattern, or it was not applied on that instance (in the case of the Classifier module). The "Manual" tool represents the manual evaluation of the instances. The only different column is the one for the Classifier module, which reports the confidence value for each instance; when the value is positive the instance was considered as correct, otherwise it is negative. No value (represented with "-") is specified for the Classifier if the Joiner column contains false (F), because the instance was not evaluated. Only the patterns tested by the Classifier module were included in the comparison.

The set of compared instances is composed of 85 *Adapter* instances, 2 *Singleton* instances, 5 *Composite* instances, 5 *Decorator* instances, and 10 *Factory Method* instances.

For each pattern the comparison reports the analytic data and a comparison matrix. The comparison matrix reports the tools (considering also the manual evaluation) on rows and columns, and reports in each cell the number of instances considered correct by the two respective tools. Only the upper half matrix is filled, and the values in the lower half are represented as "-". Each cell gives the idea of the agreement of correct instances. The row corresponding to the manual evaluation tells how many instances were really correct for each tool; the diagonal, instead tell

how many instances were reported as correct by each tool. The fraction of the values contained in the manual row and the respective values on the diagonal (for each column) gives the idea of the precision of each tool.

C.1 Adapter

Table C.1 reports the list of all the instances considered for the comparison, and their evaluation by each of the considered tools. Table C.2, instead, shows the comparison matrix.

Table C.1: Adapter comparison

Instance	Manual	PMARt:1	Joiner:135	Classifier:75	DPJF:-	WoP:0	DPD-Tool:22
pmart 73	T	T	T	0.70	-	F	F
dpdtool 1 - DrawApplication	T	F	F	-	-	F	T
dpdtool 2 - AlignCommand	T	F	T	0.64	-	F	T
dpdtool 3 - AbstractTool	T	F	T	0.71	-	F	T
dpdtool 4 - BringToFrontCommand	T	F	T	0.85	-	F	T
dpdtool 5 - StandardDrawingView	T	F	F	-	-	F	T
dpdtool 6 - StandardDrawingView	T	F	F	-	-	F	T
dpdtool 7 - GroupCommand	T	F	T	0.87	-	F	T
dpdtool 8 - TextFigure	T	F	F	-	-	F	T
dpdtool 9 - ChangeAttributeCommand	T	F	T	0.68	-	F	T
dpdtool 10 - ToggleGridCommand	T	F	T	0.74	-	F	T
dpdtool 11 - LocatorHandle	T	F	T	0.81	-	F	T
dpdtool 12 - StandardDrawingView	T	F	F	-	-	F	T
dpdtool 13 - AbstractFigure	T	F	F	-	-	F	T
dpdtool 14 - ReverseFigureEnumerator	F	F	F	-	-	F	T
dpdtool 15 - InsertImageCommand	T	F	T	0.64	-	F	T
dpdtool 16 - UngroupCommand	T	F	T	0.81	-	F	T
dpdtool 17 - AbstractConnector	T	F	F	-	-	F	T
dpdtool 18 - LineConnection	T	F	F	-	-	F	T
dpdtool 19 - PolygonHandle	T	F	T	0.71	-	F	T
dpdtool 20 - SentToBackCommand	T	F	T	0.85	-	F	T
dpdtool 21 - Toolbutton	T	F	T	0.63	-	F	T
dpdtool 22 - DrawApplet	T	F	F	-	-	F	T
marple 1 - Animator	T	F	T	0.54	-	F	F
marple 2 - JavaDrawApplet	F	F	T	0.64	-	F	F
marple 3 - JavaDrawApp	T	F	T	0.56	-	F	F
marple 4 - CommandButton	T	F	T	0.61	-	F	F
marple 5 - CommandButton	F	F	T	0.57	-	F	F
marple 6 - ChangeConnectionHandle	T	F	T	0.58	-	F	F
marple 7 - ConnectionTool	T	F	T	0.83	-	F	F
marple 8 - ConnectionHandle	T	F	T	0.53	-	F	F
marple 9 - ChangeConnectionHandle	F	F	T	0.63	-	F	F
marple 10 - ElbowHandle	T	F	T	0.75	-	F	F

Table C.1: Adapter comparison

Instance	Manual	PMARt:1	Joiner:135	Classifier:75	DPJF:-	WoP:0	DPD-Tool:22
marple 11 - ConnectionTool	T	F	T	0.85	-	F	F
marple 12 - ConnectionHandle	F	F	T	0.70	-	F	F
marple 13 - ActionTool	T	F	T	0.56	-	F	F
marple 14 - ConnectionTool	T	F	T	0.81	-	F	F
marple 15 - CreationTool	T	F	T	0.65	-	F	F
marple 16 - PolygonTool	T	F	T	0.78	-	F	F
marple 17 - ScribbleTool	T	F	T	0.70	-	F	F
marple 18 - SelectAreaTracker	T	F	T	0.68	-	F	F
marple 19 - SelectionTool	T	F	T	0.55	-	F	F
marple 20 - BringToFrontCommand	T	F	T	0.75	-	F	F
marple 21 - FigureTransferCommand	F	F	T	0.58	-	F	F
marple 22 - GRoupCommand	T	F	T	0.74	-	F	F
marple 23 - SendToBackCommand	T	F	T	0.76	-	F	F
marple 24 - UngroupCommand	T	F	T	0.70	-	F	F
marple 25 - ActionTool	T	F	T	0.57	-	F	F
marple 26 - ConnectionTool	T	F	T	0.84	-	F	F
marple 27 - CreationTool	T	F	T	0.55	-	F	F
marple 28 - PolygonTool	T	F	T	0.76	-	F	F
marple 29 - ScribbleTool	T	F	T	0.69	-	F	F
marple 30 - FigureTransferCommand	F	F	T	0.68	-	F	F
marple 31 - Animator	T	F	T	0.56	-	F	F
marple 32 - ConnectionTool	T	F	T	0.77	-	F	F
marple 33 - CreationTool	T	F	T	0.71	-	F	F
marple 34 - DragTracker	F	F	T	0.68	-	F	F
marple 35 - FollowURLTool	T	F	T	0.69	-	F	F
marple 36 - SelectAreaTracker	T	F	T	0.66	-	F	F
marple 37 - URLTool	T	F	T	0.68	-	F	F
marple 38 - AlignCommand	T	F	T	0.60	-	F	F
marple 39 - ChangeAttributeCommand	T	F	T	0.62	-	F	F
marple 40 - FigureTransferCommand	F	F	T	0.57	-	F	F
marple 41 - UngroupCommand	T	F	T	0.68	-	F	F
marple 42 - ChangeConnectionHAndle	F	F	T	0.55	-	F	F
marple 43 - ConnectionTool	T	F	T	0.74	-	F	F
marple 44 - DragTRacker	T	F	T	0.68	-	F	F
marple 45 - SelectAreaTracker	T	F	T	0.58	-	F	F
marple 46 - AlignCommand	T	F	T	0.68	-	F	F
marple 47 - BringToFrontCommand	T	F	T	0.78	-	F	F
marple 48 - ChangeAttributeCommand	T	F	T	0.75	-	F	F
marple 49 - FigureTransferCommand	F	F	T	0.63	-	F	F
marple 50 - SendToBackCommand	T	F	T	0.78	-	F	F
marple 51 - UngroupCommand	T	F	T	0.68	-	F	F
marple 52 - ConnectionHandle	F	F	T	0.60	-	F	F

Table C.1: Adapter comparison

Instance	Manual	PMARt:1	Joiner:135	Classifier:75	DPJF:-	WoP:0	DPD-Tool:22
marple 53 - DuplicateCommand	T	F	T	0.69	-	F	F
marple 54 - PasteCommand	T	F	T	0.58	-	F	F
marple 55 - TextTool	T	F	T	0.55	-	F	F
marple 56 - HandleTracker	T	F	T	0.68	-	F	F
marple 57 - InsertImageCommand	T	F	T	0.73	-	F	F
marple 58 - ToolButton	F	F	T	0.68	-	F	F
marple 59 - PaletteButton	F	F	T	0.52	-	F	F
marple 60 - ScribbleTool	T	F	T	0.65	-	F	F
marple 61 - PolygonTool	T	F	T	0.58	-	F	F
marple 62 - TextHolder	T	F	T	0.66	-	F	F

No results are available from Web of Patterns and DPJF for the *Adapter* design pattern. DPJF does not support the *Adapter* design pattern.

The other tools are very different in the number of reported instances: PMARt reports only one instance, DPD-Tool 22 instances and the Classifier module 75 instances. The precision of each tool is good, but the results are very different: the only instance reported by PMARt contains a lot of classes; it was considered correct because it contains adapters, even if it represents an entire family of *Adapter*s. The wrong evaluations of the Joiner, instead, are due to some restrictions in the rule that forbid the Adaptee and the Target to belong to the same type hierarchy; most of the mistakes are due to the implementation of Serializable , Cloneable, or other utility interfaces or classes. The one-to-one tool comparison (on positive instances), available in Table C.2, shows that the instance reported in PMARt was correctly identified by MARPLE and DPD-tool, while about half of the instances reported by DPD-tool were identified by MARPLE. The reason of this result is the already described restriction in the Joiner. Another fact is that a lot of the instances reported by MARPLE are correct (62/75) and only 12 are found by the other tools, leading to new 50 discovered pattern instances. MARPLE discovered a relevant number of unknown pattern instances, but did not report a half (ten) of the ones discovered by DPD-tool, because of the setup of the Joiner rule (which can be redesigned to enhance the detection). DPD-tool, instead, missed fifty correct instances, which is five times more than the instances missed by MARPLE and more than two times more than all the instances reported by DPD-tool. Finally, PMARt reports a single instance, which can be considered as an aggregation of instances, but is far from giving an idea of the design pattern instances contained in the project.

Table C.2: Adapter comparison matrix

	Manual	PMARt	Joiner	Classifier	DPJF	WoP	DPD-tool
Manual	71	1	62	62	0	0	21
PMARt	-	1	1	1	0	0	0
Joiner	-	-	75	75	0	0	12
Classifier	-	-	-	75	0	0	12
DPJF	-	-	-	-	0	0	0
Web of Patterns	-	-	-	-	-	0	0
DPD-tool	-	-	-	-	-	-	22

C.2 Singleton

Only two *Singleton* instances were found by the tools, one correct and the other no. MARPLE and Web of Patterns reported only the correct one, while PMARt and DPD-tool reported as correct also the wrong one. DPJF reported no results, because it does not support the pattern. Given the small number of instances, it is not possible to formulate more complex evaluations. Table C.3 and Table C.4 give the overview of the available results.

Table C.3: Singleton comparison

Instance	Manual	PMARt:2	Joiner:1	Classifier:1	DPJF:-	WoP:1	DPD-tool:2
pmart 85	T	T	T	0.99	-	T	T
pmart 86	F	T	F	-	-	F	T

Table C.4: Singleton comparison matrix

	Manual	PMARt	Joiner	Classifier	DPJF	WoP	DPD-tool
Manual	1	1	1	1	0	1	1
PMARt	-	2	1	1	0	1	2
Joiner	-	-	1	1	0	1	1
Classifier	-	-	-	1	0	1	1
DPJF	-	-	-	-	0	0	0
Web of Patterns	-	-	-	-	-	1	1
DPD-tool	-	-	-	-	-	-	2

C.3 Composite

The only correct *Composite* instance is reported by all the tools but not from the Classifier module. Other four instances were reported by Web Of Patterns, and they are all incorrect ones. Table C.5 and Table C.6 give the overview of the available data. The results available for this pattern are aligned with the ones reported in Subsection 8.2.3, which highlight *Composite* is the pattern with more detection issues. The motivations of the lower performance for the *Composite*

design pattern are mainly related to the size of the training set, as explained in Subsection 8.2.3, and to the micro-structures used to catch the peculiarities of the pattern, e.g. the aggregation of children. Some of these considerations are available in Section 6.3.2. In addition, some of the micro-structures which are useful for the detection of the *Composite* suffer from the granularity of the reported information, which is not able, e.g., to characterize the exact method cycling on the children and forwarding the invocation. Future work will be directed to the correction of these class of problems to enhance the detection of the *Composite* design pattern.

Table C.5: Composite comparison

Instance	Manual	PMARt:1	Joiner:5	Classifier:0	DPJF:1	WoP:5	DPD-tool:1
pmart 75	T	T	T	-0.74	T	T	T
wop 1 - FigureChangeListener	F	F	F	-	F	T	F
wop 2 - FigureChangeListener	F	F	F	-	F	T	F
wop 3 - CompositeFigure	F	F	F	-	F	T	F
wop 5 - FigureChangeListener	F	F	F	-	F	T	F

Table C.6: Composite comparison matrix

	Manual	PMARt	Joiner	Classifier	DPJF	WoP	DPD-tool
Manual	1	1	1	0	1	1	1
PMARt	-	1	1	0	1	1	1
Joiner	-	-	1	0	1	1	1
Classifier	-	-	-	0	0	0	0
DPJF	-	-	-	-	1	1	1
Web of Patterns	-	-	-	-	-	5	1
DPD-tool	-	-	-	-	-	-	1

C.4 Decorator

The situation for the comparison of *Decorator* instances is the opposite one: five instances were found, all correct ones, and all correctly identified by the Classifier module. PMARt contains only one correct instance, DPD-tool three correct instances and DPJF three correct instances. The instances reported by DPD-tool and DPJF are not completely overlapping, while every tool reported the only instance found in PMARt. Web of Patterns reported no results because it does not support the detection of this pattern. The overview of the results is reported in Table C.7 and Table C.8.

C.5 Factory Method

The results for the *Factory Method* design pattern (shown in Table C.9) contain 10 instances, found by PMARt, DPD-Tool and MARPLE, with little overlap. No single instance is agreed

Table C.7: Decorator comparison

Instance	Manual	PMARt:1	Joiner:14	Classifier:5	DPJF:3	WoP:-	DPD-tool:3
pmart 76	T	T	T	0.68	T	-	T
dpjf 1 - FigureEnumerator	T	F	T	0.66	T	-	F
dpjf 2 - OffsetLocator	T	F	T	0.66	T	-	T
dpdtool 1 - SelectionTool	T	F	T	0.71	F	-	T
marple 1 - FigureChangeListener	T	F	T	0.68	F	-	F

Table C.8: Decorator comparison matrix

	Manual	PMARt	Joiner	Classifier	DPJF	WoP	DPD-tool
Manual	5	1	5	5	3	0	3
PMARt	-	1	1	1	1	0	1
Joiner	-	-	5	5	3	0	3
Classifier	-	-	-	5	3	0	3
DPJF	-	-	-	-	3	0	2
Web of Patterns	-	-	-	-	-	0	0
DPD-tool	-	-	-	-	-	-	3

by the three tools. Web of Patterns and DPJF reported no instances because the pattern is not supported. PMARt contains three instances; another instance is added by DPD-tool and other 6 by MARPLE. Table C.10 shows that DPD-tool returns 2/2 correct instances; PMARt is the second one, having 2/3 correct instances and MARPLE returned 4 correct instance out of 6 found instances. MARPLE is the tool that is able to find more correct instances for this pattern.

C.6 Conclusion

The comparison experiment reported in this appendix served as an example, demonstrating that MARPLE has performances which are not worse than the ones of the other tools in most cases, and that in many cases it produces a larger number of correct pattern instances. From the analysis of the results it appears that the approach based on machine learning is able to find many correct (and less trivial) instances that are discarded by exact matching approaches, which are forced to use strong constraints to keep an acceptable level of precision. Despite some localized issue the approach demonstrated its validity and that it is worth working in this direction.

The reported data are not intended to be a full comparison of the different tools, primarily because they refer to the pattern detection applied to only one system. A full comparison would need to be done on more pattern instances, to reach a good significance, and the manual evaluation should be agreed by many people. The precision values reported in the section are therefore not to be intended as real estimations of the precision of the tools, but as clues helping to reason on the available data. In this direction, it will be interesting to exploit the DPB [65] platform to perform a wider and open comparison among the results of different tools, including

Table C.9: Factory Methods comparison

Instance	Manual	PMARt:3	Joiner:62	Classifier:7	DPJF:0	WoP:-	DPD-tool:2
pmart 77	T	T	T	-0.86	-	-	T
pmart 78	F	T	F	-	-	-	F
pmart 79	T	T	T	0.96	-	-	F
dpdtool 1 - Drawingview	T	F	T	-0.86	-	-	T
marple 1 - NetApp	F	F	T	0.67	-	-	F
marple 2 - NothingApp	F	F	T	0.67	-	-	F
marple 2 - PertApplication	F	F	T	0.67	-	-	F
marple 3 - PertFigureCreationTool	T	F	T	0.87	-	-	F
marple 4 - BoxHandleKit	T	F	T	0.82	-	-	F
marple 5 - RelativeLocator	T	F	T	0.82	-	-	F

Table C.10: Factory Method comparison matrix

	Manual	PMARt	Joiner	Classifier	DPJF	WoP	DPD-tool
Manual	6	2	6	4	0	0	2
PMARt	-	3	2	1	0	0	1
Joiner	-	-	9	7	0	0	2
Classifier	-	-	-	7	0	0	0
DPJF	-	-	-	-	0	0	0
Web of Patterns	-	-	-	-	-	0	0
DPD-tool	-	-	-	-	-	-	2

MARPLE-DPD.

Another aspect to consider in the instance comparison is the different grouping of pattern instances performed by different tools, which brings to situations where an instance for one tool can be matched to more than one instance for another tool.

Publications

[1] Francesca Arcelli Fontana, Andrea Caracciolo, and Marco Zanoni. DPB: A benchmark for design pattern detection tools. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, CSMR 2012, pages 235–244, Szeged, Hungary, March 2012. IEEE Computer Society. doi:10.1109/CSMR.2012.32.

[2] Francesca Arcelli Fontana, Marco Zanoni, Bartosz Walter, and Pawel Martenka. Code smells, micro patterns and their relations. *ERCIM News*, 88:33, January 2012. Special theme: Evolving Software. URL: http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf.

[3] Francesca Arcelli, Andrea Caracciolo, and Marco Zanoni. A benchmark for design pattern detection tools: a community driven approach. *ERCIM News*, 88:32, January 2012. Special theme: Evolving Software. URL: http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf.

[4] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, April 2011. doi:10.1016/j.ins.2010.12.002.

[5] Francesca Arcelli Fontana and Marco Zanoni. On investigating code smells correlations. In *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), RefTest Workshop*, pages 474–475, Berlin, Germany, March 2011. IEE. doi:10.1109/ICSTW.2011.14.

[6] Francesca Arcelli Fontana, Marco Zanoni, and Stefano Maggioni. Using design pattern clues to improve the precision of design pattern detection tools. *Journal of Object Technology*, 10:4:1–31, 2011. URL: http://www.jot.fm/contents/issue_2011_01/article4.html, doi:10.5381/jot.2011.10.1.a4.

[7] Francesca Arcelli Fontana, Marco Zanoni, and Andrea Caracciolo. A benchmark platform for design pattern detection. In *Proceedings of The Second International Conferences on Pervasive Patterns and Applications*, PATTERNS 2010, pages 42–47, Lisbon, Portugal, November 2010. IARIA, Think Mind. URL: http://www.thinkmind.org/index.php?view=article&articleid=patterns_2010_2_30_70046.

[8] Francesca Arcelli, Gianluigi Viscusi, and Marco Zanoni. Unifying software and data reverse engineering: a pattern based approach. In *Proc. of the 5th International Conference on Software and Data Technologies, ICSOFT2010*, pages 208–213, Athens, Greece, July 2010. SciTePress. Short paper. doi:10.5220/0003010202080213.

[9] Francesca Arcelli, Marco Zanoni, Riccardo Porrini, and Mattia Vivanti. A model proposal for program comprehension. In *Proceedings of the 16th International Conference on Distributed*

Multimedia Systems, DMS 2010: Globalization and Personalization, pages 23–28, Oak Brook, Illinois, USA, October 2010. Knowledge Systems Institute. URL: `http://www.ksi.edu/seke/Proceedings/dms/DMS2010_Proceedings.pdf`.

[10] Christian Tosi, Marco Zanoni, and Stefano Maggioni. A design pattern detection plugin for eclipse. In Angelo Gargantini, editor, *Proceedings of the 4th Italian Workshop on Eclipse Technologies (Eclipse-IT 2009)*, Bergamo, Italy, September 2009. Eclipse Italian Community. URL: `http://eit09.unibg.it/pdfs/99990089.pdf`.

[11] Francesca Arcelli Fontana, Christian Tosi, and Marco Zanoni. Can design pattern detection be useful for legacy system migration towards soa? In *SDSOA '08: IEEE Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 63–68, New York, NY, USA, 2008. ACM. `doi:10.1145/1370916.1370932`.

[12] Francesca Arcelli, Christian Tosi, and Marco Zanoni. A benchmark proposal for design pattern detection. In *Proceedings of 2nd Workshop on FAMIX and Moose in Reengineering*, FAMOOSr 2008, pages 24–27, Antwerp, Belgium, October 2008. MOOSE Technology. Co-located with WCRE 2008. URL: `http://www.moosetechnology.org/?_s=JdhwOEnqbH-6feuC`.

[13] Francesca Arcelli, Christian Tosi, Marco Zanoni, and Stefano Maggioni. The marple project — a tool for design pattern detection and software architecture reconstruction. In *Proceedings of the 1st International Workshop on Academic Software Development Tools and Techniques*, WASDeTT 2008, Paphos, Cyprus, July 2008. Software Composition Group. co-located with ECOOP 2008. URL: `http://www.iam.unibe.ch/~scg/download/wasdett/wasdett2008-paper02.pdf`.

[14] Christian Tosi, Marco Zanoni, Francesca Arcelli, and Claudia Raibulet. Joiner: from subcomponents to design patterns. In *Proceedings of the DPD4RE Workshop, co-located event with IEEE WCRE 2006 Conference*, Benevento, Italy, 2006. IEEE Computer Society.

Bibliography

[1] SQL — Part 1: Framework (SQL/Framework), 2011. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681.

[2] Md. Abul Khaer, M.M.A. Hashem, and Md. Raihan Masud. On use of design patterns in empirical assessment of software design quality. In *International Conference on Computer and Communication Engineering*, ICCCE 2008, pages 133–137, Kuala Lumpur, Malesia, May 2008. IEEE Computer Society. doi:10.1109/ICCCE.2008.4580582.

[3] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast partial evaluation of pattern matching in strings. *ACM Trans. Program. Lang. Syst.*, 28(4):696–714, July 2006. doi:10.1145/1146809.1146812.

[4] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: putting bits and pieces together. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering*, ASE 2001, pages 166–173, San Diego, CA, USA, November 2001. IEEE Computer Society. doi:10.1109/ASE.2001.989802.

[5] Giulio Antoniol, Roberto Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 153–160, Ischia, Italy, June 1998. IEEE Computer Society. doi:10.1109/WPC.1998.693342.

[6] Francesca Arcelli, Andrea Caracciolo, and Marco Zanoni. A benchmark for design pattern detection tools: a community driven approach. *ERCIM News*, 88:32, January 2012. Special theme: Evolving Software. URL: http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf.

[7] Francesca Arcelli, Luca Cristina, and Daniele Franzosi. nMARPLE: .NET reverse engineering with marple. In *Proceedings of the ECOOP 2007 Workshop on Object-Oriented Re-engineering*, WOOR 2007, Berlin, Germany, July 2007. Software Composition Group. URL: http://scg.unibe.ch/wiki/events/woor2007/nmarplenetreverseengineeringwithmarple.

[8] Francesca Arcelli and Claudia Raibulet. The role of design pattern decomposition in reverse engineering tools. In *Pre-Proceedings of the IEEE International Workshop on Software Technology and Engineering Practice (STEP 2005)*, pages 230–233, Budapest, Hungary, 2005.

[9] Francesca Arcelli, Christian Tosi, and Marco Zanoni. A benchmark proposal for design pattern detection. In *Proceedings of 2nd Workshop on FAMIX and Moose in Reengineering*, FAMOOSr 2008, pages 24–27, Antwerp, Belgium, October 2008. MOOSE Tech-

nology. Co-located with WCRE 2008. URL: http://www.moosetechnology.org/?_s=JdhwOEnqbH-6feuC.

[10] Francesca Arcelli, Christian Tosi, Marco Zanoni, and Stefano Maggioni. The marple project — a tool for design pattern detection and software architecture reconstruction. In *Proceedings of the 1st International Workshop on Academic Software Development Tools and Techniques*, WASDeTT 2008, Paphos, Cyprus, July 2008. Software Composition Group. co-located with ECOOP 2008. URL: http://www.iam.unibe.ch/~scg/download/wasdett/wasdett2008-paper02.pdf.

[11] Francesca Arcelli, Christian Tosi, Marco Zanoni, and Stefano Maggioni. Marple. Web site, 2009. URL: http://essere.disco.unimib.it/reverse/Marple.html.

[12] Francesca Arcelli, Gianluigi Viscusi, and Marco Zanoni. Unifying software and data reverse engineering: a pattern based approach. In *Proc. of the 5th International Conference on Software and Data Technologies, ICSOFT2010*, pages 208–213, Athens, Greece, July 2010. SciTePress. Short paper. doi:10.5220/0003010202080213.

[13] Francesca Arcelli, Marco Zanoni, Riccardo Porrini, and Mattia Vivanti. A model proposal for program comprehension. In *Proceedings of the 16th International Conference on Distributed Multimedia Systems*, DMS 2010: Globalization and Personalization, pages 23–28, Oak Brook, Illinois, USA, October 2010. Knowledge Systems Institute. URL: http://www.ksi.edu/seke/Proceedings/dms/DMS2010_Proceedings.pdf.

[14] Francesca Arcelli Fontana. Software evolution and reverse engineering lab at university of milano bicocca. Web site, 2012. http://essere.disco.unimib.it.

[15] Francesca Arcelli Fontana, Andrea Caracciolo, and Marco Zanoni. DPB: A benchmark for design pattern detection tools. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, CSMR 2012, pages 235–244, Szeged, Hungary, March 2012. IEEE Computer Society. doi:10.1109/CSMR.2012.32.

[16] Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Design patterns: a survey on their micro-structures. *Journal of Software Maintenance and Evolution: Research and Practice*, June 2011. doi:10.1002/smr.547.

[17] Francesca Arcelli Fontana, Christian Tosi, and Marco Zanoni. Can design pattern detection be useful for legacy system migration towards soa? In *SDSOA '08: IEEE Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 63–68, New York, NY, USA, 2008. ACM. doi:10.1145/1370916.1370932.

[18] Francesca Arcelli Fontana and Marco Zanoni. On investigating code smells correlations. In *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), RefTest Workshop*, pages 474–475, Berlin, Germany, March 2011. IEE. doi:10.1109/ICSTW.2011.14.

[19] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, April 2011. doi:10.1016/j.ins.2010.12.002.

[20] Francesca Arcelli Fontana, Marco Zanoni, and Andrea Caracciolo. A benchmark platform for design pattern detection. In *Proceedings of The Second International Conferences on Pervasive Patterns and Applications*, PATTERNS 2010, pages 42–47, Lisbon, Portugal,

November 2010. IARIA, Think Mind. URL: http://www.thinkmind.org/index.php?view=article&articleid=patterns_2010_2_30_70046.

[21] Francesca Arcelli Fontana, Marco Zanoni, and Stefano Maggioni. Using design pattern clues to improve the precision of design pattern detection tools. *Journal of Object Technology*, 10:4:1–31, 2011. URL: http://www.jot.fm/contents/issue_2011_01/article4.html, doi:10.5381/jot.2011.10.1.a4.

[22] Francesca Arcelli Fontana, Marco Zanoni, Bartosz Walter, and Pawel Martenka. Code smells, micro patterns and their relations. *ERCIM News*, 88:33, January 2012. Special theme: Evolving Software. URL: http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf.

[23] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 1027–1035, New Orleans, Louisiana, USA, January 2007. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1283383.1283494.

[24] 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, Richmond, Virginia, USA, October 2002. IEEE Computer Society. doi:10.1109/WCRE.2002.1173067.

[25] J. C. M. Baeten, F. Corradini, and C. A. Grabmayer. A characterization of regular expressions under bisimulation. *Journal of the ACM*, 54(2), April 2007. doi:10.1145/1219092.1219094.

[26] Pierre Baldi, Søren Brunak, Yves Chauvin, Claus A. F. Andersen, and Henrik Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16(5):412–424, February 2000. doi:10.1093/bioinformatics/16.5.412.

[27] Thomas Ball. The concept of dynamic analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 216–234. Springer Berlin / Heidelberg, 1999. URL: http://dx.doi.org/10.1007/3-540-48166-4_14, doi:10.1007/3-540-48166-4_14.

[28] Jagdish Bansiya. Automating design-pattern identification. *Dr Dobbs Journal*, June 1998. URL: http://drdobbs.com/architecture-and-design/184410578.

[29] Victor R. Basili and Barry Boehm. COTS-based systems top 10 list. *Computer*, 34(5):91–95, May 2001. doi:10.1109/2.920618.

[30] Ian Bayley and Hong Zhu. Formalising design patterns in predicate logic. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM 2007, pages 25–36, London, UK, September 2007. IEEE Computer Society. doi:10.1109/SEFM.2007.22.

[31] Ian Bayley and Hong Zhu. Formal specification of the variants and behavioural features of design patterns. *Journal of Systems and Software*, 83(2):209–221, 2010. Computer Software and Applications. URL: http://www.sciencedirect.com/science/article/pii/S0164121209002489, doi:10.1016/j.jss.2009.09.039.

[32] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 354–364, Szeged, Hungary, September 2011. ACM. doi:10.1145/2025113.2025162.

[33] Yoshua Bengio and Yves Grandvalet. No unbiased estimator of the variance of k-fold cross-validation. *Journal of Machine Learning Research*, 5:1089–1105, September 2004. URL: http://www.jmlr.org/papers/volume5/grandvalet04a/grandvalet04a.pdf.

[34] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 216–225, Victoria, B.C., Canada, November 2003. IEEE Computer Society. doi:10.1109/WCRE.2003.1287252.

[35] James C. Bezdek, Robert Ehrlich, and William Full. FCM: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, 10(2-3):191–203, 1984. doi:10.1016/0098-3004(84)90020-7.

[36] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, Kiarash Mahdavi, and Joachim Wegener. Dependence anti patterns. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, 2008*, pages 25–34, L'Aquila, Italy, September 2008. IEEE Computer Society. doi:10.1109/ASEW.2008.4686318.

[37] Alexander Binun and Günter Kniesel. DPJF — design pattern detection with high accuracy. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, CSMR 2012, Szeged, Hungary, March 2012. IEEE Computer Society.

[38] Marcel Birkner. Objected-oriented design pattern detection using static and dynamic analysis in java software. Master's thesis, Bonn-Rhine-Sieg University of Applied Sciences, Sankt Augustin, Germany, 2007. URL: http://mb-pde.googlecode.com/files/MasterThesis.pdf.

[39] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of java design patterns. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE'01, pages 324–327, San Diego, CA, USA, November 2001. IEEE Computer Society. doi:10.1109/ASE.2001.989821.

[40] Andrew P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997. doi:10.1016/S0031-3203(96)00142-2.

[41] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. doi:10.1023/A:1010933404324.

[42] William Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley, April 1998.

[43] Hugo Bruneliere, Jordi Cabot, and Grégoire Dupé. How to deal with your it legacy: What is coming up in modisco? *ERCIM News*, 88:43–44, January 2012. Special theme: Evolving Software. URL: http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf.

[44] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. URL: http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[45] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, January 1990. doi:10.1109/52.43044.

[46] Bertrand Clarke, Ernest Fokoue, Hao Helen Zhang, Bertrand Clarke, Ernest Fokoué, and Hao Helen Zhang. Learning in high dimensions. In *Principles and Theory for Data Mining and Machine Learning*, Springer Series in Statistics, pages 493–568. Springer New York, 2009. doi:10.1007/978-0-387-98135-2_9.

[47] Bertrand Clarke, Ernest Fokoue, Hao Helen Zhang, Bertrand Clarke, Ernest Fokoué, and Hao Helen Zhang. Unsupervised learning: Clustering. In *Principles and Theory for Data Mining and Machine Learning*, Springer Series in Statistics, pages 405–491. Springer New York, 2009. doi:10.1007/978-0-387-98135-2_8.

[48] Bertrand Clarke, Ernest Fokoue, Hao Helen Zhang, Bertrand Clarke, Ernest Fokoué, and Hao Helen Zhang. Variable selection. In *Principles and Theory for Data Mining and Machine Learning*, Springer Series in Statistics, pages 569–678. Springer New York, 2009. doi:10.1007/978-0-387-98135-2_10.

[49] William W. Cohen. Fast effective rule induction. In Armand Prieditis and Stuart J. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, ICML 1995, pages 115–123, Tahoe City, California, USA, July 1995. Morgan Kaufmann. URL: http://sci2s.ugr.es/keel/pdf/algorithm/congreso/ml-95-ripper.pdf.

[50] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. doi:10.1007/BF00994018.

[51] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery by visual language parsing. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, CSMR '05, pages 102–111, Washington, DC, USA, Manchester, UK 2005. IEEE Computer Society. doi:10.1109/CSMR.2005.23.

[52] Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, Digital Media Systems Laboratory — HP Laboratories Bristol, Bristol, September 2005. URL: http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html.

[53] Rajesh N. Davé and Raghu Krishnapuram. Robust clustering methods: a unified view. *IEEE Transactions on Fuzzy Systems*, 5(2):270–293, May 1997. doi:10.1109/91.580801.

[54] Lawrence Davis. *Handbook of genetic algorithms*. Van Nostrand Reinhold, 1991.

[55] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. An eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–6, Timisoara, Romania, September 2010. IEEE Computer Society. doi:10.1109/ICSM.2010.5609707.

[56] Simon Denier and Mircea Lungu, editors. *FAMOOSr 2010 4th Workshop on FAMIX and MOOSE in Software Reengineering*, Timisoara, Romania, September 2010. Self Published. Co-located with ICSM 2010. URL: http://www.moosetechnology.org/events/famoosr2010/FAMOOSr2010Proceedings.

[57] Jens Dietrich and Chris Elgar. A formal description of design patterns using owl. In *Proceedings of the 2005 Australian Software Engineering Conference*, ASWEC 2005, pages 243–250, Brisbane, Australia, March-1 April 2005. IEEE Computer Society. doi:10.1109/ASWEC.2005.6.

[58] Jens Dietrich and Chris Elgar. Owl ontology used by wop. Web Page, 2005. URL: http://www-ist.massey.ac.nz/wop/20050204/owldoc/index.html.

[59] Jens Dietrich and Chris Elgar. Towards a web of patterns. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):108–116, 2007. Software Engineering and the Semantic Web. doi:10.1016/j.websem.2006.11.007.

[60] Jing Dong, Dushyant S. Lad, and Yajing Zhao. Dp-miner: Design pattern discovery using matrix. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 371–380, Tucson, Arizona, USA, March 2007. IEEE Computer Society. doi:10.1109/ECBS.2007.33.

[61] Jing Dong, Yajing Zhao, and Tu Peng. Architecture and design pattern discovery techniques – a review. In *Proceedings of the 6th International Workshop on System/Software Architectures*, IWSSA'07, Las Vegas, Nevada, USA, June 2007. URL: http://www.utdallas.edu/~jdong/papers/iwssa07.pdf.

[62] Gregoire Dupe and Hugo Bruneliere. MoDisco, 2012. URL: http://www.eclipse.org/MoDisco/.

[63] Yasser EL-Manzalawy. WLSVM. Web site, 2005. URL: http://www.cs.iastate.edu/~yasser/wlsvm/.

[64] Félix Agustín Castro Espinoza, Gustavo Núñez Esquer, and Joel Suárez Cansino. Automatic design patterns identification of C++ programs. In Hassan Shafazand and A. Tjoa, editors, *EurAsia-ICT 2002: Information and Communication Technology*, volume 2510 of *Lecture Notes in Computer Science*, pages 816–823. Springer Berlin / Heidelberg, 2002. doi:10.1007/3-540-36087-5_94.

[65] ESSeRE lab. Design pattern benchmark platform. Web Site, 2010. http://essere.disco.unimib.it/DPB/.

[66] Jean-Marie Favre. CaCOphoNy: metamodel-driven software architecture reconstruction. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE 2004, pages 204–213, Delft, The Netherlands, November 2004. IEEE Computer Society. doi:10.1109/WCRE.2004.15.

[67] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006. doi:10.1016/j.patrec.2005.10.010.

[68] Rudolf Ferenc and Árpád Beszédes. Data exchange with the Columbus schema for C++. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, CSMR 2002, pages 59–66, Budapest, Hungary, March 2002. IEEE Computer Society. doi:10.1109/CSMR.2002.995790.

[69] Rudolf Ferenc, Árpád Beszédes, Lajos Fülöp, and Janos Lele. Design pattern mining enhanced by machine learning. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 295–304, Budapest, Hungary, September 2005. IEEE Computer Society. doi:10.1109/ICSM.2005.40.

[70] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus — reverse engineering tool and schema for C++. In *Proceedings of the International Conference on Software Maintenance*, ICSM'02, pages 172–181, Montréal, Canada, October 2002. IEEE Computer Society. doi:10.1109/ICSM.2002.1167764.

[71] Douglas H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2):139–172, 1987. doi:10.1007/BF00114265.

[72] Lajos Jeno Fülöp. *Evaluating and Improving Reverse Engineering Tools*. PhD thesis, Department of Software Engineering, University of Szeged, Szeged, Hungary, June 2011. URL: http://www.inf.u-szeged.hu/~flajos/LajosJenoFulop_thesis.pdf.

[73] Lajos Jeno Fülöp, Péter Hegedus, Rudolf Ferenc, and Tibor Gyimóthy. Towards a benchmark for evaluating reverse engineering tools. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 335–336, October 2008. doi:10.1109/WCRE.2008.18.

[74] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[75] John H. Gennari, Pat Langley, and Doug Fisher. Models of incremental concept formation. *Artificial Intelligence*, 40(1–3):11–61, 1989. doi:10.1016/0004-3702(89)90046-5.

[76] Joseph (Yossi) Gil and Itay Maman. Micro patterns in java code. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 97–116, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094819.

[77] Tudor Gîrba. *The Moose Book*. Self Published, 2010. URL: http://www.themoosebook.org/book.

[78] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[79] Orla Greevy. Dynamix — a meta-model to support feature-centric analysis. In *Proceedings of the 1st Workshop on FAMIX and Moose in Reengineering*, FAMOOSr 2007, Zurich, Switzerland, June 2007. Software Composition Group. co-located with TOOLS Europe 2007. URL: http://scg.unibe.ch/archive/papers/Gree07cDynamixFAMOOSr2007.pdf.

[80] Orla Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Bern, May 2007. URL: http://scg.unibe.ch/archive/phd/greevy-phd.pdf.

[81] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM. URL: http://doi.acm.org/10.1145/263698.264352, doi:10.1145/263698.264352.

[82] Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*. Springer Verlag, 2005.

[83] Yann-Gaël Guéhéneuc. PMARt: pattern-like micro architecture repository. In Michael Weiss, Aliaksandr Birukou, and Paolo Giorgini, editors, *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, July 2007.

[84] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 172–181, Victoria, BC, Canada, November 2004. IEEE Computer Society. doi:10.1109/WCRE.2004.21.

[85] Yann-Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, September 2008. doi:10.1109/TSE.2008.48.

[86] Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, and Houari Sahraoui. Improving design-pattern identification: a new approach and an exploratory study. *Software Quality Journal*, 18(1):145–174, 2010. doi:10.1007/s11219-009-9082-y.

[87] Manjari Gupta, Akshara Pande, Rajwant Singh Rao, and A.K. Tripathi. Design pattern detection by normalized cross correlation. In *International Conference on Methods and Models in Computer Science*, ICM2CS 2010, pages 81–84, New Delhi, India, December 2010. IEEE Computer Society. doi:10.1109/ICM2CS.2010.5706723.

[88] Manjari Gupta, Akshara Pande, and A. K. Tripathi. Design patterns detection using sop expressions for graphs. *ACM SIGSOFT Software Engineering Notes*, 36:1–5, January 2011. doi:10.1145/1921532.1921541.

[89] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1-2):83–114, 2005. Special Issue on the Static Analysis Symposium 2003 - SAS'03. doi:10.1016/j.scico.2005.02.005.

[90] Isabelle Guyon and Andre Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003. URL: http://www.jmlr.org/papers/volume3/guyon03a/guyon03a.pdf.

[91] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009. doi:10.1145/1656274.1656278.

[92] David Hand. Measuring classifier performance: a coherent alternative to the area under the ROC curve. *Machine Learning*, 77(1):103–123, 2009. doi:10.1007/s10994-009-5119-5.

[93] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java(tm) message service specification final release 1.1, April 2002. URL: http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec/.

[94] Trevor Hastie and Robert Tibshirani. Classification by pairwise coupling. *Annals of Statistics*, 26(2):451–471, 1998. doi:10.1214/aos/1028144844.

[95] Shinpei Hayashi, Junya Katada, Ryota Sakamoto, Takashi Kobayashi, and Motoshi Saeki. Design pattern detection by using meta patterns. *IEICE Transactions on Information and Systems*, E91-D(4):933–944, April 2008. doi:10.1093/ietisy/e91-d.4.933.

[96] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 94–103, Portland, Oregon, USA, May 2003. IEEE Computer Society. doi:10.1109/WPC.2003.1199193.

[97] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Gxl. Web site, 2002. URL: http://www.gupro.de/GXL/.

[98] Richard C. Holt, Ahmed E. Hasan, Bruno Laguë, Sébastien Lapierre, and Charles Leduc. E/R schema for the Datrix C/C++/Java exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, WCRE'00, pages 284 –286, Brisbane, QLD, Australia, November 2000. IEEE Computer Society. doi:10.1109/WCRE.2000.891481.

[99] Richard C. Holt, Andreas Winter, and Andy Schurr. GXL: toward a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, WCRE 2000, pages 162–171, Brisbane, QLD, Australia, November 2000. IEEE Computer Society. doi:10.1109/WCRE.2000.891463.

[100] Robert C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–90, 1993. doi:10.1023/A:1022631118932.

[101] Mei Hong, Tao Xie, and Fuqing Yang. Jbooret: an automated tool to recover oo design and source models. In *Proceedings of the 25th Annual International Computer Software and Applications Conference*, COMPSAC 2001, pages 71–76, Chicago, IL, USA, October 2001. IEEE Computer Society. doi:10.1109/CMPSAC.2001.960600.

[102] Richard C. Hull and Andreas Winter. A short introduction to the gxl software exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, WCRE 2000, pages 299–301, Brisbane, QLD, Australia, November 2000. IEEE Computer Society. doi:10.1109/WCRE.2000.891486.

[103] IBM. Jikes. Web site, 2009. http://jikes.sourceforge.net.

[104] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999. doi:10.1145/331499.331504.

[105] Java.net. Java compiler compiler [tm] (javacc [tm]) - the java parser generator. Web site, 2012. URL: http://javacc.java.net/.

[106] JBoss. Hibernate. Web Site, 2012. http://hibernate.org. URL: http://hibernate.org.

[107] Jena. ARQ — a SPARQL processor for Jena. Web site, 2011. URL: http://jena.sourceforge.net/ARQ/.

[108] Jena. Jena — a semantic web framework for Java. Web site, 2011. URL: http://openjena.org/.

[109] George John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In Philippe Besnard and Steve Hanks, editors, *Proceedings of the Eleventh Conference Annual Conference on Uncertainty in Artificial Intelligence*, UAI-95, pages 338–345, Montreal, Quebec, Canada, August 1995. Morgan Kaufmann. URL: http://uai.sis.pitt.edu/papers/95/p338-john.pdf.

[110] Narendra Jussien. e-constraints: Explanation-based constraint programming. In *Proceedings of the First International Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, December 2001. Self Published. Held in conjunction with Seventh International Conference on Principles and Practice of Constraint Programming. URL: http://www.cs.ucc.ie/~osullb/cp01/papers/jussien.ps.

[111] Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Efficient identification of design patterns with bit-vector algorithm. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, CSMR 2006, pages 175–184, Bari, Italy, March 2006. IEEE Computer Society. doi:10.1109/CSMR.2006.25.

[112] KDM Analytics. KDM 1.0 annotated reference, 2010. URL: http://kdmanalytics.com/kdmspec/index.php.

[113] S.S. Keerthi, S.K. Shevade, C. Bhattacharyya, and K.R.K. Murthy. Improvements to platt's smo algorithm for svm classifier design. *Neural Computation*, 13(3):637–649, March 2001. doi:10.1162/089976601300014493.

[114] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 226–235, Los Angeles, California, USA, May 1999. ACM. doi:10.1145/302405.302622.

[115] Duck Hoon Kim, Il Dong Yun, and Sang Uk Lee. A new attributed relational graph matching algorithm using the nested structure of earth mover's distance. In *Proceedings of the 17th International Conference on Pattern Recognition*, volume 1 of *ICPR 2004*, pages 48–51, Cambridge, England, UK, August 2004. IEEE Computer Society. doi:10.1109/ICPR.2004.1334002.

[116] Günter Kniesel. Type-safe delegation for run-time component adaptation. In Rachid Guerraoui, editor, *ECOOP' 99 — Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 668–668. Springer Berlin / Heidelberg, 1999. doi:10.1007/3-540-48743-3_16.

[117] Günter Kniesel, Alexander Binun, Péter Hegedüs, Lajos Jeno Fülöp, Alexander Chatzigeorgiouz, Yann-Gaël Guéhéneuc, and Nikolaos Tsantalis. A common exchange format for design pattern detection tools. Technical report, University of Bonn, 2010. URL: http://java.uom.gr/~nikos/publications/dpdx-techreport.pdf.

[118] Günter Kniesel and Alexander Binun. Standing on the shoulders of giants — a data fusion approach to design pattern detection. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 208–217, May 2009. doi:10.1109/ICPC.2009.5090044.

[119] Günter Kniesel, Alexander Binun, Péter Hegedüs, Lajos Jeno Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhenéuc, and Nikolaos Tsantalis. DPDX — towards a common result exchange format for design pattern detection tools. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, CSMR 2010, pages 232–235, Madrid, Spain, March 2010. IEEE Computer Society. doi:10.1109/CSMR.2010.40.

[120] Teuvo Kohonen. *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences*. Springer, Berlin, Heidelberg, New York, 3rd edition, 2001.

[121] Teuvo Kohonen. *The Handbook of Brain Theory and Neural Networks*, chapter Learning vector quantization, pages 537–540. MIT Press, Cambridge, MA, USA, 2 edition, November 2002.

[122] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the Third Working Conference*

on Reverse Engineering (WCRE'06), pages 208–215, Monterey, CA, USA, November 1996. IEEE Computer Society. doi:10.1109/WCRE.1996.558905.

[123] Adrian Kuhn and Toon Verwaest. Fame, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, MRT 2008, pages 57–66, Toulouse, France, September 2008. Self Published. In conjunction with MODELS 2008. URL: http://www.comp.lancs.ac.uk/~bencomo/MRT08/MRT2008Proceedings.pdf.

[124] Timothy C. Lethbridge. Report from the dagstuhl seminar on interoperability of reengineering tools. In *Proceedings of the 9th International Workshop on Program Comprehension*, IWPC 2001, page 119, Toronto, Ontario, Canada, May 2001. IEEE Computer Society. doi:10.1109/WPC.2001.921722.

[125] Timothy C. Lethbridge, Erhard Plödereder, Sander Tichelaar, Claudio Riva, Panos Linos, and Sergei Marchenko. The dagstuhl middle model (dmm), 2002. URL: http://www.site.uottawa.ca/~tcl/dmm/DMMDescriptionV0006.pdf.

[126] Timothy C. Lethbridge, Sander Tichelaar, and Erhard Ploedereder. The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, May 2004. Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003). doi:10.1016/j.entcs.2004.01.008.

[127] Ting Lim. Structured population genetic algorithms: a literature survey. *Artificial Intelligence Review*, pages 1–15, 2012. doi:10.1007/s10462-012-9314-6.

[128] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982. doi:10.1016/0022-0000(82)90009-5.

[129] Stefano Maggioni. Design patterns clues for creational design patterns. In *Proceedings of the First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE 2006), co-located event with WCRE 2006*, October 2006.

[130] Stefano Maggioni. *Design Pattern Detection and Software Architecture Reconstruction: an Integrated Approach based on Software Micro-structures*. PhD thesis, Università degli Studi di Milano Bicocca — Dipartimento di Informatica, Sistemistica e Comunicazione, Viale Sarca, 336 20126 Milano, Italy, October 2009.

[131] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990. doi:10.1007/BF01237234.

[132] Jacqueline A. McQuillan and James F. Power. Experiences of using the dagstuhl middle metamodel for defining software metrics. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, PPPJ '06, pages 194–198, Mannheim, Germany, 2006. ACM. doi:10.1145/1168054.1168082.

[133] Naouel Moha and Yann-Gael Guéhéneuc. Decor: a tool for the detection of design defects. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 527–528, Atlanta, Georgia, USA, November 2007. ACM. doi:10.1145/1321631.1321727.

[134] Naouel Moha and Yann-Gaël Guéhéneuc. PTIDEJ and DECOR: identification of design patterns and design defects. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 868–869, Montréal, Québec, Canada, 2007. ACM. doi:10.1145/1297846.1297930.

[135] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. The compiler generator Coco/R. Web Site, July 2011. URL: http://ssw.jku.at/Coco/.

[136] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA, 2000. ACM. doi:10.1145/336512.336526.

[137] Philip Newcomb. Architecture-driven modernization (adm). In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, page 237, Pittsburgh, Pennsylvania, USA, November 2005. IEEE Computer Society. doi:10.1109/WCRE.2005.7.

[138] Jörg Niere. Fuzzy logic based interactive recovery of software design. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 727–728, Orlando, Florida, USA, May 2002. ACM. doi:10.1145/581339.581473.

[139] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 338–348, Orlando, FL, USA, May 2002. ACM. doi:10.1145/581339.581382.

[140] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: an agile reengineering environment. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 1–10, Lisbon, Portugal, September 2005. ACM. doi:10.1145/1095430.1081707.

[141] Object Management Group, Inc. Meta Object Facility (MOF) core specification, January 2006. URL: http://www.omg.org/spec/MOF/2.0/PDF/.

[142] Object Management Group, Inc. XML Metadata Interchange (XMI®), 2011. URL: http://www.omg.org/spec/XMI/Current/.

[143] Object Management Group, Inc. Architecture-driven modernization task force. Web site, 2012. URL: http://adm.omg.org/.

[144] Object Management Group, Inc. Catalog of OMG modernization specifications, 2012. URL: http://www.omg.org/technology/documents/modernization_spec_catalog.htm.

[145] Object Management Group, Inc. Knowledge discovery metamodel (kdm). Web page, 2012. URL: http://www.omg.org/technology/kdm/index.htm.

[146] Object Management Group, Inc. Object management group, 2012. URL: http://www.omg.org.

[147] Oracle. GlassFish. Web site, 2012. http://glassfish.java.net/. URL: http://glassfish.java.net/.

[148] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, LLC, May 2007.

[149] Witold Pedrycz. *Clustering and Fuzzy Clustering*, chapter 1, pages 1–27. John Wiley & Sons, Inc., 2005. doi:10.1002/0471708607.ch1.

[150] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):16:1–16:45, September 2009. doi:10.1145/1567274.1567278.

[151] Claudia Perlich, Foster Provost, and Jeffrey S. Simonoff. Tree induction vs. logistic regression: a learning-curve analysis. *The Journal of Machine Learning Research*, 4:211–255, December 2003. doi:10.1162/153244304322972694.

[152] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*. MIT Press, 1998. URL: http://research.microsoft.com/~jplatt/smo.html.

[153] Wolfgang Pree. *Design patterns for object-oriented software development*. Addison-Wesley, 1995.

[154] Corina Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009. Special Section on HVC 07. doi:10.1007/s10009-009-0118-1.

[155] Ptidej Team. PADL. Web site, September 2011. URL: http://wiki.ptidej.net/doku.php?id=padl.

[156] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[157] Ghulam Rasool and Patrick Mäder. Flexible design pattern detection based on feature types. In *Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2011, pages 243–252, Lawrence, KS, USA, November 2011. IEEE Computer Society. doi:10.1109/ASE.2011.6100060.

[158] Ghulam Rasool, Ilka Philippow, and Patrick Mäder. Design pattern recovery based on annotations. *Advances in Engineering Software*, 41(4):519–526, April 2010. doi:10.1016/j.advengsoft.2009.10.014.

[159] RDF Working Group. Resource Description Framework (RDF). Web site, 2004. URL: http://www.w3.org/RDF/.

[160] Charles Rich and Linda M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 07(1):82–89, January 1990. doi:10.1109/52.43053.

[161] Lorenza Saitta and Filippo Neri. Learning in the "real world". *Machine Learning*, 30(2-3):133–163, 1998. doi:10.1023/A:1007448122119.

[162] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988. doi:10.1016/0306-4573(88)90021-0.

[163] F. Samadzadegan, A. Soleymani, and R. Ali Abbaspour. Evaluation of genetic algorithms for tuning svm parameters in multi-class problems. In *Proceedings of the 11th International Symposium on Computational Intelligence and Informatics*, CINTI 2010, pages 323–328, Budapest, Hungary, November 2010. IEEE Computer Society. doi:10.1109/CINTI.2010.5672224.

[164] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '98/FSE-6, pages 10–16, Orlando, FL, USA, November 1998. ACM. doi:10.1145/288195.288207.

[165] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 571–572, Atlanta, Georgia, USA, November 2007. ACM. doi:10.1145/1321631.1321746.

[166] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge, UK, 2004.

[167] 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, Tokyo, Japan, September 2006. IEEE Computer Society. doi:10.1109/ASE.2006.57.

[168] Jason McC Smith. An elemental design pattern catalog. Technical Report 02-040, Dept. of Computer Science, Univ. of North Carolina - Chapel Hill, December 2002. URL: ftp://ftp.cs.unc.edu/pub/publications/techreports/02-040.pdf.

[169] Jason McC. Smith and David Stotts. Elemental Design Patterns: A formal semantics for composition of OO software architecture. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*, SEW-27'02, pages 183–190, Greenbelt, Maryland, USA, December 2002. IEEE Computer Society. doi:10.1109/SEW.2002.1199472.

[170] Jason McC. Smith and David Stotts. SPQR: Flexible automated design pattern extraction from source code. In *Proceedings of the 18st IEEE/ACM International Conference on Automated Software Engineering*, ASE '03, pages 215–224, Montreal, Canada, October 2003. IEEE Computer Society. doi:10.1109/ASE.2003.1240309.

[171] Jason McC Smith and David Stotts. SPQR: Formalized design pattern detection and software architecture analysis. Technical Report TR05-012, Dept. of Computer Science, University of North Carolina, Chapel Hill, NC, USA, May 2005. URL: http://rockfish.cs.unc.edu/pubs/TR05-012.pdf.

[172] Software Composition Group. FM3. Web site, 2009. URL: http://scg.unibe.ch/wiki/projects/fame/fm3.

[173] Software Composition Group. MSE. Web site, 2009. URL: http://scg.unibe.ch/wiki/projects/fame/mse.

[174] Software Composition Group. Moose. Web Site, 2010. URL: http://www.moosetechnology.org/.

[175] Software Engineering Research Center. Research projects: Design patterns. Web Site, 2012. URL: http://research.ciitlahore.edu.pk/Groups/SERC/DesignPatterns.aspx.

[176] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. The Eclipse Series. Addison-Wesley Professional, 2nd edition, December 2008. URL: http://www.informit.com/store/product.aspx?isbn=9780321331885.

[177] Krzysztof Stencel and Patrycja Wegrzynowicz. Detection of diverse design pattern variants. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference*, APSEC '08, pages 25–32, Beijing, China, December 2008. IEEE Computer Society. doi:10.1109/APSEC.2008.67.

[178] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 36(2):111–147, 1974. URL: http://www.jstor.org/stable/2984809.

[179] Detlef Streitferdt, Christian Heller, and Ilka Philippow. Searching design patterns in source code. In *Proceedings of the 29th Annual International Computer Software and Applications Conference*, COMPSAC '05, pages 33–34, Hong Kong, China, September 2005. IEEE Computer Society. doi:10.1109/COMPSAC.2005.135.

[180] The Eclipse Foundation. Eclipse Java development tools (JDT). Web site, 2011. http://www.eclipse.org/jdt/. URL: http://www.eclipse.org/jdt/.

[181] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF). Web Site, 2011. URL: http://www.eclipse.org/modeling/emf/.

[182] The Eclipse Foundation. EclipseLink. Web site, 2011. http://www.eclipse.org/eclipselink/. URL: http://www.eclipse.org/eclipselink/.

[183] The Eclipse Foundation. GEF (Graphical Editing Framework). Web site, 2011. URL: http://www.eclipse.org/gef/.

[184] The Eclipse Foundation. JFace. Web site, 2011. http://wiki.eclipse.org/JFace. URL: http://wiki.eclipse.org/JFace.

[185] The Eclipse Foundation. SWT: The Standard Widget Toolkit. Web site, 2011. http://www.eclipse.org/swt/. URL: http://www.eclipse.org/swt/.

[186] The Eclipse Foundation. Eclipse, 2012. URL: http://www.eclipse.org/.

[187] The XML Schema Working Group. XML Schema. Web Site, 2004. http://www.w3.org/XML/Schema. URL: http://www.w3.org/XML/Schema.

[188] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.

[189] Sander Tichelaar. FAMIX 2.2. Web page, 2012. URL: http://www.moosetechnology.org/docs/others/famix2.2.

[190] Sander Tichelaar. Famix 3.0 core beta. Web page, 2012. URL: http://www.moosetechnology.org/docs/others/famix3.0.

[191] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 4:501–520, August 1994. doi:10.1142/S0218194094000246.

[192] Christian Tosi, Marco Zanoni, Francesca Arcelli, and Claudia Raibulet. Joiner: from subcomponents to design patterns. In *Proceedings of the DPD4RE Workshop, co-located event with IEEE WCRE 2006 Conference*, Benevento, Italy, 2006. IEEE Computer Society.

[193] Christian Tosi, Marco Zanoni, and Stefano Maggioni. A design pattern detection plugin for eclipse. In Angelo Gargantini, editor, *Proceedings of the 4th Italian Workshop on Eclipse Technologies (Eclipse-IT 2009)*, Bergamo, Italy, September 2009. Eclipse Italian Community. URL: `http://eit09.unibg.it/pdfs/99990089.pdf`.

[194] Nikolaos Tsantalis. Dpd tool results. Web Site, 2006. `http://java.uom.gr/~nikos/pattern-detection.html`.

[195] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006. `doi:10.1109/TSE.2006.112`.

[196] University of Waikato. Weka. Web site, 2009. URL: `http://www.cs.waikato.ac.nz/ml/weka/`.

[197] C. J. van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 2nd edition, 1979. URL: `http://www.dcs.gla.ac.uk/Keith/Preface.html`.

[198] W3C. SPARQL query language for RDF. Web site, January 2008. URL: `http://www.w3.org/TR/rdf-sparql-query/`.

[199] Wei Wang and Vassilios Tzerpos. Design pattern detection in eiffel systems. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 165–174, Pittsburgh, Pennsylvania, USA, November 2005. IEEE Computer Society. `doi:10.1109/WCRE.2005.14`.

[200] Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis*, WODA 2003, pages 29–32, Portland, Oregon, USA, May 2003. Jonathan Cook, New Mexico State University. URL: `http://www.cs.nmsu.edu/~jcook/woda2003/woda2003.pdf`.

[201] Rebecca J. Wirfs-Brock. Valuing design repair. *IEEE Software*, 25(1):76–77, January-February 2008. `doi:10.1109/MS.2008.26`.

[202] Yiling Yang, Xudong Guan, and Jinyuan You. CLOPE: a fast and effective clustering algorithm for transactional data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '02, pages 682–687, Edmonton, Alberta, Canada, July 2002. ACM. `doi:10.1145/775047.775149`.

[203] Marco Zanoni. MARPLE: discovering structured groups of classes for design pattern detection. Master's thesis, Università degli studi di Milano-Bicocca, Milano, Italy, July 2008.

[204] Zhi-Xiang Zhang, Qing-Hua Li, and Ke-Rong Ben. A new method for design pattern mining. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics*, volume 3 of *ICMLC 2004*, pages 1755–1759, Shanghai, China, August 2004. IEEE Computer Society. `doi:10.1109/ICMLC.2004.1382059`.