BABEŞ-BOLYAI UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Machine learning based software development

PhD Thesis Abstract

PhD student: Zsuzsanna-Edit Marian
Scientific supervisor: Prof. Dr. Gabriela Czibula

September 2014

List of publications

Publications in ISI Web of Knowledge

Publications in ISI Science Citation Index Expanded

1. [CMC14b] Gabriela Czibula, **Zsuzsanna Marian** and Istvan-Gergely Czibula, Software defect prediction using relational association rule mining. *Information Sciences* published by *Elsevier*, Vol. 264, pp. 260-278, DOI: 10.1016/j.ins.2013.12.031, 2014. (**IF = 3.643**)

2. [CMC14a] Gabriela Czibula, **Zsuzsanna Marian** and Istvan-Gergely Czibula. Detecting software design defects using relational association rule mining. *Knowledge and Information Systems* published by *Springer*, DOI: 10.1007/s10115-013-0721-z, published on-line in January, 2014. (**IF = 2.225**)

3. [MCC12] **Zsuzsanna Marian**, Gabriela Czibula and Istvan-Gergely Czibula. Using software metrics for automatic software design improvement. *SIC Journal, Studies in Informatics and Control*, Romania, Vol. 21, Number 3, pp. 249-258, 2012. (**IF = 0.578**)

4. [MCC14] **Zsuzsanna Marian** Gabriela Czibula and Istvan-Gergely Czibula, Software packages refactoring using a hierarchical clustering-based approach. *Fundamenta Informaticae*, Under review, 2014. (**IF = 0.399**)

Publications in ISI Conference Proceedings Citation Index

5. [SMV09] Adrian Sterca, **Zsuzsanna Marian** and Alexandru Vancea. Distortion-based media-friendly congestion control. *Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques*, Cluj-Napoca, Romania, pp. 265-267, 2009.

6. [ŢIM09] Radu Ţurcaş, Oana Iova and **Zsuzsanna Marian**. The autonomous robotic tank (ART): an innovative lego mindstorm NXT battle vehicle. *Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques*, Cluj-Napoca, Romania, pp. 95-98, 2009.

Papers published in international journals and proceedings of international conferences

7. [Mar12a] **Zsuzsanna Marian**. Aggregated metrics guided software restructuring. *Proceedings of 8th IEEE International Conference on Intelligent Computer Communication and Processing*, Cluj-Napoca, Romania, pp. 259-266, 2012. (**Indexed IEEE**)

8. [Mar12c] **Zsuzsanna Marian**. A study on hierarchical clustering based software restructuring. *Studia Universitatis "Babes-Bolyai", Informatica*, Romania, Vol. LVII, Number 2, pp. 20-31, 2012. (**Indexed MathSciNet**)

9. [Mar13b] **Zsuzsanna Marian**. A study on Relational Association Rule Mining Based Software Design Defect Detection. *Studia Universitatis "Babes-Bolyai", Informatica*, Romania, Vol. LVIII, Number 1, pp. 42-57, 2013. (**Indexed MathSciNet**)

10. [Mar13a] **Zsuzsanna Marian**. On the software metrics influence in Relational Association Rule-based Software Defect Prediction. *Studia Universitatis "Babes-Bolyai", Informatica*, Romania, Vol. LVIII, Number 4, pp. 35-48, 2013. (**Indexed MathSciNet**)

11. [Mar14b] **Zsuzsanna Marian**. On evaluating the structure of software packages. *Studia Universitatis "Babes-Bolyai", Informatica*, Romania, Vol. LIX, Number 1, pp. 46-58, 2014. (**Indexed MathSciNet**)

12. [Mar14a] **Zsuzsanna Marian**. FAOS - A framework for analyzing object-oriented software systems. *Studia Universitatis "Babes-Bolyai", Informatica*, Romania, Under review, 2014. (**Indexed MathSciNet**)

13. [MS10] **Zsuzsanna Marian** and Christian Săcărea. Using contextual topology to discover similarities in modern music. *Proceedings of the IEEE International Conference on Automation Quality and Testing, Robotics*, Cluj-Napoca, Romania, vol. 3, pp. 1–6, 2010. (**Indexed IEEE**)

14. [TLM11] Doina Tatar, Mihaiela Lupea and **Zsuzsanna Marian**. Text summarization by formal concept analysis approach. *Studia Universitatis "Babes-Bolyai", Informatica*, Vol. LVI, Number 2, pp. 7-12, 2011. (**Indexed MathSciNet**)

15. [MCB11] **Zsuzsanna Marian**, Cosmin Coman and Attila Bartha. Learning to play the guessing game. *Studia Universitatis "Babes-Bolyai", Informatica*, Vol. LVI, Number 2, pp. 119–124, 2011. (**Indexed MathSciNet**)

Papers published in proceedings of national conferences

16. [Mar12b] **Zsuzsanna Marian**. Software metrics based refactoring: a case study. *Proceedings of the National Symposium ZAC 2012*, Cluj-Napoca, pp. 59-64, 2012.

17. [Mar10] **Zsuzsanna Marian**. Solving the subset sum problem with DNA computation, *Proceedings of the National Symposium ZAC 2010*, Cluj-Napoca, pp. 25-29, 2010.

In the case of co-authored publications, we have contributed to: conceiving, designing and performing the experiments; analyzing the data; developing the structure and arguments of the papers; writing the manuscripts; making critical revisions and approving the final versions of the papers.

Keywords

- Machine Learning
- Clustering
- Software metrics
- Relational Association Rules
- Software Development
- Software Restructuring
- Software Defect Detection
- Design Defect Detection
- Defect Prediction
- Nasa datasets
- Software Frameworks

Contents

List of publications **1**

Introduction **5**

1 Machine Learning in Software Engineering **8**
1.1 Computational Intelligence in Software Engineering 8
1.2 Software Remodularization 9
1.3 Software Defect Detection 10
1.4 A Background on Related Machine Learning Methods 11

2 New Approaches to Software Remodularization **13**
2.1 Package-Level Remodularization 13
2.2 Class-Level Remodularization 16
2.3 Conclusions and Further work 20

3 New Approaches to Software Defect Detection **21**
3.1 Theoretical model 21
3.2 Design Defect Detection 22
3.3 Defect Prediction 26
3.4 Conclusions and Further work 28

4 A Software Framework for Analyzing Object-Oriented Software Systems **29**
4.1 The FAOS framework 29
4.2 Comparison to similar frameworks 30
4.3 Conclusions and Further work 30

Conclusions **31**

Bibliography **32**

Introduction

This Ph.D. thesis is the result of my research in the field of developing *machine learning models for solving different problems related to the software development process.* This research was started in 2011, under the supervision of Prof. Dr. Gabriela Czibula.

The main research direction we are focusing on is the application of machine learning methods and algorithms on different tasks and problems from software engineering. The problems approached in this thesis are of major importance during the maintenance and evolution of software systems. Solutions to these problems would help software developers by assisting them in different software development tasks.

Machine Learning (ML) is a branch of Artificial Intelligence which seeks to answer the question *"How can we build computer systems that automatically improve with experience, and what are the fundamental laws that govern all learning processes?"* [Mit06]. Consequently, ML is composed of a series of different approaches and algorithms that can learn from existing data.

In 2001 with a paper of Mark Harman and Brian Jones a new research field, *Search-Based Software Engineering* (*SBSE*), was born [HJ01]. This paper describes that many software engineering problems can be formulated as search problems, and search algorithms can be applied to them. After the publication of this paper many researchers started working in this direction and soon they realized that search methods are not the only intelligent approaches possible for software engineering tasks. They realized that there are software engineering tasks which can be easily formulated as clustering problems and they started applying clustering algorithms for them. They also found that there are problems that can be formulated as classification or prediction problems to which different classifiers can be applied.

But even if intelligent methods *can* be applied to software engineering tasks, *why* should they be applied? Why should we develop approaches for solving tasks that can be solved by software developers? The answer is that these tasks cannot always be solved by software developers and developers can use the help provided by these approaches. Due to the complexity and size of software systems in our days, there are many situations when people who work on a system might not see through all the relations between elements, they might not see which parts should be improved, or which parts should be focused on during testing. This is why search-based, computational intelligence-based and machine learning-based approaches are all welcome: they can help software developers with their everyday tasks.

Out of the different software engineering tasks on which intelligent methods were applied, we have chosen two main directions to focus on in this thesis. The first direction was to develop approaches for restructuring object-oriented software systems using clustering algorithms both on package level and class level. The second main direction tackled in this thesis is the problem of defect detection in a software system. This direction can be divided into design defect detection and defect prediction, and for each problem we have introduced a novel relational association rule mining-based approach. Finally, an original contribution towards the development of software systems is presented as well, in form of a software framework, called *FAOS* (*Framework for Analyzing Object-oriented Software systems*). This framework was developed to support the experimental evaluation of most approaches presented in this thesis.

The thesis is structured in four chapters, as follows.

The first chapter, **Machine Learning in Software Engineering. Background**, starts with a short presentation of different computational intelligence methods in software engineering. This is followed by the presentation of the two main research directions approached in this thesis: *Software Remodularization* and *Software Defect Detection.* For both directions the two problems that we have worked on is detailed, together with a short review of existing methods. Finally, we give a short overview of two important machine learning methods that we have used in our approaches: relational association rules and clustering.

Our original contributions are presented in Chapters 2, 3 and 4, where we describe the machine learning models that we have proposed for solving the software engineering tasks that we have chosen to focus on in this thesis.

Chapter 2, **New Approaches to Software Remodularization**, is original and presents our work for the first main research direction of this thesis, namely software restructuring using clustering algorithms. We begin with the first problem from this domain: package-level remodularization. We present two algorithms, one which identifies a good division of the classes into packages and one that, given an existing package structure of the software system, can identify the suitable package for a newly added application class. This is followed by the experimental evaluation of these two algorithms, an analysis of the approach and a comparison to existing methods from the literature for package-level restructuring. The second problem tackled in Chapter 2, software restructuring at the class level, is presented next. We present three algorithms that can identify an improved class structure of a software system, together with an experimental evaluation of them. Finally, the three algorithms are first compared to each other, then to existing approaches from the literature. The last section presents the conclusions of this chapter together with future work directions.

Chapter 3, **New Approaches to Software Defect Detection** is original and it presents our approaches for the second main research direction of this thesis, software defect detection. Both problems from this direction use the same theoretical model, which is described in the first section. This is followed by the introduction of our approach for identifying classes with design defects in a software system, using relational association rules. Next, the experimental evaluation of this approach and a comparison to existing approaches is given. We also present a study on the effects of parameter variations on our approach. The second part of Chapter 3, presents our approach for the second problem from this direction, namely software defect prediction. We describe our original approach, a classifier, which can classify entities from a software system as defective or not defective using relational association rules. The experimental evaluation and a comparison to existing approaches is given next, followed by two studies on the classifier. Finally we present the conclusions of this chapter.

Chapter 4, **A Software Framework for Analyzing Object-Oriented Software Systems** is original and it presents the *FAOS* (*Framework for Analyzing Object-oriented Software systems*) framework, a general framework written in Java for analyzing object-oriented software systems. The chapter presents the three main modules of the framework, one for the analysis of compiled Java code and the extraction a list of elements (classes, methods, fields) and the relations between them, one for computing the value of different software metrics, and one for the implementation of our approach for package-level restructuring of a software system. After the presentation of the modules, a short comparison to similar frameworks and tools is given. Conclusions and directions for further work are drawn in the last section.

The original contributions introduced in this thesis are presented in Chapters 2, 3 and 4 and they are the following.

- A novel clustering-based approach for software remodularization at the package level - the *CASP* (*Clustering Approach for Software Package Restructuring*) approach (Subsection 2.1.1) together with a hierarchical agglomerative clustering-based algorithm, *HASP*(*Hierarchical Clustering Algorithms for Software Packages Restructuring*), to be used in the *CASP* approach (Subsection 2.1.2) [MCC14].

  - The definition of seven features, which can be aggregated into a single value (called *overallScore*), which can be used as a similarity measure in the *HASP* algorithm (Subsection 2.1.2) and the definition of a new measure, *CIP*(*Cohesion of Identified Packages*), which computes how close a given partition is to an apriori known good partition (Subsection 2.1.4). [MCC14].
  - The *AssignClass* algorithm, which can find the suitable package in a software system for a newly added application class, using the same features and score as the *HASP* algorithm (Subsection 2.1.3) [MCC14].
  - An experimental evaluation of the *HASP* and *AssignClass* algorithms (Subsection 2.1.4), an analysis of the provided results (Subsection 2.1.5) and a comparison of the *overallScore* measure to other existing measures (Subsection 2.1.6). [MCC14, Mar14b].

- Three novel algorithms for software systems restructuring at the class level: *ARI*, *HAC* (Subsection 2.2.1) and an aggregated metrics-based approach (Subsection 2.2.2) [MCC12, Mar12c, Mar12b, Mar12a].

  - Experimental evaluation of the three algorithms for class-level restructuring of a software system, and a comparative analysis of them (Subsection 2.2.3).

- A method for design defect detection based on relational association rules discovery - *SDDRAR* (Subsection 3.2.2) [CMC14a].

  - A novel *Apriori*-like [AS94] algorithm, for mining arbitrary length relational association rules from a dataset - *DRAR* (Subsection 3.2.1) [CMC14a].
  - A detailed experimental evaluation of the *SDDRAR* approach (Subsection 3.2.3) followed by an analysis of the approach with respect to its robustness, precision, recall, advantages and disadvantage (Subsection 3.2.4) [CMC14a].
  - A study on the effect of parameter variations on the results provided by the *SDDRAR* approach (Subsection 3.2.5) [Mar13b].

- A novel binary classification approach, which, using relational association rules, can classify entities from a software system as defective or non-defective - *DPRAR* (Subsection 3.3.1) [CMC14b].

  - A thorough experimental evaluation of the *DPRAR* approach, using as case study 10 NASA datasets (Subsection 3.3.2) and a comparison of the results to the results reported in the literature for the same datasets (Subsection 3.3.3) [CMC14b].
  - Studies on the effect of feature elimination and changing the score computation formula on the results of the *DPRAR* approach (Subsection 3.3.4) [Mar13a].

- A generic software framework, *FAOS*, for the analysis of object-oriented software systems. (Section 4.1) [Mar14a]. This framework was implemented to support the original approaches that we have developed, and it was used for almost all of the above presented methods.

Chapter 1

Machine Learning in Software Engineering. Background

Applying intelligent methods to software engineering tasks is a relatively new research field, started in 2001 with a paper of Mark Harman and Bryan Jones, [HJ01]. Since then, the application of intelligent methods for software engineering tasks has come a long way. In [ZT05] the authors present 44 different tasks for which at least one machine learning algorithm was applied. Similarly, a review of current trends in *SBSE*, [HMZ09], gathers more than 500 publications in the domain.

1.1 Computational Intelligence in Software Engineering

In 2001 Mark Harman and Bryan Jones published a paper, [HJ01], which is considered to be the manifesto of a new research field, called *Search-Based Software Engineering* (*SBSE*). They present that metaheuristic search-based optimization techniques (namely Genetic Algorithms, Simulated Annealing and Tabu Search) have been applied in different engineering fields and they would be suitable for software engineering (SE), too.

In the following years, a large number of approaches were presented in the literature, approaches consisting of the application of search algorithms for different software engineering tasks. Eight years after the appearance of the field, in 2009, Harman et al. published a review of the existing trends, techniques and applications in *SBSE* [HMZ09] based on more than 500 publications in the field.

Search algorithms are not the only intelligent methods applied to software engineering tasks, as presented by Harman in [Har12], where he mentions three broad areas of Artificial Intelligence techniques used by the software engineering community: computational search and optimization, fuzzy and probabilistic methods and classification, learning and prediction.

Besides [Har12], another recent surveys on the role of Artificial Intelligence in Software Engineering, [AAH12], presents that Computational Intelligence methods have been applied to requirement engineering, software architectural design, software coding and testing, software reliability, software cost estimation and prediction and automatic bug fixing. In [DK05] Dick and Kandel present examples of how CI and ML methods can be applied to software quality assurance.

Machine learning algorithms have also been applied to different software engineering tasks. In [ZT05] the authors present 44 different software engineering problems to which at least one ML algorithm has been applied.

1.2 Software Remodularization

A software system is either changed over its lifetime to adapt to different requirements or it will no longer be used. Changes to a software system after its release are done during maintenance, and if only changes that add new functionalities, or correct existing errors are done, the system will become harder and harder to maintain in the future. In order to prevent this, another important activity to do during maintenance (and not just then) is restructuring the code. Restructuring is the activity when the internal structure of a software system is changed (improved), without affecting the system's external behavior. For object-oriented systems the term *refactoring* is often used as a synonym for restructuring, but, according to Fowler, a refactoring is a simple transformation, performed in a couple of minutes, while restructuring is more complex and can last longer (and it can be composed of a series of refactorings) [Fow].

Software remodularization can be done on different levels of the software system. For an object-oriented software system we can talk about method-level, class-level, package-level and architecture-level remodularization.

1.2.1 Package-Level Remodularization

Problem statement and relevance. Nowadays software systems are becoming more and more complex, consisting of thousands of application classes which are grouped into software packages. Without an appropriate package structure of the software, the system becomes hard to maintain. Thus, *software remodularization* at the package level is an important process in software maintenance and evolution. The more complex the system the higher its maintenance cost is. Therefore it is very hard for software developers to decide the appropriate package structure for the system. When the number of application classes is large, the class assignment decision is not an easy one, since it involves a good knowledge of the overall system design.

The problem of software packages restructuring arises from practical needs, thus intelligent approaches can be useful for helping software developers in their daily works of restructuring packages in software systems.

Literature review. There are several different methods reported in the literature for identifying how classes should be grouped into packages. One such method is presented in [AAM11], where clustering is used to find the ideal grouping of classes. A constrained community detection algorithm-based approach is presented in [PJL13]. Even if it can not restructure a whole system, the method presented by Bavota et al. in [BLMO10] can divide a package, which has a low cohesion, into several, more cohesive packages. Search-based methods were also proposed for this problem, for example [MHH03, MMCG99].

Another direction of research that should be mentioned, is the definition of different metrics, which measure the quality of packages in a software system. Such metrics are defined in [SKR08] and in [DABH11].

1.2.2 Class-Level Remodularization

Problem statement and relevance. Most of the approaches presented in the literature of software remodularization identify or perform restructurings at the class level. These approaches try to identify those classes in a software system, which do not have a good design and they also identify those refactorings which would improve the design of these classes.

An excellent and frequently cited book, when it comes to refactoring is *Refactoring: Improving the Design of Existing Code* written by Martin Fowler [Fow99]. In this book Fowler presents, through a set of examples, why refactoring is important and how to do it.

He also presents a catalog of 72 different refactorings (each with description, motivation and example. Even with such a complete catalog of possible refactorings, there are two important questions: when and where to refactor. To answer these questions, Fowler introduces the notation of *bad smells*, which are *"certain structure in the code that suggest (sometimes they scream for) the possibility of refactoring"* [Fow99]. In the book there is also a catalog of 22 different bad smells and for each of them, the list of refactorings which can remove them is also presented.

Literature review. The identification of refactoring opportunities in a software system is a well-researched domain. Different methods were proposed in the literature which use Game Theory [BOL$^+$10, HT07], Concept Analysis [ST98], Bayesian Belief Networks [KVGS09], clustering [RR11, FTCS09, AL99] or different search methods [MC11, OC08, GWI11, SSB06, HT07]. While most of the previous techniques also use software metrics to different extents, there are also methods based on software metrics alone [ISIE12, SS07, HKI08, CLMM05, SSL01, TK03, DDN00].

1.3 Software Defect Detection

Software Defect Detection is the branch of software engineering which tries to identify or predict defects in a software system. It is an important activity, according to [Kan03], defect detection and correction can consume about 75 % of total software life-cycle costs. Kandt in [Kan03] presents two important and complimentary defect detection techniques: inspection and testing. Both require a considerable amount of time and human effort, which, especially in case of large software systems, is not always available. This lack of time, combined with the increasing size of software systems, is the reason for which intelligent methods that could help developers and software testers focus their attention on those parts of the system where the problems can be found are developed.

1.3.1 Design Defect Detection

Problem statement and relevance. The internal structure of the entities within a software system can change many times during the system life-cycle and thus has a major impact on the maintainability of the system. Even when a software system originally has a good design, during maintenance, the design of the system can degrade, which, in turn, makes further maintenance harder and more costly. This is why monitoring the design of the system and continuously identifying and correcting design defects is essential.

Software design defect detection is closely related to another software engineering task, software restructuring or refactoring. Usually, the solution for eliminating design defects is to restructure the software system, possibly by applying different refactorings.

What are design defects? In order to be able to detect design defects, one has to define what a design defect is. When defining what exactly is a design defect, one can start from a set of principles for a correct design, and look for violations of these principles. For example, Larman in [Lar04] presents nine such principles (or patterns), known under the name of *GRASP* (*General Responsibility Assignment Software Patterns*) patterns.

Design defects are often divided into two categories: higher-level design defects (also called antipatterns) and lower-level design defects (also called bad smells).

Literature review. Manually identifying design errors is at least time-consuming, if not impossible, for large software systems, so in the literature different approaches are presented for automatically finding them. Such approaches are Marinescu's *detection strategies* [Mar02], Moha's *rule cards* [MGL06, Moh06], the *JDeodorant* Eclipse plug-in [FTSC11] and Munro's metric-based approach [Mun05].

The application of search techniques in design defect detection is presented in [KSBW11], while in [MKD13] multi-objective genetic programming approach to find three types of bad smells is described. The use of change history to detect bad smells is presented in [PBP$^+$13].

Fontana et al. present in [FBZ12] a comparison of different code smell detection tools. They mention several existing commercial or freely available tools, but use for experiments only four of them. These tools are applied on different systems to detect six types of smells. One conclusions that they draw in this paper is that different tools, when applied to the same source code, for finding the same bad smell will produce different results (the only exception was the *God Class* smell).

1.3.2 Defect Prediction

Problem statement and relevance. Software defect prediction tries to pinpoint those parts of a software system, where errors are present, so that software testers can spend more time testing those components which are likely to contain errors, and less time on the ones which are probably error-free. Many software defect predictors use *software metrics*to measure the software quality in order to predict software defects. Thus, *software defect prediction* is the task of classifying software modules into the fault-prone and the non-fault-prone ones by using metric-based classification [BMW02]. Most defect prediction techniques used in planning are based on historical data, hence rely on supervised classification.

The NASA datasets. Many approaches for defect prediction from the literature use for evaluation the so-called *NASA datasets*. These can be found at [Nas], which is a software engineering repository where the datasets are available to anyone in order to help researchers create repeatable, verifiable, and improvable predictive models of software engineering.

Literature review. One of the early software defect prediction methods is the *CBA* method, presented in [LHM98], where class association rules are used. [LMW01] presents an extension of this method, *CBA2*, which tries to solve the data imbalance problem. A hybrid model, combining association rule mining and logistic regression is presented in [KMMiM08]. Another rule-based method is, *EDER-SD*, which is based on evolutionary computation and generates rules describing only fault-prone modules [RRRAR12].

Besides rule-based methods, many different machine learning algorithms have been applied to the problem of defect prediction. One such work is [MGF07], in which they evaluate the Naive Bayes classifier, OneR and J48. Challagulla et al. evaluate in [CBYP05] some predictor models on four *NASA* datasets. Haghighi et al. provide in [HDF12] a comparative analysis of 37 different classifiers in fault detection systems and use the *NASA* datasets for performing experiments.

Another direction is the use of disagreement-based semi-supervised learning methods. Such approaches are *ROCUS* [JLZ11], *ACoForest* [LZWZ12] and the use of Random Forests presented in [GMCS04].

1.4 A Background on Related Machine Learning Methods

1.4.1 Relational Association Rules

Association rule mining of itemsets in large databases of transactions was first described by Agrawal et. al in 1993 [AIS93]. An *association rule* is an implication of the form $X \Rightarrow Y$, where X and Y are disjoint sets of items and it means that the presence of the itemset X implies the presence of the itemset Y as well.

One of the disadvantages of association rules is that they are only interested in the presence or absence of an item in an itemset, but they completely ignore any possible relationships that might exist between these items. In order to overcome this disadvantage, Marcus et al. proposed in [MML01], an extension of association rules, called *ordinal association rules*, where ordinal relations are defined between the elements of an itemset.

Still, in datasets ordinal relationships are not always sufficient, many different, not ordinal, relationships can exist between items. This is why *relational association rules*, an extension of ordinal association rules, were introduced in [SCC06]. Relational association rules allow any kind of relation between elements of an itemset, not just ordinal ones.

Formally, as presented in [SCC06], let us consider that $R = \{r_1, r_2, \ldots, r_n\}$ is a set of *instances*. Each instance from R is characterized by a list of m attributes, $(a_1, \ldots, a_m)$. The value of attribute a_i for the instance r_j is denoted by $\Phi(r_j, a_i)$. For each attribute a_i there is a domain D_i, where its values come from. This domain contains the empty value denoted by ε as well. Between two domains D_i and D_j relations can be defined (not necessarily ordinal relations), such as *less or equal* ($\leq$), *equal* ($=$) and *greater or equal* ($\geq$). The set of all possible relations that can defined on D_i x D_j is denoted by M.

Definition 1 *[SCC06] A* relational association rule *is an expression* $(a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}) \Rightarrow (a_{i_1}\ \mu_1\, a_{i_2}\ \mu_2\ a_{i_3} \ldots \mu_{\ell-1}\ a_{i_\ell})$*, where* $\{a_{i_1}, a_{i_2}, a_{i_3}, \ \ldots, a_{i_\ell}\} \subseteq \mathcal{A} = \{a_1, \ldots, a_m\}$, $a_{i_j} \neq a_{i_k}$, $j, k = 1..\ell$, $j \neq k$ *and* $\mu_i \in M$ *is a relation over* $D_{i_j} \times D_{i_{j+1}}$, D_{i_j} *is the domain of the attribute* a_{i_j} *if the attributes* $a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}$ *occur together (are non-empty) in* $s\%$ *of the* n *instances (we call* s *the* support *of the rule) and we denote by* $R' \subseteq R$ *the set of instances where* $a_{i_1}, a_{i_2}, \ldots, a_{i_l}$ *occur together and* $\Phi(r_j, a_{i_k})\ \mu_k\ \Phi(r_j, a_{i_{k+1}})$ *holds for each* $1 \leq k \leq l-1$ *for each instance* r_j *from* R' *(we call* $c = |R'|/|R|$ *the* confidence *of the rule).*

Besides support and confidence, a rule can also by characterized by its length, which is the number of attributes in it. In a dataset a great number of relational association rules can be found, but we are usually interested only in the rules whose support and confidence is greater than some user provided threshold s_{min} and c_{min}. Such rules are called interesting.

1.4.2 Clustering

Clustering is an unsupervised learning method, considered the most important one, whose main goal is to group entities from a data set in different groups (called clusters) in a way in which entities in a cluster are more similar to each other than to those outside their cluster.

Let $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$ be the set of objects to be clustered. During the clustering process, a *cluster* C is a nonempty set of entities from $\mathcal{O}$, which belong together based on a distance function d. A *partition* $P = (C_1, C_2, ..., C_k)$ is a set of clusters, such that each entity from $\mathcal{O}$ belongs to one and only one cluster from P.

Clustering algorithms can in general be divided into two big groups: *hierarchical* and *partitional* algorithms, while hierarchical clustering algorithms can further be divided into *agglomerative* or *divisive* algorithms [Han05]. In case of an agglomerative clustering algorithm the process starts with a partition where each entity is in its own cluster and the number of clusters is decreased until all entities are in the same cluster.

An important step in hierarchical clustering is determining the distance between two clusters, denoted by $dist(C_i, C_j)$, because this distance decides which clusters will be merged. The most well-known distance metrics for two clusters in the literature are: *simple-link* - where the minimum distance between two entities from C_i and C_j is considered -, *complete-link* - where the maximum distance is considered - and *average-link*, where the average distance is considered.

Partitional clustering algorithms [Han05] instead of creating a series of partitions (as hierarchical algorithms do), create a single partition containing K clusters. In this case, the algorithm starts with K initial cluster centers, and each entity is added to the cluster which is closest to it. Then the cluster centers are recomputed the entities are added again to the closest cluster. These steps are usually repeated until the clusters become stable.

Chapter 2

New Approaches to Software Remodularization

In this chapter, which is entirely original, we are approaching the problem of software remodularization. Software remodularization means changing the internal structure of a software system, without affecting its external behavior, and it is an important activity, because it helps maintain a good design and modularization of a software system. In this chapter we will present our approaches for package-level remodularization and class-level remodularization, using hierarchical clustering.

The approaches presented in this chapter are original works published in [Mar14b], [MCC12], [Mar12c], [Mar12b], [Mar12a], or under review: [MCC14].

Motivation. Software systems nowadays have to adapt to the changing requirements of the users, or to the changing environment in which they are used. Most changes related to adaptation are done after the release, and in many cases affect negatively the quality of the software system, which can make further maintenance more complicated and costly. In order to avoid these costs, maintaining the quality of the system is also an important goal, but due to the size of most software systems it is almost impossible for a developer to see through the whole structure of the system, and decide about the necessary modifications. This is why different methods and tools which determine which refactorings should be applied for improving the quality of the systems are gaining importance.

We have started from the idea that a software system can be considered a data set, containing records, which can be elements of the software system. In such a setting the application of a clustering algorithm to group together those entities that belong together seems natural. To guide the clustering process, we have decided to use software metrics, because they have been proven useful in other approaches.

2.1 Package-Level Remodularization

In this section we will present our original approach for package-level restructuring of a software system. It takes an existing software system and remodularizes it at the package level using hierarchical clustering, in order to obtain better-structured packages. Considering an existing package structure of a software system, the method proposed in this section would also be useful for suggesting the developer the appropriate package for a newly added application class.

2.1.1 The *CASP* approach

In this subsection we introduce the clustering-based approach (*CASP - Clustering Approach for Software Packages Restructuring*) for package-level software remodularization, which consists of two steps:

- **Data collection** - The existing software system is analyzed in order to extract from it relevant information about application classes, methods, attributes and the existing relationships between them.
- **Grouping** - The set of classes from the software system, considering the relevant information extracted at the previous step, are grouped in clusters (packages) using a clustering algorithm (*HASP* in our approach). The goal of this step is to obtain a partitioning of the software system into packages.

2.1.2 Grouping into packages

In the following we introduce a novel hierarchical agglomerative clustering algorithm (*HASP - Hierarchical Clustering Algorithm for Software Packages Restructuring*), which aims at identifying a partition of a software framework S, that corresponds to a good structure of packages of the software system. From the partition created with this algorithm, each cluster will correspond to a package of the software system.

We have identified a set of seven different features that we considered relevant for the package clustering problem. These features measure the cohesion, coupling, reuse potential of a possible package, but also how similar the elements of a package are. These seven features were aggregated into a single value, called *score* in the following way:

$$score(K_i \bigcup K_j, \mathcal{K}^*) = \frac{\sum\limits_{i=1}^{2} w_i \cdot F_i - w_3 \cdot F_3}{|K_i \bigcup K_j|^2 - 1} + \sum_{i=4}^{7} w_i \cdot F_i \qquad (2.1)$$

where K_i and K_j represent the two clusters that are going to be merged, $\mathcal{K}^*$ represents the partition without K_i and K_j, F_i, $1 \le i \le 7$, are the values of the seven features and w_i ($0 \le w_i \le 1$) are the weights associated to these features. These weights were determined using a grid search method and a software system for which a good package structure is known. To guide the grid search process we have introduced the *CIP* (*Cohesion of Identified Package*) measure, which computes how close two partitions of a software system are to each other.

HASP is based on the idea of hierarchical agglomerative clustering. At a given step, the pair of clusters that have the maximum associated *score* are merged. The agglomerative hierarchical clustering process is performed until a single cluster is obtained and creates a series of partitions with decreasing numbers of clusters. In order to identify the "best" partition from all the generated partitions, we introduced a new measure, *overallScore*, which is computed considering the value of a slightly modified version of the *score* measure.

2.1.3 Assigning application classes to packages

Another problem that software engineers can often encounter is finding the suitable package for a newly added application class. For solving this problem we have introduced the *AssignClass* algorithm. Let us consider that $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ is the actual partition into packages of the software system S. A new application class C is added to the system. In order to find the best package for the application class C, we compute the value of $score(K_i \bigcup C,\ \mathcal{K}^*)$ for every current package $K_i \in \mathcal{K}$. The package where this *score* is the maximum is the one where C should be placed.

2.1.4 Experimental evaluation

For the experimental evaluation of our approach, we used two open-source software frameworks, *Commons DbUtils* [DbU] and a *Reinformcement Learning* framework [rlf]. For both frameworks we performed two experiments: we executed the *HASP* algorithm, to identify a good division of application classes into packages, and we identified the suitable package for some application classes using the *AssignClass* algorithm.

2.1.4.1 Commons DbUtils. Results

We applied the *HASP* algorithm on the *DbUtils* framework and it returned a partition with 4 clusters (packages) as the optimal one, instead of the original structure, composed of 3 packages. Analyzing the results of the $HASP$ algorithm, we concluded that the partition suggested by the *HASP* algorithm is better than the original one.

During the second experiment, we repeatedly executed the *AssignClass* algorithm, each time removing one class from *DbUtils* and trying to find the suitable package where it should go. Out of 19 runs in 16 cases it identified the correct package, which is an accuracy of 84.2 %.

2.1.4.2 Reinforment Learning. Results

We applied the *HASP* algorithm for the Reinforcement Learning framework and it returned a partition with 11 packages instead of the original correct partition with 10 packages. Still, the two partitions are very similar so we computed the value of the *CIP* metric to see how close the partition generated by the *HASP* algorithm is from the original, correct, partition and it was 0.95 which is a quite high value.

For the second experiment we created three new application classes and used the *AssignClass* algorithm to suggest the suitable package from them. All three classes were placed in the correct package by the *AssignClass* algorithm.

2.1.5 Discussion and comparison to related work

Besides the experimental evaluation presented above we have conducted some additional experiments to investigate the effectiveness of the *HASP* algorithm. Besides the *DbUtils* framework, we have used two other open-source software systems, *Email* [Ema] and *EL* [EL].

During this analysis we have computed the value of different metrics and measures from the literature for both the original package structure of the considered frameworks, and for the package structure suggested by the *HASP* algorithm. There were a total of **29** evaluation measures and in **25** cases the partition provided by the *HASP* algorithm had a better or equal value than the original partition.

Out of the methods reported in the literature, the closest to our method is the approach from [AAM11], which uses hierarchical clustering, just like our method. It presents two approaches, and we applied both of them on the *DbUtils* system. The first approach is based on the number of class initializations, but this might not be a good direction in case of software frameworks, which usually contain many abstract classes and interfaces which are never initialized.For the *DbUtils* framework this approach does not suggest modifications.

The second method presented in [AAM11] uses a vector-space representation based on method-call dependencies. In [AAM11] three linkage metrics are used for clustering and a new algorithm is introduced as well, with a lower complexity. We tried the three clustering algorithms for the vector-space representation created for classes from the *DbUtils* system, using as stopping criteria the point where the distance between all clusters was one. Then we computed the *CIP* measure between the results and the original and the *HASP* partition. It had values between 0.3 and 0.43, suggesting that these partitions are far from both of them.

2.1.6 A comparison of the *overallScore* measure to other measures

In the following we will present a study performed in order to evaluate how well packages in a software system are structured using the evaluation measures introduced previously: *overallScore* and *CIP*. Through the performed experiments we aim at emphasizing that the *overallScore* measure is well-correlated with the *CIP* measure. Thus, instead of *CIP*, which requires the apriori knowledge of a good partition, *overallScore* can be used for evaluating a software package structure.

For the experiments, we have considered the three open-source software systems that were used before: *DbUtils*, *Email* and *EL*. For each of these software frameworks we have considered four different package structures: the original, the one generated by the *HASP* algorithm, and two "wrong partitions" created manually by modifying the original package structure.

During the experimental evaluation we have computed the values of the *overallScore* and *CIP* measures for each partition. We have then computed two rank-based correlation measures between them: the Spearman correlation [Spe04], and Spearman's footrule [DG77]. Both correlation measures suggest that the *overallScore* and the *CIP* measure are well-correlated.

In order to compare our evaluation measure with other measures from the literature, we computed the value of the seven metrics introduced in [DABH11] for the four partitions of the three software systems. Then we computed the Spearman's footrank between these metrics and the *overallScore* measure. A visual comparison of the footrule is presented on Figure 2.1.

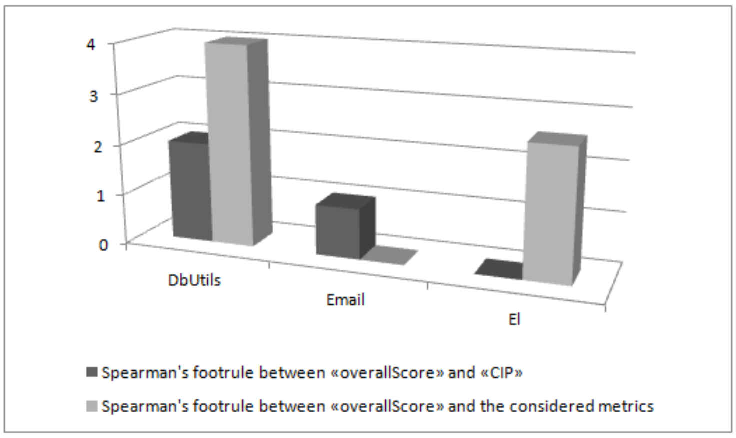

Figure 2.1: Comparison of footrule values.

Considering the comparison of the footrule values presented in Figure 2.1 we can conclude that the *overallScore* measure is more suitable for evaluating a partition of a software system than the other metrics considered. Moreover, *overallScore* can be used instead of *CIP* since the two values are strongly positively correlated.

2.2 Class-Level Remodularization

In this section we will present our original approaches for class-level remodularization of a software system [MCC12, Mar12c, Mar12b, Mar12a]. We will present methods that are capable of automatically identifying an improved internal structure of a software system using different software metric values. This improved internal structure can be achieved by applying different refactorings on the original software system.

2.2.1 Metrics-based Software Restructuring

In the next subsections the following notations will be used: we will consider a software system S to be a set $S = \{s_1, s_2, ..., s_n\}$ where s_i, $1 \leq i \leq n$, is called an *entity* and it can be an application class or a methods from an application class. We wanted to use the values of software metrics for representing these entities, so we needed software metrics that can be computed for both an application class and a method. We have chosen the following five metrics: Relevant Properties (*RP*), Depth in Inheritance Tree (*DIT*) [CK91], Number of Children (*NOC*) [CK91], Fan-In (*FI*) [HK81, Mai09], Fan-Out (*FO*) [HK81, Mai09]. Out of these metrics, *RP* has as value a set, while the other have integer values.

2.2.1.1 Vector Space Model and Distance Function

The main idea of our approach was to characterize each entity from the software system S by a list of values for the above described metrics. Thus, each entity s_i $(1 \leq i \leq n)$ from the software system S will be represented as a 5-dimensional vector, having as components: $(rp(s_i), dit(s_i), noc(s_i), fi(s_i), fo(s_i))$. The values in the vector are scaled to [0,1], where necessary. In order to perform clustering in this vector-space representation of the entities, we defined a distance function, which expressed in Formula 2.2.

$$d(s_i, s_j) = \begin{cases} 0 & \textit{if} \;\; i = j, \\ \sqrt{\frac{1}{5} \cdot \left(1 - \frac{|s_{i1} \cap s_{j1}|}{|s_{i1} \cup s_{j1}|} + \sum\limits_{k=2}^{5} (s_{ik} - s_{jk})^2\right)} & \textit{if} \;\; s_{i1} \cap s_{j1} \neq \emptyset \;\; , \\ \infty & \textit{otherwise} \end{cases} \tag{2.2}$$

2.2.1.2 The *ARI* algorithm

Using the above presented software metrics, the vector space model and the distance function we have defined the *Automatic Refactoring Identification* (*ARI*) algorithm. The main idea is to identify a partition $\mathcal{K} = \{K_1, K_2, ..., K_m\}$ of the software system S, which corresponds to an improved structure of the software system. Each cluster K_i corresponds to an application class in this improved structure.

The *ARI* algorithm starts with an empty partition. Than for each method we search for the class which is closest to it (considering the distance function from Formula 2.2). If there is a class with distance less than ∞ the method is either placed in the cluster where the class appears (if there is such a cluster) or a new cluster is created for them. If the distance to all classes is ∞, we are searching in the clusters without classes for the method closest to this method. If there is no such cluster, or all distances are ∞, we create a new cluster and put the method in it. Otherwise, we add it to the found cluster. Finally, when all methods were added, we compute the distance between all pairs of classes. If there is a pair with distance less than ∞ the corresponding clusters are merged.

2.2.1.3 The *HAC* algorithm

Using the same metrics, vector space representation and distance function, we have also introduced an extension of the *ARI* algorithm, named *Hierarchical Agglomerative Clustering* (*HAC*) algorithm. It is based on *hierarchical agglomerative clustering* and uses a heuristic function which expresses that two clusters are merged only if their distance is less than 1. The reason for this function is that distances greater than 1 are obtained for entities that are not cohesive.

2.2.1.4 Identified Refactorings

Both the ARI and the HAC algorithms create a partition of the software system S, which corresponds to a good internal structure. Comparing this partition to the original structure of the software system, a set of refactorings can be identified, which would improve the structure of the software system. For both methods these refactorings are: *Move Method*, *Inline Class* and *Extract Class.*

2.2.2 Aggregated Metrics-based Software Restructuring

This subsection presents our third algorithm for software restructuring, which, like the previous two algorithms, uses software metrics, but the values of these metrics are aggregated into a single value.

For this approach we needed metrics whose value can be computed for an application class. Also, since the aggregated value of these metrics will be used to characterize the quality of an application class (and the quality of a software system through its application classes) our intention was to choose metrics that measure the size, coupling and cohesion in a software system. We have studied the list of software metrics implemented by different plugins and tools and have selected the following 16 metrics: Afferent Coupling (CA) [BJWD99], Coupling Between Objects (CBO) [CK94], Data Abstraction Coupling (DAC) [LH93], Information Flow Based Cohesion (ICH) [LLWW95], Instability (INS), Tight Class Coupling (TCC) [BK95], Loose Class Coupling (LCC) [BK95], Lack of Cohesion in Methods 1 (LCOM1) [CK94], Lack of Cohesion in Methods 2 (LCOM2) [HS96], Lack of Cohesion in Methods 4 (LCOM4) [HM95], Lack of Cohesion in Methods 5 (LCOM5), Locality of Data (LD) [HM95], Message Passing Coupling (MPC) [LH93], Number of Attributes (NOA) [LH93], Number of Methods (NOM) [LH93], Response for a Class (RFC) [CK94].

2.2.2.1 Case Study

Since good values for these software metrics are not known, we have performed a study, to analyze how the value of these metrics changes as we improve the software system. We have taken four different case studies, each of them being a small example with 2 or 3 classes, and in each case there was at least one method placed in a different class, than it should be.

We have computed the value of each metric presented in the previous subsection for the above presented examples. Then, we have "corrected" the examples, by moving the methods to the classes where they should be, and computed the value of the metrics again. We were interested in how the value of the metrics changes for the two versions. The result of this case study is presented on Table 2.1, where the label *Same* means the value did not change, *Up* means that the value of the metric increased for the correct version of the example, and *Down* means that the value of the metric decreased for the correct version.

The most important thing to notice when analyzing Table 2.1 is that for each metric the direction of change is consistent: for each example it either did not change or if it changed it always did in the same direction. This is important, because it allows the comparison of two versions of the same system based on the change in the values.

2.2.2.2 Aggregated Metrics-based Software Restructuring algorithm

The main idea of our algorithm is to characterize each class from a software system by a single aggregated value of these software metrics. In order to do so, we first had to normalize the value of these metrics. We performed the normalization in such a way that lower values of the metrics correspond to a better design.

After normalizing the metric values, we introduced a metric, $M(C)$, defined for an application class C as the sum of the normalized value for the above presented 16 software metrics.

Metric	Ex 1	Ex 2	Ex 3	Ex 4	Metric	Ex 1	Ex 2	Ex 3	Ex4
NOA	Same	Same	Same	Same	ICH	Up	Same	Same	Same
NOM	Up	Same	Same	Up	CA	Same	Same	Same	Same
CBO	Same	Same	Same	Same	INS	Same	Same	Same	Same
DAC	Same	Same	Same	Same	LCOM1	Same	Down	Down	Down
TCC	Same	Up	Up	Up	LCOM2	Same	Down	Down	Down
LCC	Same	Up	Up	Up	LCOM4	Same	Same	Down	Down
MPC	Down	Same	Down	Down	LCOM5	Same	Up	Same	Up
RFC	Up	Same	Up	Same	LD	Same	Same	Up	Same

Table 2.1: The direction of change for metric values for the original and correct version of the examples.

Since the metric values are normalized in such a way that small values correspond to a better design, the smaller the value of the $M(C)$ metric for a class, the better its internal structure is. Since a software system S is a set of n application classes, we have defined an aggregated metric $AM(S)$, as the average of the values of the $M(C)$ metric computed for all classes of the software system.

Using the *AM(S)* metric and hierarchical agglomerative clustering, our approach will first place every method from the software into its own cluster (representing an application class with only one method), and will keep merging clusters that lead to the partition with the lowest value for *AM(S)* until all methods will be in the same application class. Out of the generated partitions, the one with the lowest *AM(S)* will be reported as solution.

2.2.3 Experimental Evaluation

***ARI* algorithm.** In order to evaluate the performance of the *ARI* algorithm, we have used two different case studies. The first case study is a simple example with two classes, where one class contains a method, which should be placed into the other class. For this example the *ARI* algorithm identifies that the method should be placed into the other class.

The second case study that we used to evaluate the *ARI* algorithm is the open-source software *JHotDraw*, version 5.1 [Gam]. We chose *JHotDraw* because it is a well-known example for the use of design patterns and for good design and because it is a complex project, consisting of 173 classes, 1375 methods and 475 attributes.

After applying the *ARI* algorithm for the *JHotDraw* case study, 20 *Move Method* refactorings were identified. After an analysis of the results, we concluded that 11 of them are justifiable, while 9 are misplaced.

***HAC* algorithm.** We have used the *JHotDraw* project as case study for the *HAC* algorithm, too. The result returned by the algorithm contained only 3 clusters (i.e., application classes) that did not correspond to the existing structure of the framework. These three clusters represent possible *Extract Class* refactorings, meaning that new classes should be created, and the selected methods should be moved to the classes. After an analysis we concluded that out of three suggested *Extract Class* refactorings one is justifiable, and the other two are partially justifiable, meaning that some of the suggested methods should indeed be moved to a new class.

Aggregated metrics-based method. The evaluation of the aggregated metrics based method was, so far, performed only for small examples (namely, the four case studies presented in Subsection 2.2.2). The main reason for this is that some steps of the algorithm have to be done manually. This is caused by the fact, that, in order for the results of our approach to be valid, at each step of the algorithm, we have to keep the behavior of the system unchanged and we did not find yet a good way of automating this step.

We applied the algorithm for three case studies from Subsection 2.2.2 and in all three cases it identified the partition corresponding to the correct design of the software systems.

2.2.3.1 Comparative analysis

Comparative analysis of the three algorithms. The *ARI* and *HAC* algorithms are quite similar, they use the same vector space model to represent entities from a software system, and they use the same software metrics as well. Still, the *HAC* algorithm uses an unsupervised learning method, hierarchical agglomerative clustering.

Both algorithms identified most of the *JHotDraw* system as correctly designed, as we have expected, since *JHotDraw* is considered an example of good design. Out of the reported refactorings, the *HAC* algorithm had only 5 misplaced methods, while the *ARI* algorithm had 9. Also, the *HAC* algorithm identified 3 *Extract Class* refactorings, which were not identified by the *ARI* algorithm, thus showing that using unsupervised learning for automatic restructuring of a software system is beneficial, since they can find hidden patterns in data.

In case of the aggregated metrics-based software restructuring algorithm, there is one common case study with the *ARI*, the small artificial example which was the first case study for the former, and the second example for the latter. For this example both algorithms found the correct restructuring.

Comparison to related work. Seng et al. apply in [SSB06] a weighted multi-objective search, in which metrics are combined into a single objective function. This approach partially gives the results obtained on *JHotDraw*, but the *ARI* algorithm has less misplaced methods, it is deterministic and has lower running time.

In [CS06] an approach is introduced that uses clustering for improving the class structure of a software system and a partitional clustering algorithm, *kRED* (*k-means for REfactorings Determination*), is presented. Unlike the *kRED* algorithm, our algorithm also identifies the *Extract Class* refactoring. Compared to [RR11] and [FTCS09], the *HAC* and *ARI* algorithms are capable of identifying three types of refactorings (not just the *Extract Class*).

The closest to these two methods is the *HARS* algorithm, in [CS07], which uses only the set of relevant properties for computing the distance between two entities, and considers attributes as entities, too. [CS07] uses *JHotDraw* as a case study too, and it finds only one out of the three *Extract Class* refactorings that *HAC* finds (but it finds also two *Move Attribute* refactorings, which neither *ARI* nor *HAC* can find).

2.3 Conclusions and Further work

This chapter presented different methods for automatically restructuring a software system to improve its quality. We have started with the presentation of the *CASP* approach and the *HASP* algorithm, which is a package-level restructuring method. We have experimentally evaluated our approach using two open-source case studies, and the results show that the *HASP* algorithm is capable of finding a good package structure.

We have also described three different algorithms which use the values of software metrics and hierarchical clustering, for class-level restructuring of a software system. The experimental evaluation of the presented methods shows that these methods can indeed identify a partition of a software system that corresponds to a better design.

As further work we would like to extend the experimental evaluation of all approaches for other open-source software systems. We would also like to address the shortcomings of our approaches to improve them.

Chapter 3

New Approaches to Software Defect Detection

In this chapter, which is entirely original, we are approaching the problem of software defect detection, an important problem in software engineering, since the increasing size and complexity of software systems makes finding defects manually harder and harder. Consequently, different intelligent approaches, which help software developers to identify which parts of a software system are defective and need attention, are welcome. Machine learning-based methods are presented in the literature for both directions, *design defect detection* and *defect prediction*. In this chapter we will present a novel approach, namely how relational association rules can be used both for design defect detection and defect prediction.

The two approaches presented in this chapter are original works published in [CMC14a], [Mar13b], [CMC14b] and [Mar13a].

Motivation. Each software system goes through many changes during its lifetime, otherwise it becomes obsolete and will no longer be used. These changes, mostly done during maintenance, can be performed to add new functionalities, to correct existing defects or to adapt to changing environments in which the system is used. These modifications can degrade the structure of the system, which, in turn, will make the next modifications even more complicated. In order to avoid this vicious circle, the identification of defective software entities is of major importance.

A software system may be represented as a dataset where the elements are the application classes, using a high-dimensional representation based on software metrics. From this dataset significant information can be extracted from the software metrics values characterizing the application classes. Different types of relationships between the numerical feature values can be defined and a relational association rule mining process can be performed on the dataset representing the software system. Such a mining process can provide interesting patterns, such as patterns indicating classes with a defective design, or it can provide rules which describe defective and non-defective entities from the software system.

3.1 Theoretical model

In this section we will present the theoretical model that is used in the approaches for both *design defect detection* and *defect prediction*.

The main idea of this approach is to represent the entities (classes, modules, methods, functions) of a software system as a multidimensional vector, whose elements are the values of software metrics applied to the given component. We consider that a software system S is a set of components (called *entities*) $\mathcal{S} = \{s_1, s_2, ..., s_n\}$. As we aim at identifying ill-structured software entities, we consider a set of software metrics relevant for characterizing the internal structure of a software entity. Consequently, we have a feature set of software metrics $\mathcal{SM} = \{sm_1, sm_2, ..., sm_k\}$ and thus each entity $s_i \in \mathcal{S}$ from the software system can be represented as a k-dimensional vector, having as components the values of the software metrics from $\mathcal{SM}$.

3.2 Design Defect Detection

This section presents our approach for detecting design defects in a software system, called *SDDRAR* (Software Design Defect detection using Relational Association Rules) [CMC14a].

3.2.1 *DRAR* algorithm

As presented in Subsection 1.4.1 relational association rules represent an extension of ordinal association rules, but they allow more general relations to be defined between the attributes, not just ordinal ones. In [CSTM06] an Apriori-like algorithm, called *DOAR* (Discovery of Ordinal Association Rules), was presented, an algorithm which can find efficiently all ordinal association rules of any length that hold over a dataset.

The *DOAR* algorithm was extended in [CMC14a] towards the *DRAR* algorithm (Discovery of Relational Association Rules) for finding interesting relational association rules of any length.

3.2.2 The *SDDRAR* approach

A set of well-designed software systems, $\mathcal{S}_{good}$, is considered. Using the theoretical model described in Section 3.1, let us consider that $\mathcal{DS} = \{ds_1, ds_2, \ldots, ds_n\}$ is the dataset of k-dimensional software entities extracted from the software systems from $\mathcal{S}_{good}$. The feature set characterizing the dataset consists of a set of software metrics $\mathcal{SM} = \{sm_1, sm_2, ..., sm_k\}$ identified to be relevant in the mining process.

For detecting ill-structured application classes using relational association rule mining, the following steps will be performed:

1. Data collection and preprocessing.
2. Building the *SDDRAR* model.
3. Testing.

3.2.2.1 Data collection and preprocessing

During this step, a statistical analysis is carried out on the (training) dataset $\mathcal{DS}$ in order to find a subset of features that are relevant for the considered task. To determine the dependencies between features, the Pearson correlation coefficient is used [Tuf11]. For each software metrics sm, we compute its *absolute average correlation* (denoted by $avg(sm)$) as the average of the correlation of the metric to the other software metrics. We compute the mean (denoted by m) and standard deviation (denoted by $stdev$) of these values, and remove those metrics for which $|avg(sm) - m| > stdev$.

3.2.2.2 Building the *SDDRAR* model

The first step of building the *SDDRAR* model is to define the possible relations between the software metrics, which will be needed for the relational association rule mining process. Then the interesting relational association rules of any length, having a minimum support s_{min} and a minimum confidence c_{min} are discovered in the dataset DS. The set of these rules is denoted by $\mathcal{RAR}$. These rules will be used to identify badly designed software entities.

3.2.2.3 Testing

At the detection stage, a new software system, $\mathcal{S}_{new}$, has to be analyzed in order to detect software entities that have a bad design. First, $\mathcal{S}_{new}$ is analyzed and the k-dimensional representation of the software entities from $\mathcal{S}_{new}$ is computed. Then the following three steps are executed:

1. **Errors computation step.** For each entity e_i from the analyzed software system the *number of errors*, $err(e_i)$, is computed as the number of relational association rules from $\mathcal{RAR}$ that are not verified in the k-dimensional representation. Then, the *percentage of errors*, pe_{e_i}, is also computed as the number of errors divided by the number of rules.

2. **Detection step.** After determining the *number of errors* corresponding to each software entity, we report as possibly badly designed entities those that do not verify a large enough number of rules from $\mathcal{RAR}$.

3. **Defect analysis.** For the ill-designed software entities reported at the previous step, we compute for each software metrics the *number of errors*, as the number of binary relational association rules from $\mathcal{RAR}$ that are not verified by the entity. These numbers of errors are further analyzed to determine the cause of the defect.

3.2.3 Experimental evaluation

For the experimental evaluation of the *SDDRAR* approach, we have chosen to consider as entities the application classes from the software systems. As indicated in Section 3.1, using the vector space model approach, a set of software metrics relevant for characterizing the design of application classes has to be selected. We have chosen the same 16 software metrics that were used for our aggregated metrics-based restructuring approach (Section 2.2.2).

3.2.3.1 Training data

As presented above, for the training part of our approach a set of well-designed software systems $\mathcal{S}_{good}$ has to be considered in order to collect from it the k- dimensional representation of the application classes. For the current implementation we have selected a single well-designed software system, the open-source case study *JHotDraw* version 5.1 [Gam].

For the first steps of the *SDDRAR* approach we have first analyzed the *JHotDraw* software system and extracted from it the dataset $\mathcal{DS}$ consisting of the 16-dimensional representation of the application classes. We have scaled the value of the software metrics to the interval [0,1] and have performed the statistical analysis, which led to the elimination of four software metrics: *CA*, *NOM*, *RFC* and *TCC*.

The second step of the *SDDRAR* approach consists of the mining of the interesting relational association rules of any length, having a minimum support $s_{min} = 0.9$ and a minimum confidence $c_{min} = 0.85$ from the dataset $\mathcal{DS}$.

3.2.3.2 Case studies

The third step of the *SDDRAR* approach is *Testing*, where the *SDDRAR* model, built at the previous steps, is used in order to detect ill-designed entities in a software system. For this step we considered six case studies, the first two were simple, artificial examples, while the other four were open-source softwares taken from the *SourceForge* repository. The results for the six case studies are presented below:

- First simple case study taken from [SSL01] - the *SDDRAR* method identified the application class having a design defect.

- Second simple case study taken from [Fow99] pages 22-26 - the *SDDRAR* method identified the application class having a design defect.
- *FTP4J* open-source software [FTP13] versions 1.5, 1.5.1, 1.6 and 1.6.1 - for all versions the *FTPClient* class was identified as having design defects. Our analysis showed that it is a *God Class*, so it was correctly identified by the *SDDRAR* method.
- *ISO8583* open-source software [ISO13] versions 1.5.2, 1.5.3 and 1.5.4 - for all versions the *MessageFactory* class was identified as having design defects. Our analysis showed that it is a *God Class*, so it was correctly identified by the *SDDRAR* method.
- *Profiler4J-Agent* open-source software [Pro13] versions 1.0-alpha5, 1.0-alpha6, 1.0-alpha7 and 1.0-beta1 - for version 1.0-alpha5 the *Server* class is reported; for version 1.0-alpha6 the *MemoryInfo* class is reported; for the last two versions the *Config* class is reported. Our analysis showed that the *Server* class is too coupled to the rest of the classes, and *MemoryInfo* and *Config* are *Data Classes*, so they were correctly identified by the *SDDRAR* method.
- *WinRun4J* open-source software [Win13] versions 0.4.0, 0.4.1, 0.4.2, 0.4.3 and 0.4.4. - for all versions the *NativeBinder* class was identified as having design defects. Our analysis showed that this class has some problems and could be improved, but it does not have one serious design defect, so we considered it a borderline decision.

3.2.4 Discussion and comparison to related work

In this subsection we will present an analysis the *SDDRAR* approach from multiple viewpoints and a comparison with similar approaches from the software engineering literature.

Detection precision. We have considered that for the first five case studies we have a precision of 1 (the identified classes had design defects) and for the last case study we considered a precision of 0.5, since it was a borderline decision. Thus, the average *detection precision* was 0.917 on the six considered case studies.

Average error measure. Besides determining the potentially badly designed software entities, the *SDDRAR* method was applied for conducting a study as follows. For each version of each open-source software system we have computed the average number of errors of the application classes. This value decreases (or remains the same) for each new version of the systems. This confirms that smaller values for the error indicate better designs, since we expect that higher number of versions of a software system have better design.

Robustness. As the process of detecting ill-structured entities that we propose is dependent on the initial entities from $\mathcal{DS}$ that are assumed to be well-designed, we studied what happens if we introduce noise in this dataset. We have introduced some ill-designed entities in the dataset $\mathcal{DS}$ and applied the *SDDRAR* method to the above presented case studies. As we have expected, the *SDDRAR* approach is robust, the presence of a small number of ill-designed entities did not influence the results.

Set of relations. The last perspective is the set of relations $\mathcal{R}$ considered in the process of relational association rule discovery. We have experimented with different combinations of the possible relations and concluded that these combinations are also capable of finding design defects in the datasets.

3.2.4.1 Comparison to related work

In order to compare our approach to the approaches presented in the literature, we have chosen two openly available tools: the *JDeodorant* Eclipse plugin [JDe13] and the *iPlasma* tool, presented in [Mar02] and available at [IPl13]. The classes reported as having design defects by the *JDeodorant* and *iPlasma* tools and our method for the first version of three

	Ftp4J	ISO8583	WinRun4J
SDDRAR	FtpClient	MessageFactory	NativeBinder
JDeodorant	FtpClient (36) NVTASCIIWriter (0) NVTASCIIReader (0) FTPProxyConnector (2) FTPCommunication - - Channel (0)	ISOMessage (28) MessageFactory (38)	Launcher (28) FileAssociation (23) RegistryKey (0)
iPlasma	FTPClient - Brain Class (36) FTPFile - Data Class (19)	MessageFactory - God Class (38)	Closure - God Class (27) FileVerb - Data Class (26) NativeBinder - God Class (31)

Table 3.1: Classes with design defects reported by the JDeodorant, iPlasma tools and *SDDRAR* method.

open-source systems are presented in Table 3.1. For *JDeodorant*, we only considered the *God Class* defects. For *iPlasma* we specified after each class the defect that was reported. Also, after each class, between parenthesis, we added the number of errors that the *SDDRAR* method reports for the given class.

From Table 3.1, we can see that not even the two tools agree on which classes have defects and which do not. Like most other approaches, we decided to manually verify the reported classes, to decide which tool's results to use for comparison. In case of the *iPlasma* project, we agree with the reported classes. In case of the *JDeodorant* tools, it seems like it rather tries to find *Extract Class* opportunities, instead of finding real *God Classes*.

Comparing our results to the ones reported by the *iPlasma* tool, we can observe that the application classes reported by the *SDDRAR* approach are reported by *iPlasma* as well, and for the other classes reported by *iPlasma* our approach reports a high number of errors as well (even if the number is not high enough to report the classes as defective). Using the results reported by *iPlasma* we computed the recall of the *SDDRAR* approach, which is 0.85.

3.2.5 Study on parameter variations

The *SDDRAR* method depends on a couple of different parameters, whose values can influence the results, so we have performed a study to investigate the following aspects:

- Using the original or normalized values for software metrics.
- Using binary or any length relational association rules.
- Using only maximal relational association rules, or using all mined rules.
- The effect of modifications for the τ parameter and the minimum confidence on the detection accuracy.

As a result of this study we have concluded that it is better to use normalized software metric values, to mine relational association rules of any length, and to use the maximal association rules only. For the last aspect, if τ was lowered, more application classes were reported as defective, and most of these classes were defective to a given extent. Lowering the value of the minimum confidence led to more application classes reported as well, and many of them were *Data Classes*.

3.3 Defect Prediction

This section presents our supervised learning approach for predicting defects in a software system, called *DPRAR - Defect Prediction using Relational Association Rules* [CMC14b]. For this approach we will use the theoretical model presented in Section 3.1 and the $DRAR$ algorithm for mining the relational association rules.

In our approach, the problem of *defect prediction* is considered a supervised binary classification problem, since each entity should be classified as either *defective* also noted with "+" or *non-defective* noted with "-". And, like many other approaches, our model is built based on training data, i.e., a dataset containing data about entities for which we know whether they are defective or not, thus we have a supervised classification problem.

3.3.1 The $DPRAR$ classifier

Since *defect prediction* is a supervised classification problem, we have a training and a testing step. For the training we consider two datasets: DS_+ consisting of the defective k-dimensional entities and DS_- consisting of the non-defective k-dimensional entities. For classifying a a software entity as being or not defective, the following steps will be performed:

1. Data preprocessing.
2. Training/building the $DPRAR$ classifier.
3. Testing/classification.

3.3.1.1 Data preprocessing

During this step, the training data are scaled to [0,1] and a statistical analysis is carried out on the training datasets DS_+ and DS_- in order to find a subset of features that are correlated with the target output. To determine the dependencies between features and the target output, the Spearman's rank correlation coefficient [Spe04] is used.

In order to decide which feature(s) to remove for each feature (software metric) $sm_i \in \mathcal{SM}$ we compute the Spearman correlation ($cor(sm_i, target)$) between the feature and the target output (defect or non-defect). Let us denote by m the average value and $stdev$ the standard deviation of the correlations between all features and the target output. A feature sm_i is removed from the feature set if the absolute value of its correlation is less than $m - stdev$, i.e., $abs(cor(sm_i, target)) < m - stdev$.

3.3.1.2 Training/building the $DPRAR$ classifier

First, we define a set of relations between the feature values that will be used in the relational association rule mining process. Then we perform the relational association rule mining on the two training datasets separately, resulting in two sets of relational association rules: RAR_+ and RAR_-. For each rule r from these sets we associate a value, called $ratio(r)$, computed by dividing its confidence to its support.

3.3.1.3 Testing/classification

At the classification stage, after the training was completed and the $DPRAR$ classifier was built, when a new software entity e has to be classified, we calculate two scores $score_+(e)$ (the similarity of e to the *positive* class) and $score_-(e)$ (the similarity of e to the *negative* class). For computing $score_+(e)$ we consider the average ratio of rules from RAR_+ which are verified by the entity e and the average ratio of rules from RAR_- which are not verified by e. $Score_-(e)$ is computed similarly.

Name	**Description**	**Defective entities**	**Non-defective entities**
CM1	NASA spacecraft instrument written in C	42 (12.84 %)	285 (87.16%)
KC1	Storage management for receiving and processing ground data	314 (26.54%)	869 (73.46%)
KC3	Processing and delivery of satellite metadata	36 (18.56 %)	158 (81.44 %)
PC1	Built for functions from a flight software for earth orbiting satellite	61 (8.65 %)	644 (91.35%)
JM1	Real-time predictive ground system	1672 (21.49 %)	6110 (78.51 %)
MC2	Video guidance system	44 (35.2 %)	81 (64.8%)
MW1	Zero gravity experiment related to combustion	27 (10.67%)	226 (89.33 %)
PC2	Dynamic simulator for attitude control systems	16 (2%)	729 (98%)
PC3	Flight software for earth orbiting satellite	134 (12.4%)	943 (87.6 %)
PC4	Flight software for earth orbiting satellite	177 (13.8%)	1110 (86.2 %)

Table 3.2: The *NASA* datasets used for the experiments.

Case study	c^{+}_{min}	c^{-}_{min}	**Length**	**Acc**	**Pd**	**Spec**	**Prec**	**AUC**
CM1	0.927	0.94	any	0.8716	0.929	0.8632	0.5	0.896
KC1	0.8	0.822	2	0.823	0.818	0.825	0.628	0.822
KC3	0.885	0.96	2	0.83	0.889	0.8165	0.5246	0.85225
PC1	0.95	0.995	2	0.956	0.885	0.963	0.692	0.924
JM1	0.95	0.995	any	0.96	0.842	0.992	0.967	0.917
MC2	0.96	0.99	any	0.896	0.773	0.9632	0.919	0.868
MW1	0.97	0.975	any	0.941	0.889	0.947	0.667	0.918
PC2	0.95	0.995	any	0.984	0.938	0.985	0.577	0.962
PC3	0.95	0.995	2	0.967	0.85	0.983	0.877	0.917
PC4	0.95	0.995	2	0.961	0.814	0.985	0.894	0.899

Table 3.3: Obtained results for all datasets for the *DPRAR* classifier.

At the classification stage of a new instance e if $score_{+} > score_{-}$ e will be classified as a *positive* instance (defect), otherwise it will classified as a *negative* instance (non defect).

3.3.2 Experimental evaluation

For the experimental evaluation of our approach we have used ten *NASA* datasets [Nas]. The names and characteristics of these datasets are presented in Table 3.2.

For evaluating the performance of the *DPRAR* classifier, a cross-validation using a "leave-one-out" methodology was applied and the following performance measures were used: *accuracy*, *probability of detection*, *specificity*, *precision* and *AUC*. Table 3.3 presents the best results obtained by the *DPRAR* classifier for all datasets considered for evaluation.

3.3.3 Discussion and comparison to related work

We have compared the results of the *DPRAR* method to the results reported in the literature for other approaches, for which the experimental evaluation was performed on the same datasets. We have chosen the *CBA*2 method [BDVB11], the 1*R* classifier [CBYP05], the *Bagging* classifier [HDF12] and *EDER-SD* [RRRAR12].

Not all approaches were tested on all 10 datasets that we used, and not all approaches report all the performance measures that we used. Taking into account all evaluation measures for all considered case studies, *DPRAR* performed better in 45 measures, similarly in

1, and worse in 23 out of the 69 evaluation measures. Moreover, the AUC measure reported by the $DPRAR$ classifier (considered in the literature one of the best evaluation measures for classifiers) outperforms the average AUC value reported by existing defect detectors on all considered case studies. This indicates a very good efficiency of the $DPRAR$ classifier.

3.3.4 A study on *DPRAR* classifier

We have performed a study on the *DPRAR* classifier to analyze the influence of feature elimination on its results, and to analyze the effects of using an alternative formula for computing the scores on which the classification of a new entity is based.

The first step of the *DPRAR* classifier is data preprocessing, when different software metrics can be eliminated. For the *DPRAR* classifier we have chosen to eliminate those features whose Spearman correlation to the target value is less than the difference between the average correlations, m, and the standard deviation of the correlations, *stdev*. In this study we compared this to three other approaches (no feature elimination, eliminate features whose correlation is greater than $m + stdev$, eliminate feature whose correlation is less than $m - stdev$ or greater than $m + stdev$). For the experiments we have used the PC3 dataset.

The used performance measures suggested that the best option is to eliminate those features whose correlation is less than $m - stdev$ or greater than $m + stdev$, while the worst option is to use no feature elimination at all.

We have performed another study, this one on the score computation of the $DPRAR$ method, considering as case study the *PC3* dataset, and using the feature elimination technique that provided the best results in the previous study. Instead of using the *ratio* of rules to compute the scores on which the classification is based, we used the number of rules. The performance measures suggested that this new score computation method is better.

We have also compared the results of the two score computation techniques to four methods presented in the literature: *CBA2*, *ROCUS*, Random Forests with one - against - one coding and *Dynamic AdaBoost.NC*. The *DPRAR* classifier with score computation based on the number of rules had the best results, followed by the *DPRAR* classifier with score computation based on the *ratio* of rules.

3.4 Conclusions and Further work

This chapter presented our relational association rule mining-based approaches for software defect detection. Software defect detection has two main directions, *software design defect detection* and *software defect prediction*, and we have worked on both directions.

In the first part of the chapter we described the *SDDRAR* approach, which is capable of identifying entities with design defects in a software system. The experimental evaluation of the approach was performed using 6 case studies and the manual analysis of the results showed that the *SDDRAR* approach is capable of identifying entities with design defects in a software system. We have compared our approach to two other approaches from the literature *JDeodorant* and *iPlasma*, and the results were similar to those reported by the *iPlasma* tool. These results show the potential of our proposed approach.

In the second part of the chapter we presented the *DPRAR* approach, a binary classification model based on relational association rules for defect prediction. For the experimental evaluation of this model, 10 NASA datasets were used. Comparing the results of our model with results reported in the literature, shows that the *DPRAR* approach is better than, or comparable to, the classifiers already applied for software defect detection.

As further work we would like to extend the experimental evaluation of these approaches to other software systems, datasets and software metrics. We would also like to extend our approaches to use fuzzy relational association rules.

Chapter 4

A Software Framework for Analyzing Object-Oriented Software Systems

We have presented in the previous chapters within the search-based software engineering domain several computational intelligence-based techniques for solving problems of great importance during software maintenance and evolution. For the development of most of these techniques, we have used the *FAOS* framework (*Framework for Analyzing Object-oriented Software systems*), which we have designed to be generic enough to offer a support for analyzing an object-oriented software system and to easily extract from it relevant information, as well as to provide reference implementation for several software metrics which are useful to measure the software quality. In this chapter we will present this framework in more detail.

4.1 The FAOS framework

The *FAOS* framework is currently composed of three main modules, *Analyzer*, *Metrics* and *PackageRestructuring.*

The *Analyzer* module. The first module of the *FAOS* framework is the *Analyzer*, whose current implementation can analyze software systems written in Java, and it requires either the compiled *.class* files or a *jar* archive for the analysis. We have defined a datamodel which consists of classes to represent an application class, a method, a field or a package from the analyzed software system. The main role of the *Analyzer* module is to perform the analysis of a software system and extract a list of the application classes together with their methods and fields and the relations between them. For the actual analysis of the compiled Java code we used the *ASM* bytecode manipulation framework [ASM13].

The *Metrics* module. The second module of the *FAOS* framework is the *Metrics* module, which contains the implementation of different software metrics. This module is divided into two main packages, the *classLevel* package contains the implementation of 17 software metrics computed for a class, while the *packageLevel* package contains the implementation of 20 software metrics and measures computed for a software package.

The *PackageRestructuring* module. The third module of the *FAOS* framework is called *packageRestructuring* and the package-level restructuring approach presented in Section 2.1 is implemented in it, so, it is a little less abstract that the previous two modules.

4.2 Comparison to similar frameworks

In computational intelligence-based software engineering researchers often implement their approaches in the form of different tools or frameworks that can be used both by researchers for the comparison of results, but also by software developers to get help with their everyday tasks. While there is no other framework with the exact same functionalities as *FAOS*, there are some frameworks in the literature which are similar to parts of it, like *JDeodorant* - an Eclipse plug-in for identifying bad smells in a software system -, *iPlasma* - an environment designed for the quality analysis of object-oriented software systems - and the *Bunch* tool - which tries to find a good partition of a software system using search algorithms. The first two have their own internal representation (similar to our datamodel), while the *Bunch* tool needs only the Module Dependency Graph of the system to be restructured. Thus, *Bunch* is programming language independent, *JDeodorant* works for systems written in Java and *iPlasma* can analyze both Java and C++ code. Their advantage over the *FAOS* framework is that they are complete tools, ready to be used. On the other hand, the *FAOS* framework is more flexible, it can be easily extended to perform other kinds of analyses, not just the currently implemented ones.

In conclusion, *JDeodorant* and *iPlasma* are suitable for situations when somebody wants just to analyze the source code, look for abnormal metric values or bad smells. The *FAOS* framework is suitable when somebody wants to analyze a system, and perform some tasks on the resulting list of entities. Thus the main advantage of *FAOS* can be considered that it is easily extendable: new software metrics can easily be added to it, and the list of entities returned by the *Analyzer* module can be used for further analysis. Such a case can be observed in the *PackageResturcturing* module, where this list is used to find a suitable division into packages of the classes.

4.3 Conclusions and Further work

In this chapter we have presented the *FAOS* framework, a framework for analyzing object-oriented software systems. The *FAOS* framework was developed to support most of the machine learning-based approaches that were presented in the previous chapters. Thus, the *Analyzer* module was used for all approaches, with the exception of defect prediction, where the NASA datasets were used as input. The *Metrics* module was used both for class-level and package-level restructuring of a software system, but it was used in our design defect detection approach as well. Finally, the *PackageRestructuring* module is the implementation of our approach for package-level restructuring of a software system.

As future work, we would like to make the *FAOS* framework openly available, so that other people can use and extend it according to their needs. In order to make working with it easier, we are thinking about developing an Eclipse plugin based on *FAOS*. We would also like to extend the analysis part to other programming languages as well, because once the internal representation of the entities is built, the other parts of the framework can be used.

Conclusions

In this thesis we have presented our original work for the application of machine learning methods and algorithms for solving different software engineering tasks. Such intelligent approaches are needed because software systems in our days are quite complex, composed of many elements with many relations between them and approaches that suggest which parts of such systems need attention can be of great help for software developers. Therefore, we can say that such approaches can be considered important tools that are welcome by software developers.

We have chosen two software engineering tasks that represented the main research directions of this thesis and from each we have tackled two problems. The first main research direction is the application of clustering algorithms for software remodularization both on package- and class-level. The second main research direction was the application of relational association rule mining for the detection of defects in software systems. From this research direction we have worked on two problems: design defects detection and software defect prediction. Finally, we presented our contribution towards the development of software systems, the *FAOS* framework, developed for the analysis of object-oriented software systems.

The experimental evaluation of the approaches developed for the first primary research direction show that clustering approaches, in combination with a good measure of the similarity/dissimilarity between the entities, can indeed identify a good partitioning of application classes into packages, or a good partitioning of methods and fields into classes.

In the second main research direction we have used relational association rules for defect detection in a software system. The first problem from this direction was the identification of classes with design defects. The performed experiments showed that the classes identified by our approach did have different design defects. We have compared our results to the results of existing tools and the comparison showed that application classes considered to have design problems by our approach are identified by the *iPlasma* tool [IPl13] as well. Regarding defect prediction, a classification task, our approach was tested on the NASA datasets [Nas]. We have performed a comparison to other approaches over many datasets and evaluation measures, and it showed that the performance of our classifier is better than or comparable with the existing approaches, thus demonstrating the potential of our approach.

The *FAOS* framework presented in this thesis for the analysis of object-oriented software systems was primarily developed to support the experimental evaluation of our proposed approaches. Nevertheless, it was designed to be general and easily extensible so that others can use it as well. The framework offers support for the analysis of a software system, through which an internal representation of the analyzed system is built. This can later be used for different tasks, the tasks being currently implemented allow the computation of different software metrics and the package-level restructuring of the software system.

Regarding future research directions, we intend to address the drawbacks of our approaches in order to improve them. We are also planning on extending the experimental evaluation of these approaches, to use other software systems or datasets. We will also consider the application of the fuzzy version of the proposed approaches, where possible. Finally, we want to develop new approaches for other software engineering tasks, either by using the same machine learning approaches presented in this thesis or by developing new ones.

Bibliography

[AAH12] H. H. Ammar, W. Abdelmoez, and M. S. Hamdi. Software engineering using artificial intelligence techniques: Current state and open problems. In *Proceedings of First Taibah University International Conference on Computing and Information Technology*, 2012.

[AAM11] A. Alkhalid, M. Alshayeb, and S.A. Mahmoud. Software refactoring at the package level using clustering techniques. *IET Software*, 5(3):274–286, 2011.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[AL99] Nicolas Anquetil and Timothy Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of 6th Working Conference on Reverse Engineering*, pages 235–255, Atlanta, USA, October 1999.

[AS94] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. In *Proceeding of the 20th VLDB Conference*, pages 487–499, 1994.

[ASM13] ObjectWeb: Open Source Middleware, 2013. http://asm.objectweb.org/.

[BDVB11] Ma Baojun, Karel Dejaeger, Jan Vanthienen, and Bart Baesens. Software defect prediction based on association rule classification. Technical report, Katholieke Universiteit Leuven, February 2011.

[BJWD99] L.C. Briand and and al. J. W. Daly. A unified framework for coupling measurement in object-oriented systems. 25(1):91–121, 1999.

[BK95] J. M. Bieman and B. K. Kang. Cohesion and reuse in an object-oriented system. *ACM SIGSOFT Software Engineering Notes*, 20(SI):259–262, 1995.

[BLMO10] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Software remodularization based on structural and semantic metrics. In *17th Working Conference on Reverse Engineering*, pages 195–204, 2010.

[BMW02] Lionel C. Briand, Walcelio L. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. Softw. Eng.*, 28(7):706–720, July 2002.

[BOL$^+$10] Gabriele Bavota, Rocco Oliveta, Andrea De Lucia, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Playing with refactoring: Identifying extract class opportunities through game theory. In *Software Maintenance, 2010 IEEE International Conference*, pages 1–5, 2010.

[CBYP05] Venkata U. B. Challagulla, Farokh B. Bastani, I-Ling Yen, and Raymond A. Paul. Empirical assessment of machine learning based software defect prediction techniques. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, WORDS '05, pages 263–270, Washington, DC, USA, 2005. IEEE Computer Society.

[CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object-oriented design. In *Conference Proceedings on Object Oriented Programming Systems, Languages, and Applications*, pages 197–211, 1991.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[CLMM05] Yania Crespo, Carlos López, Esperanza Manso, and Raúl Marticorena. Language independent metric support towards refactoring inference. In *Proceedings of the 9th Workshop on QAOOSE*, pages 18–29, 2005.

[CMC14a] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Detecting software design defects using relational association rule mining. *Knowledge and Information Systems*, 2014. DOI: 10.1007/s10115-013-0721-z (available online at http://link.springer.com/article/10.1007/s10115-013-0721-z.

[CMC14b] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Software defect prediction using relational association rule mining. *Information Sciences*, 264:260–278, 2014.

[CS06] I.G. Czibula and G. Serban. Improving systems design using a clustering approach. *International Journal of Computer Science and Network Security*, 6(12):40–49, 2006.

[CS07] Istvan Gergely Czibula and Gabriela Serban. Hierarchical clustering for software systems restructuring. *INFOCOMP Journal of Computer Science*, 6(4):43–51, 2007.

[CSTM06] Alina Campan, Gabriela Serban, Traian Marius Truta, and Andrian Marcus. An algorithm for the discovery of arbitrary length ordinal association rules. In *Proceedings of the 2006 International Conference on Data Mining (DMIN)*, pages 107–113, 2006.

[DABH11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, and Andre Cavalcante Hora. Software metrics for package remodularization. Technical report, Institut National de Recherche en Informatique et en Automatique, 2011.

[DbU] Commons dbutils. http://commons.apache.org/proper/commons-dbutils/index.html.

[DDN00] Serge Demeyer, Stephance Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 166–177, 2000.

[DG77] Persi Diaconis and R. L. Graham. Spearman's footrule as a measure of disarray. *Journal of the Royal Statistical Society*, 39(2):262–268, 1977.

[DK05] Scott Dick and Abraham Kandel. *Computational Intelligence in Software Quality Assurance.* Series in Machine Perception and Artificial Intelligence. World Scientific Publishing, 2005.

[EL] El. http://commons.apache.org/proper/commons-el/.

[Ema] Email. http://commons.apache.org/proper/commons-email/.

[FBZ12] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11:5:1–38, 2012.

[Fow] Martin Fowler. Refactoring malapropism. http://martinfowler.com/bliki/RefactoringMalapropism.html.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, 1999.

[FTCS09] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiu, and Jorg Sander. Decomposing object-orietend class modules using an agglomerative clustering technique. In *Proceedings of International Conference on Software Maintenance*, pages 93–101, Edmonton, Canada, 2009.

[FTP13] Ftp4j, 2013. http://sourceforge.net/projects/ftp4j/.

[FTSC11] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: Identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE*, pages 1037–1049, 2011.

[Gam] E. Gamma. JHotDraw Project. http://sourceforge.net/projects/jhotdraw.

[GMCS04] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *ISSRE*, pages 417–428, 2004.

[GWI11] Isaac Griffith, Scott Wahl, and Clemente Izurieta. Truerefactor: An automated refactoring tool to improve legacy system and application comprehensibility. In *Proceedings of the ISCA 24th International Conference on Computer Applications in Industry and Engineering*, pages 316–321, 2011.

[Han05] Jiawei Han. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Har12] Mark Harman. The role of artificial intelligence in software engineering. In *1st International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2012.

[HDF12] A.A. Shahrjooi Haghighi, M. Abbasi Dezfuli, and S.M. Fakhrahmad. Applying mining schemes to software fault prediction: A proposed approach aimed at test cost reduction. In *Proceedings of the World Congress on Engineering 2012 Vol I*, WCE 2012,, pages 1–5, Washington, DC, USA, 2012. IEEE Computer Society.

[HJ01] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[HK81] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[HKI08] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20:435 – 461, 2008.

[HM95] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.

[HMZ09] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search based software enginerring: A comprehensive analysis and review of trend techniques and applications. Technical report, Department of Computer Science, King's College, London, 2009.

[HS96] B. Henderson-Sellers. *Object-Oriented Metrics Measures of Complexity.* Prentice-Hall, 1996.

[HT07] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *Conference on Genetic and Evolutionary Computation*, pages 1106–1113, 2007.

[IPl13] iplasma, 2013. http://loose.upt.ro/reengineering/research/iplasma.

[ISIE12] Safwat M. Ibrahim, Sameh A. Salem, Manal A. Ismail, and Mohamed Eladawy. Identification of nominated classes for software refactoring using object-oriented cohesion metrics. *International Journal of Computer Science Issues*, 9(2):68–76, 2012.

[ISO13] Iso8583, 2013. http://sourceforge.net/projects/j8583/.

[JDe13] Jdeodorant, 2013. http://www.jdeodorant.com/.

[JLZ11] Yuan Jiang, Ming Li, and Zhi-Hua Zhou. Software defect detection with rocus. *Journal of Computer Science and Technology*, 26(2):328–342, 2011.

[Kan03] Ronald Kirk Kandt. A software defect detection methodology, 2003.

[KMMiM08] Yasutaka Kamei, Akito Monden, Shuji Morisaki, and Ken ichi Matsumoto. A hybrid faulty module prediction using association rule mining and logistic regression analysis. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurements(ESEM)*, pages 279–281, 2008.

[KSBW11] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer. Search-based design defects detection by example. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*, pages 401–415, 2011.

[KVGS09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software*, pages 305–314, 2009.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Addison Wesley Professional, third edition, 2004.

[LH93] Wei Li and Sallie Henry. Object oriented metrics which predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[LHM98] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 80–86, 1998.

[LLWW95] Y. S. Lee, B. S. Liang, S. F. Wu, and F. J. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proceedings of International Conference on Software Quality*, Maribor, Slovenia, 1995.

[LMW01] Bing Liu, Yiming Ma, and Ching-Kian Wong. *Data Mining for Scientific and Engineering Applications*, chapter Classification Using Association Rules: Weaknesses and Enhancements. Kluwer Academic, 2001.

[LZWZ12] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012.

[Mai09] Sayyed Garba Maisikeli. *Aspect Mining Using Self-Organizing Maps With Method Level Dynamic Software Metrics as Input Vectors*. PhD thesis, Graduate School of Computer and Information Sciences Nova Southeastern University, 2009.

[Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Politechnica University Timisoara, Faculty of Automatics and Computer Science, 2002.

[Mar10] Zsuzsanna Marian. Solving the subset sum problem with dna computation. In *Proceedings of the National Symposium ZAC*, pages 25–29, 2010.

[Mar12a] Zsuzsanna Marian. Aggregated metrics guided software refactoring. In *Proceedings of the 8th IEEE International Conference on Intelligent Computer Communication and Processing*, pages 259–266, 2012.

[Mar12b] Zsuzsanna Marian. Software metrics based refactoring: a case study. In *Proceedings of the National Symposium ZAC*, pages 59–64, 2012.

[Mar12c] Zsuzsanna Marian. A study on hierarchical clustering based software restructuring. *Studia Universitatis "Babes-Bolyai" Informatica*, LVII(2):20–31, 2012.

[Mar13a] Zsuzsanna Marian. On the software metrics influence in relational association rule-based software defect prediction. *Studia Universitatis "Babes-Bolyai" Informatica*, LVIII(4):35–48, 2013.

[Mar13b] Zsuzsanna Marian. A study on association rule mining based software defect detection. *Studia Universitatis "Babes-Bolyai" Informatica*, LVIII(1):42–57, 2013.

[Mar14a] Zsuzsanna Marian. Faos - a framework for analyzing object-oriented software systems. *Studia Universitatis "Babes-Bolyai" Informatica*, 2014. Under review.

[Mar14b] Zsuzsanna Marian. On evaluating the structure of software packages. *Studia Universitatis "Babes-Bolyai" Informatica*, LIX(1):46–58, 2014.

[MC11] Iman Hemati Moghadam and Mel Ó. Cinnéide. Code-imp: A tool for automated search-based refactoring. In *Proceeding of the 4th Workshop on Refactoring Tools*, pages 41–44, Honolulu, USA, 2011.

[MCB11] Zsuzsanna Marian, Cosmin Coman, and Attila Bartha. Learning to play the guessing game. *Studia Universitatis "Babes-Bolyai" Informatica*, LVI(2):119–124, 2011.

[MCC12] Zsuzsanna Marian, Gabriela Czibula, and Istvan-Gergely Czibula. Using software metrics for automatic software design improvement. *SIC Journal, Studies in Informatics and Control*, 21(3):249–258, 2012.

[MCC14] Zsuzsanna Marian, Gabriela Czibula, and Istvan Gergely Czibula. Software packages refactoring using a hierarchical clustering-based approach. *Fundamenta Informaticae*, 2014. Under review.

[MGF07] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[MGL06] Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 297–300, 2006.

[MHH03] Kiarash Mahdavi, Mark Harman, and Robert Mark Hierons. A multiple hill climbing approach to software module clustering. In *Proceedings of the International Conference on Software Maintenance*, pages 315–324, 2003.

[Mit06] Tom M. Mitchell. The discipline of machine learning. Working paper, 2006.

[MKD13] Usman Mansoor, Marouane Kessentini, and Slim Bechikhand Kalyanmoy Deb. Code-smells detection using good and bad software design examples. Technical report, University of Michigan, 2013.

[MMCG99] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *In Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59, 1999.

[MML01] Andrian Marcus, Jonathan I. Maletic, and K. I. Lin. Ordinal association rules for error identification in data sets. In *Proceedings of the 10th International Conference on Information and Knowledge Management*, pages 589–591, 2001.

[Moh06] Naouel Moha. Detection and correction of design defects in object-oriented architectures. In *Doctoral Symposium, 20th edition of the European Conference on Object-Oriented Programming*, 2006.

[MS10] Zsuzsanna Marian and Christian Săcărea. Using contextual topology to discover similarities in modern music. In *Proceedings of the IEEE International Conference on Automation Quality and Testing, Robotics*, volume 3, pages 1–6, 2010.

[Mun05] Matthew James Munro. Product metrics for automatic identification of “bad smell” design problems in java source code. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, 2005.

[Nas] Nasa software defect datasets.

[OC08] Mark O’Keeffe and Mel Ó. Cinnéide. Search-based refactoring for software maintenance. *The Journal of Systems and Software*, 81:502–516, 2008.

[PBP$^+$13] Fabio Palomba, Gabriele Bavota, Massimiliani Di Penta, Rocco Oliverto, Andrea de Lucia, and Denys Poshyvanyk. Detecting bad smells in source codeusing change history information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013.

[PJL13] Wei-Feng Pan, Bo Jiang, and Bing Li. Refactoring software packages via community detection in complex software networks. *International Journal of Automation and Computing*, 10(2):157–166, 2013.

[Pro13] Profiler4j, 2013. http://sourceforge.net/projects/profiler4j/.

[rlf] Reinforcement learning framework. http://www.cs.ubbcluj.ro/$\sim$gabis/rl.

[RR11] Akepogu Ananda Rao and Kalam Narendar Reddy. Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique. *International Journal of Computer Science Issues*, 8(2):185–194, 2011.

[RRRAR12] D. Rodríguez, R. Ruiz, J. C. Riquelme, and J. S. Aguilar-Ruiz. Searching for rules to detect defective modules: A subgroup discovery approach. *Information Sciences*, 191:14–30, May 2012.

[SCC06] Gabriela Serban, Alina Câmpan, and Istvan Gergely Czibula. A programming interface for finding relational association rules. *International Journal of Computers, Communications & Control*, I(S.):439–444, June 2006.

[SKR08] Santonu Sarkar, Avinash C. Kak, and Girish Maskeri Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering*, 34(5):700–720, 2008.

[SMV09] Adrian Sterca, Zsuzsanna Marian, and Alexandru Vancea. Distortion-based media-friendly congestion control. In *Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques*, pages 265–267, 2009.

[Spe04] C. Spearman. The proof and measurement of association between two things. *Amer. J. Psychol.**15***, pages 72–101, 1904.

[SS07] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring - does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, 2007.

[SSB06] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Conference on Genetic and Evolutionary Computation*, pages 1909 – 1916, 2006.

[SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 30–38, 2001.

[ST98] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 99–110, 1998.

[ŢIM09] Radu Ţurcaş, Oana Iova, and Zsuzsanna Marian. The autonomous robotic tank (art): an innovative lego mindstorm nxt battle vehicle. In *Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques*, pages 95–98, 2009.

[TK03] Ladan Tahvildari and Kostas Kontogiannis. A metric-based approach to enhance the design quality through meta-pattern transformations. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 183–192, 2003.

[TLM11] Doina Tatar, Mihaiela Lupea, and Zsuzsanna Marian. Text summarization by formal concept analysis approach. *Studia Universitatis "Babes-Bolyai" Informatica*, LVI(2):7–12, 2011.

[Tuf11] Stphane Tuffry. *Data Mining and Statistics for Decision Making.* John Wiley and Sons, 2011.

[Win13] Winrun4j, 2013. http://sourceforge.net/projects/winrun4j/.

[ZT05] Du Zhang and Jeffrey J. P. Tsai. *Machine Learning Applications in Software Engineering.* Series on Software Engineering and Knowledge Engineering. World Scientific Publishing, 2005.