A New Approach for Software Quality Assessment Based on Automated Code Anomalies Detection

Andrea Biaggi, Umberto Azadi and Francesca Arcelli Fontana
Università degli Studi di Milano-Bicocca, Milano, Italy
{*a.biaggi1, u.azadi*}*@campus.unimib.it, arcelli@disco.unimib.it*

Keywords: Empirical Study, Software Quality Assessment, Software Evolution, Code Anomalies Detection, Artificial Immune Systems.

Abstract: Methods and tools to support quality assessment and code anomaly detection are crucial to enable software evolution and maintenance. In this work, we aim to detect an increase or decrease in code anomalies leveraging on the concept of microstructures, which are relationships between entities in the code. We introduce a tools pipeline, called Cadartis, which uses an innovative immune-inspired approach for code anomaly detection, tailored to the organization's needs. This approach has been evaluated on 3882 versions of fifteen open-source projects belonging to three different organizations and the results confirm that the approach can be applied to recognize a decrease or increase of code anomalies (anomalous status). The tools pipeline has been designed to automatically learn patterns of microstructures from previous versions of existing systems belonging to the same organization, to build a personalized quality profiler based on its codebase. This work represents a first step towards new perspectives in the field of software quality assessment and it could be integrated into continuous integration pipelines to profile software quality during the development process.

1 INTRODUCTION

Code anomalies is a term that refers to several categories of code sub-optimalities that include code smells, bugs, antipatterns and several other software issues (Fowler, 2018). A system containing such anomalies should be maintained to avoid progressive software degradation. Therefore, methods and tools to support quality assessment and code anomaly detection are crucial during both software development and software evolution and maintenance. In fact, during the development of large software systems where several developers are involved, it is useful to have tools that report a kind of software quality profile so that the team leader can monitor and correct some aspects (such as code anomalies) during the development process.

This work aims to identify a decrease or increase in software quality through an approach based on code anomaly detection. We decided to accomplish this objective by using a rule-based approach that exploits a particular kind of structure, called the microstructure (MS). These MSs are defined as facts or relationships between two entities in the code (e.g. classes, attributes, methods) (Arcelli Fontana et al., 2013). Examples of microstructures are *Data Manager* (Gil and Maman, 2005), which is a MS that represents a class where all methods are either getters or setters, and *Empty Method*, a MS that occurs when a method does not contain any implementation except for returning a primitive type. The advantage of using these metrics instead of the classical ones (LOC, CC, ...) (Fenton and Bieman, 2014) is that the microstructures incorporate information about architectural and design decisions, and they can be directly influenced by changes that concern these aspects. Therefore, our goal is to extract rules, expressed in terms of microstructures, that could be able to predict an increase or a decrease in code anomalies. These rules can be seen as patterns of microstructures, which we will call in this paper Microstructure Patterns (MSP). Furthermore, we require human-readable rules to allow developers to understand why our tools pipeline could suggest an increase or decrease of code anomalies in order to understand which problems caused a given rule to be triggered.

The problem defined in this way can be tackled as a problem of anomaly detection. To understand this conceptual jump we can use an example that intentionally exasperates this concept: if we analyze the history of a system and we observe that usually 100 code anomalies are identified, and then in the next re-

lease of the system we identify 1000 code anomalies, we could state that the status of the system is anomalous. The objective is to identify some rules that can describe why such anomalous status has occurred and allow developers to take the right decisions. These rules represent the kind of *anomaly* that we aim to detect: an anomalous status represented through an increase of code anomalies in the monitored system. In the context of data mining, a paradigm used to describe problems of anomaly detection is through the artificial immune systems (AIS) (Dasgupta, 2012), which represent a class of rule-based machine learning systems, inspired by the principles and processes of the human immune system, which is particularly suited to obtain human-readable results.

To perform the detection of such anomalies, this work introduces a tools pipeline called Cadartis, (Code Anomalies Detection using ARTificial Immune Systems). The pipeline analyzes the evolution in terms of versions of Java software systems to infer rules that measure the quality of the system, which are then used to assess newer versions. The main benefit of this approach is that the rules are automatically inferred through machine learning algorithms based on existing systems' history belonging to the same organization, thus defining a kind of *personalized* quality profiler tailored for the organization's needs. Furthermore, such a quality profiler can be used both on existing projects, during the maintenance phase, and on new projects during development.

The paper is organized through the following sections: in Section 2 we introduce some related works on code anomalies detection and works on the exploitation of AIS, in Section 3 we describe the different components of the Cadartis pipeline, and two main Research Questions, in Section 4 we describe the results obtained in the evaluation of Cadartis on 15 projects of different organizations (Apache, Eclipse and Google). Finally, we report the threats to validity of our work and the possible future developments.

2 RELATED WORK

In the past few years, artificial immune systems have been applied in different areas of software engineering, from software cost estimation (SCE) (Lee and Kwon, 2009), software testing, to software maintenance and evolution (Parrend et al., 2018). For instance, the work of Gharehchopogh et al. (Gharehchopogh et al., 2014) proposes a hybrid model for SCE based on the combination of AIS and genetic algorithms. Regarding software testing, there are several works involving AIS. For example, the work of Liaskos and Roper (Liaskos and Roper, 2008) in the context of search-based test case generation. As for software maintenance and evolution, the work of Hassaine et al. (Hassaine et al., 2010) proposes an AIS-based model for software design smell detection to compare it with state-of-art approaches such as the DECOR tool (Moha et al., 2009). According to our knowledge, we have not found approaches for code anomaly detection based on artificial immune systems, as the one described in this paper, used to build a personalized software quality profile. While according to code anomalies detection, the most commonly detected anomalies are code smells, software bugs and other issues/code violations such as those computed for example by the tool SonarQube[1]. Many approaches have been proposed for code smells detection implemented in a variety of tools which exploit different techniques, such as: static code analysis, refactoring identification e.g., JDeodorant (Tsantalis and Chatzigeorgiou, 2011) or metrics computation, used by many tools e.g., inFusion, inCode, PMD, Checkstyle, and JCodeOdor. These approaches rely on a single metric or a combination of metrics that correspond to code properties relevant to a given smell. Other tools exploit techniques based on a dedicated domain-specific language (DSL), that use high-level abstractions to uncover design anomalies, e.g., DECOR (Moha et al., 2009); machine learning classifiers (Arcelli Fontana et al., 2016; Azadi et al., 2018; Maiga et al., 2012); techniques based on Bayesian belief networks (Khomh et al., 2009), on the analysis of software repositories (Palomba et al., 2013; Rapu et al., 2004) and on design change propagation probability (Rao and Reddy, 2007).

The variety of approaches leads to considerable differences in the anomalies detected (Mantyla, 2005; Arcelli Fontana et al., 2012), which make it difficult to analyze anomalies and compare the results. Through our AIS-based approach, we aim to provide a new perspective in this area.

3 CADARTIS

Cadartis (Code Anomalies Detection using ARTificial Immune Systems) is a tools pipeline designed for organizations interested in building an artificial immune system (AIS) based on their codebase. As illustrated in Figure 1, the pipeline consists of two main components: the first one, called *AIS Automated Learner*, is responsible for analyzing the dataset through a machine learning process and reporting a rule-based

[1] https://www.sonarqube.org/

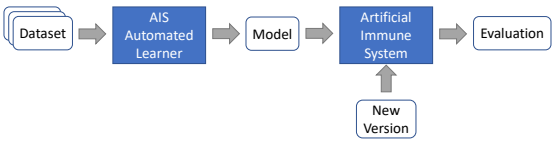

Figure 1: Cadartis pipeline.

human-readable model from which it is possible to infer the *microstructure patterns* (MSP), that are the rules used to identify a possible system degradation of the system (an increase of anomalies). The second one, *Artificial Immune System*, is the component that is responsible for analyzing the versions to be assessed to report an evaluation according to the MSP reported in the model, as explained in Section 3.4.

Specifically, a *microstructure pattern* represents a recurring scheme of microstructures expressed by a set of conditions that are associated with an increase or a decrease of the anomalies.

More formally, a MSP is expressed as: $MS_i \leq k_i \ \& \ MS_j > k_j \longrightarrow variation$, where MS_i and MS_j are instances of microstructures, i.e. one of the features of the dataset described in Section 3.2, k_i and k_j are the numbers of instances of the corresponding microstructure and *variation* is the target variable and it can assume two possible values: "Increase", if the number of code anomalies has increased compared to the previous version, "Decrease" otherwise (see Section 3.3.1).

Thanks to these patterns it is possible *1)* to assess whether there is a relationship between the variation of the code anomalies and the microstructures and *2)* to understand which microstructures contribute the most in the variation of code anomalies. The components are released as standalone Java applications [2] , so that they can be used separately, for example in a continuous integration pipeline.

3.1 Problem Statement

This work aims to answer the following research questions (RQs):

RQ1*: Can the microstructures patterns (MSP) describe and help to predict an anomalous state of the system?*

RQ2: *Which microstructures in the MSP are the most effective to describe the overall code quality?*

The answers to these research questions are useful to understand if the approach proposed in this study can be applied to the problem issued (RQ1), thus opening new perspectives in the field of software quality assessment. Moreover, answering RQ2 allows us to understand whether specific microstructures are more useful than others to recognize code anomalies.

3.2 Collected Data

In this work, we analyze the evolution, in terms of versions, of entire software systems of specific organizations to collect the microstructures and the number of code anomalies. The dataset is made of 47 features variables and one target variable[3]. The features are the microstructures, which are divided into three categories: Elemental Design Patterns (EDP) (Arcelli Fontana et al., 2013), Micro Pattern (MP) (Gil and Maman, 2005) and Design Pattern Clues (DPC) (Arcelli Fontana et al., 2013). The target variable in this step is the number of code anomalies as reported by the PMD tool, which will be then discretized to represent the variation from one version to the next one (see Section 3.3.1). The following sections explain how all the data are collected.

3.2.1 Microstructure Extraction

Since we were not able to find any tool for the microstructure extraction suitable for our purposes, a new one has been developed. It is distributed as a standalone Java application and takes as input the directory of the Git repository to be analyzed using static analysis. Since the microstructures are meant to be mechanically recognizable (Gil and Maman, 2005), this approach has already been used and tested in several other applications (Zanoni et al., 2015) (Arcelli Fontana et al., 2005) in the context of specific tools or Eclipse plug-ins. The definitions of the considered 47 MS can be found in the replication package.

3.2.2 Code Anomalies Extraction

In this work, the number of code anomalies has been considered as a target and they are extracted using a tool called PMD[4]. This tool is able to recognize several anomalies belonging to different categories according to the PMD Java rule reference[5]. We considered the following categories: Design, Security, Performance, Multithreading, Error Prone. We selected PMD as a tool for software quality assessment because it is widely employed (Allier et al., 2012) and it can be easily used standalone from a command line, hence particularly suitable for our purposes. As a preliminary analysis, we computed squared Pearson correlation (R^2) between the variable and the target and

[2]Replication package available `here`

[3]Dataset available inside `replication package`

[4]https://pmd.github.io/

[5]https://pmd.github.io/latest/pmd_rules_java.html

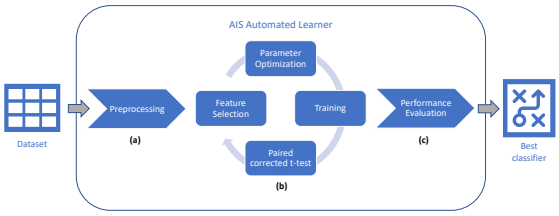

Figure 2: Pipeline of the AIS Automated Learner.

measured how well a variable predicts the value of the target. The R^2 value ranges from 0 to 1, where 0 means that a variable is independent from the target, while 1 means that a variable accurately predicts the value of the target. The results showed near-zero values for each variable, so we can conclude that the features are almost independent by the target.

3.3 AIS Automated Learner

This section describes the component responsible for analyzing the dataset to infer the microstructure patterns that define the rule set of the artificial immune system. Thanks to this component it is possible to personalize the analysis to report feedback based on the systems' history for each organization. To do this, the collected data are the input of the machine learning process illustrated in Figure 2 and explained in the following sections. In particular, Section 3.3.1 explains which preprocessing operations are made on the dataset before the machine learning analysis and the feature selection criteria to reduce the dimensionality of the dataset (Figure 2 (a) and (b)); Section 3.3.2 explains which algorithms are used in this study, how the parameter optimization is performed and how the trained models are compared to find the best one (Figure 2 (b) and (c)).

3.3.1 Preprocessing and Feature Selection

The dataset target variable is discretized to be suitable for a classification task. The discretization consists in computing the difference between the number of anomalies per class in the current version and the number of anomalies per class in the previous version so that the target becomes *Increase* if the difference is positive, or *Decrease* otherwise. Moreover, the features related to the microstructures have a very high variance, which can cause a decrease in the performance of the classification task. To overcome this problem, the dataset has been normalized according to the min-max feature scaling method (Han et al., 2011) that consists of scaling all the values in a fixed range $[a,b]$. After that, we reduced the dimensionality of the dataset to remove redundant features and to consequently improve the performance during the classification task and avoid the phenomenon known as *curse of dimensionality* (Bellman and Dreyfus, 2015). The first step was performing a correlation analysis to understand the degree of correlation among features to have an insight into which threshold could be suitable for a correlation-based feature selection. In this work, the feature selection is made by using a correlation interval instead of a single value to perform the classification, and the optimal threshold in this interval is chosen through an iterative process (Figure 2 (b)).

3.3.2 Algorithms Selection and Comparison

The choice of the algorithm has been guided by the need of having a human-readable model. Thanks to this kind of models it is possible to infer human-readable rules, the microstructures patterns, that associate a combination of microstructures with an outcome. As a consequence, three well known algorithms have been chosen: J48 (Quinlan, 2014), RandomForest (Breiman, 2001) and JRip (Cohen, 1995). All of them have been used with and without the boosting techniques AdaBoostM1 (Freund et al., 1996), therefore in total six models have been trained and tested. Furthermore, we used pruning techniques to reduce the depth of rules/trees, in order to make them easier to be manually inspected. For each algorithm, a step of parameter optimization has been performed. The goal of this phase is to find the best combination of parameters to achieve the most performing classifier for each algorithm at every step of classification to be computed in the correlation interval. For each possible configuration the classifier is trained in a 10-fold cross-validation and the best parameters are chosen according to the minimum error rate. To evaluate the overall best classifier, the six classifiers took part in a paired corrected t-test that produces a ranking of algorithms. To achieve this, the dataset has been divided into two partitions: the training partition consists of four out of five projects belonging to the organization, while the test partition is the fifth. This separation has been chosen because it simulates the introduction of a new project in the organization's codebase. The test compares the classifiers to one another and computes whether the difference in performances among algorithms is statistically significant, producing a ranking ordered by the number of victories, that is the number of times that a classifier outperformed another. The paired corrected t-test with ranking was performed by using the implementation provided by Weka Experimenter (Hall et al., 2009).

3.4 Artificial Immune System

The artificial immune system (AIS) is the component responsible for assessing new versions of systems. As shown in Figure 1, this component takes as input the model reported by the learner that has been trained on the existing codebase of a specific organization and contains the microstructure patterns.

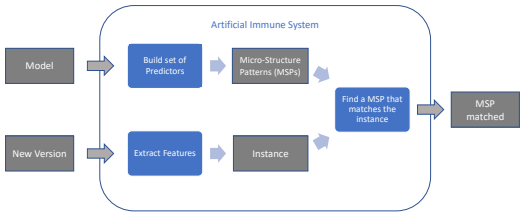

Figure 3: AIS Component Workflow.

As illustrated in Figure 3, whenever a new version of a system of an organization is ready, that can be either a new version of an existing system used for the analysis or a version of a new system that the organization wants to profile during the development, the system analyzes the new version and extracts the features, as explained in Section 3.2, thus building an instance of the dataset. The artificial immune system finds the MSP that matches the input instance and reports an evaluation that consists of the outcome for that instance together with the MSPs that have been matched.

4 RESULTS

Cadartis has been tested by analysing 15 projects, taken from 3 different organizations: Apache, Eclipse and Google. All the projects can be found on GitHub[6] and they are described in the replication package. In this section, we provide an overview of the performance of the best classifier for the three different organizations (Section 4.1), to evaluate and discuss strong and weak points that emerged during the learning of the patterns. Moreover, we illustrate the microstructure patterns (MSP) reported for each organization (Section 4.2). The replication package containing the tools developed and the data collected is available[7].

4.1 Performance Evaluation

We analyze the performance of the three models, illustrating the best-performing algorithm as a result of the paired corrected t-test with ranking discussed in Section 3.3.2 and the performance achieved by the classifiers using the most common performance metrics: Precision, Recall, F-measure and Accuracy. Table 1 reports the resulting performance metrics of the best classifier for each organization. From the metrics reported in Table 1 regarding all the organizations, it is possible to conclude that the classifiers perform better when predicting instances that belong to the "Decrease" class, while the performances decrease, with different degrees, when it comes to predicting instances of "Increase" class, but the overall performance for each organization remains satisfactory.

Table 1: Resulting performance metrics of the best classifier for each organization.

	Metric	Decrease	Increase	Best
Apache	Precision	84.26 %	55.36 %	J48
	Recall	78.45 %	64.58 %	
	F-measure	81.25 %	59.62 %	
	Accuracy	74.39 %	74.39 %	
Eclipse	Precision	85.00 %	32.26 %	JRip
	Recall	66.93 %	57.14 %	
	F-measure	74.89 %	41.24 %	
	Accuracy	64.81 %	64.81 %	
Google	Precision	89.23 %	22.00 %	J48
	Recall	74.84 %	44.00 %	
	F-measure	81.40 %	29.33 %	
	Accuracy	70.56 %	70.56 %	

4.2 Results on MSPs

The outcome of this experimentation is a catalog of 73 MSPs, which is available inside replication package[8]. Each MSPs has been manually validated to assess whether the microstructures involved in a pattern can affect positively or negatively the software quality according to known object-oriented programming principles (Savitch, 2000) object-oriented design principles (Martin, 2002), software quality attributes or bad practices (Brown et al., 1998).

These principles have been used to understand whether the presence of a microstructure can lead to a better or worse software quality and according to which principles. This process allowed us to find several interesting *facts* (F) on the MS related to the principle enforced or violated. By analyzing the definition of each microstructure[9], we observed that some MSs can be seen as indicators of good or bad programming practices.

[6] https://github.com
[7] Data available inside `replication package`
[8] Microstructure Patterns detected available `here`
[9] Microstructure catalog available `here`

An example of MSP and the related *facts* found through the analysis follows, where Record, Revert Method and Data Manger are the names of three MS: $Record \geq 49$ & $RevertMethod \geq 12$ & $DataManager \geq 76 \longrightarrow$ Increase.

F1: *Record* instances represent classes that declare only public fields without methods. Instances of this MS represent a bad practice. Hence, many Record instances could cause an increase in anomalies.

F2: *Revert Method* instances involve bypassing the current class' implementation of a method, instead using the superclass' implementation, thus violating the encapsulation principle.

F3: *Data Manager* is a MS that represents classes composed only by private fields, getters and setters. This is considered bad practice because it corresponds to the Data Class code smell, hence many instances of this MS could lead to an increase in anomalies. These MS provide hints of bad quality, hence a possible increase of code anomalies. We did the same analysis on all the other MSs involved in all the 73 MSPs. It is important to note that not all the MSPs can be explained with the aforementioned criteria, in fact around 40% of them cannot be explained with any of the principles taken into consideration. The reason is that some of the definitions of the MS are neutral, meaning that it is not possible to assess whether many or few instances of them could cause an increase or a decrease in the anomalies.

4.3 Answers to the RQs

Concerning ***RQ1***, we can observe that several MSPs reflect well-known software quality attributes, object-oriented principles or bad practices that have been taken into consideration during the manual evaluation of the MSPs. This result highlights that the obtained rules are already useful and relevant and we can argue that the MSP can already help to predict an anomalous state since they are already able to detect when the principles and practice mentioned above are violated. However, we are also quite confident that several additional insights might be extracted by those who developed and maintained the system since they were familiar with the code and the development process. Furthermore, we can argue that these insights might be the most valuable since our purpose was to create a customise quality profiler. We are also aware that the performance of the classifier are quite low however we still think that should be considered as a good baseline for the problem issued in this research because *1)* the classification task tackled in this work was particularly hard since it involved learning patterns on a set of projects and validating the results on a different set of projects and *2)* we decided to use only algorithms that output human-readable model, which often performs worse compared to other algorithms commonly used, such as Support Vector Machines and Neural Network. Concerning ***RQ2***, there is no category of microstructures (EDP, MP, DPC) more effective than another to describe the code quality. All the patterns are composed of microstructures belonging to different categories. It's worth mentioning that specific categories of patterns are more present in some organizations than others. This information supports the decision to focus our work on the customization of the AIS accomplished through the AIS Automated Learner, described in Section 3.3. Finally, we provide some examples of data that we were able to extract from the manual evaluation of the MSPs that support the answer to the RQs above, which are reported in Table 2. An example concerns the MS *Data Manager*, *Record*, *Trait* and others indicated in the first row of Table 2, that is MS whose presence is always associated with a degradation of the system quality because they all represent violations of different object-oriented programming principles.

Table 2: Qualitative analysis of the results.

Result	Related microstructures
Increase	*Data Manager, Record, Trait, Empty Concrete Product Getter, Controlled Instantiation, Revert Method*
Decrease	*Concrete Product Returned, Extend Method, Factory Parameter, Conglomeration, Controlled Exception, Sink*
Increase if lack or exceed	*Redirect, Delegate, Delegated Conglomeration, Abstract Class, Retrieve*

4.4 Threats to Validity

As for threats to internal validity, the usage of only the number of one kind of anomaly as a target may have affected the results, since the usage of other measures may produce different results. However, PMD is one of the commonly used commercial tools for code anomaly detection, so the choice of this tool remains relevant. When it comes to external validity, the number of projects considered for each organization is limited and doesn't consider every application domain. For this reason, the results may be affected by a lack of generalization within an organization, because it is possible that some MSPs can

be inferred only in relation to a specific application domain. In any case, all the projects belong to relevant and significant organizations (Google, Eclipse, Apache). Finally, we were not able to involve the developers of the three organizations taken into account during the evaluation of the MSP, this is a limitation since one of the aims of this work was to provide a *customized* software quality assessment, that represents a more useful way than a generalized rule set, but this aspect will be faced in a future extension of this work. To mitigate this threat, well-known software quality principles and best practices have been taken into consideration during the evaluation of the MSPs. This decision has led us to discover some patterns that often reflect the well-known software quality issues that have been derived from those principles and practices. This result highlights that the obtained rules are already useful and relevant and it is very likely that a practitioner familiar with how the code has been developed in each organization might be able to notice several additional insights by analyzing the MSPs. Moreover, all the data of this study can be found in the replication package[10].

5 CONCLUSIONS

In this work, we introduced a tools pipeline, called Cadartis, for the detection of an anomalous status of software systems (an increase of code anomalies) based on an artificial immune system approach. It is designed to automatically learn patterns from previous versions of existing systems belonging to the same organization, to build a personalized quality profiler based on its codebase. The patterns, that are automatically inferred from the data, are composed of MSs. The results have been evaluated on 3882 versions of fifteen projects belonging to three different organizations, thus building three different artificial immune systems, one for each organization, to assess *1)* whether this approach can be applied for software quality assessment and *2)* to understand which relationships exist between MS and the variation of the code anomalies. The identified MS patterns can be useful to developers, team leaders and quality analysts involved in the software development or maintenance process in order to understand which MSs affect the quality and consequently plan a strategy of corrective maintenance.

Our results confirm that this approach can be applied in this field and since there is no other study to compare the results with, the performance obtained represents a good baseline for other studies that aim to apply the same approach, although they have been affected by the constraint of choosing only intelligible learning algorithms. The results showed that there is no category more useful than another, however, a given category is more or less representative based on the organization taken into account.

Considering the results, future developments will focus on two main aspects. The first one is improving the performance of the models, either by using other classes of algorithms or by experimenting with other approaches (active and semi-supervised learning).

The second regards the reduction of the number of rules that cannot be associated with the outcome. To do this, it is necessary to introduce more information in the dataset, such as the category of the anomalies explained in Section 3.2.2. Another alternative could be to change the target using other tools to extract code anomalies, using quality indices or technical debt indices (Roveda et al., 2018), for example allowing organizations that already use a quality index to use it as a target for learning the patterns.

As for the pipeline components' development, the introduction of new algorithms requires the development of modules that can parse the output of the new models to build the set of patterns of the AIS. Moreover, since Cadartis can be used during the development of a pipeline of continuous integration, another direction can be the development of plugins, for the platforms that support it, to ease the distribution of the Artificial Immune System component.

[10]Replication package available `here`

REFERENCES

Allier, S., Anquetil, N., Hora, A., and Ducasse, S. (2012). A framework to compare alert ranking algorithms. In *2012 19th Working Conference on Reverse Engineering*, pages 277–285.

Arcelli Fontana, F., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5–1.

Arcelli Fontana, F., Maggioni, S., and Raibulet, C. (2013). Design patterns: a survey on their micro-structures. *Journal of Software: Evolution and Process*, 25(1):27–52.

Arcelli Fontana, F., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191.

Arcelli Fontana, F., Masiero, S., and Raibulet, C. (2005). Elemental design patterns recognition in java. In *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pages 196–205. IEEE.

Azadi, U., Arcelli Fontana, F., and Zanoni, M. (2018). Poster: Machine learning based code smell detection through wekanose. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 288–289. IEEE.

Bellman, R. E. and Dreyfus, S. E. (2015). *Applied dynamic programming*, volume 2050. Princeton university press.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Brown, W. H., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Cohen, W. W. (1995). Fast effective rule induction. In *Machine learning proceedings 1995*, pages 115–123. Elsevier.

Dasgupta, D. (2012). *Artificial immune systems and their applications*. Springer Science & Business Media.

Fenton, N. and Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC press.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Freund, Y., Schapire, R. E., et al. (1996). Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, volume 96, pages 148–156. Citeseer.

Gharehchopogh, F. S., Maleki, I., Ghoyunchizad, N., and Mostafaee, E. (2014). A novel hybrid artificial immune system with genetic algorithm for software cost estimation. *Magnt Research Report*, 2(6):506–517.

Gil, J. Y. and Maman, I. (2005). Micro patterns in java code. *ACM SIGPLAN Notices*, 40(10):97–116.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18.

Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier.

Hassaine, S., Khomh, F., Guéhéneuc, Y.-G., and Hamel, S. (2010). Ids: An immune-inspired approach for the detection of software design smells. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 343–348. IEEE.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., and Sahraoui, H. (2009). A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314. IEEE.

Lee, J.-k. and Kwon, K.-T. (2009). Software cost estimation using svr based on immune algorithm. In *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 462–466. IEEE.

Liaskos, K. and Roper, M. (2008). Hybridizing evolutionary testing with artificial immune systems and local search. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 211–220. IEEE.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Gueheneuc, Y.-G., and Aimeur, E. (2012). Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering*, pages 466–475. IEEE.

Mantyla, M. V. (2005). An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10–pp. IEEE.

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 268–278. IEEE Press.

Parrend, P., Guigou, F., Navarro, J., Deruyver, A., and Collet, P. (2018). For a refoundation of artificial immune system research: Ais is a design pattern. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1122–1129. IEEE.

Quinlan, J. R. (2014). *C4.5: programs for machine learning*. Elsevier.

Rao, A. A. and Reddy, K. N. (2007). Detecting bad smells in object oriented design using design change propagation probability matrix.

Rapu, D., Ducasse, S., Gîrba, T., and Marinescu, R. (2004). Using history information to improve design flaws detection. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 223–232. IEEE.

Roveda, R., Arcelli Fontana, F., Pigazzini, I., and Zanoni, M. (2018). Towards an architectural debt index. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 408–416. IEEE.

Savitch, W. J. (2000). *Java: an introduction to computer science and programming*. Prentice Hall PTR.

Tsantalis, N. and Chatzigeorgiou, A. (2011). Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782.

Zanoni, M., Arcelli Fontana, F., and Stella, F. (2015). On applying machine learning techniques for design pattern detection. *Journal of Systems and Software*, 103:102–117.