A Study on Architectural Smells Prediction

Arcelli Fontana, Francesca; Avgeriou, Paris; Pigazzini, Ilaria; Roveda, Riccardo

Published in:
Proceedings - 45th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2019

DOI:
10.1109/SEAA.2019.00057

Publication date:
2019

2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)

A Study on Architectural Smells Prediction

Francesca Arcelli Fontana
University of Milano-Bicocca
Milano, Italy
arcelli@disco.unimib.it

Paris Avgeriou
University of Groningen
Groningen, Netherland
paris@cs.rug.nl

Ilaria Pigazzini
University of Milano-Bicocca
Milano, Italy
i.pigazzini@campus.unimib.it

Riccardo Roveda
Alten Italia
Milano, Italy
riccardo.roveda@alten.it

***Abstract*—Architectural smells can be detrimental to system maintainability and evolvability, and represent a source of architectural debt. Thus, it is very important to be able to understand how they evolved in the past and to predict their future evolution. In this paper, we evaluate if the existence of architectural smells in the past versions of a project can be used to predict their presence in the future. We analyzed four Java projects in 295 Github releases and we applied four different supervised learning models for the prediction in a repeated cross-validation setting. We found that historical architectural smell information can be used to predict the presence of architectural smells in the future. Hence, practitioners should carefully monitor the evolution of architectural smells and take preventative actions to avoid introducing them and stave off their progressive growth.**

***Index Terms*—Architectural smells prediction and evolution, architectural technical debt.**

I. Introduction

Architectural smells (AS) [1], [2] correspond to architectural decisions that negatively impact internal software quality. Along evolution, AS may lead to a progressive architecture degradation, erosion and architectural debt [3]. Thus, we need to understand how AS evolved in the past and more importantly to *predict* how they may evolve in the future. This would allow us to better understand their temporal progression and work on preventing their creation.

In this paper, we propose a prediction model that uses historical AS data of four open source Java projects to predict the presence of architectural smells in future versions of the projects (for more details see Section 4). For each studied project, we considered almost all revisions found on their respective version control repository and we collected data related to four architectural smells: *Unstable Dependency*, *Cyclic Dependency*, *Hub-like Dependency* and *Implicit Cross Package Dependency*.

The detection of the above architectural smells is performed through a tool we had previously created, called Arcan [4]. To optimize the prediction, we applied four different supervised learning models in a repeated cross-validation setting and measured the obtained performance. The results of our analysis can be used by developers/maintainers in order to plan the refactoring and pay particular attention to the AS that can be used as predictors of AS in the future.

The paper is organized through the following sections: in Section II we introduce some related work; in Section III we briefly introduce the Arcan tool and the architectural smells detected; in Section IV we provide the definition and setup of our study; in Section V we present our main results for architectural smells prediction; in Section VI we outline the threats to validity of the work; in Section VII we conclude with the answer to the Research Question and some future directions of investigations.

II. Related Work

We briefly introduce some related work on prediction of quality issues. A large number of works in the literature use history-based analysis of projects to predict different issues that impact software quality, like code smells, changes or bugs. Examples of works on bug prediction, include Khomh et al. [5] who explore the presence of antipatterns for bug prediction by analyzing multiple versions of Eclipse and ArgoUML and Palomba et al. [6], [7] who explore if it is possible to improve bug prediction performance using an Intensity Index for code smells. Another example is the work of Maneerat [8] on code smell prediction through 7 machine learning techniques by using the number of bad-smells and software design model metrics as data sets. Their experiments show that bad smell prediction from software design models can predict bad smells earlier.

Oyetoyan et al. [9] performed an empirical study on different versions of 11 systems to analyze circular dependencies and change proneness; they found that classes involved in circular dependencies are more change-prone. Kouroshfar et al. [10] assessed if co-changes spanning multiple architecture modules are more likely to introduce bugs than co-changes that are within modules. Diaz-Pace et al. [11] studied the prediction of dependency-based architectural smells by looking at structural system characteristics. Shahbazian et al. [12] described an approach to automatically detect architecturally-significant issues and classify them based on the textual and non-textual information contained in each issue. To the best of our knowledge, there are no works that analyze and exploit the evolution of different architectural smells to predict architectural smells.

III. Architectural smells detection

The detection of the AS on Java projects has been done through the tool Arcan [4]. The tool relies on graph database technology. Once a Java project has been analyzed by Arcan, a new graph-database is created containing the structural dependencies of the projects. It is then possible to run detection algorithms on this graph to extract information about

978-1-7281-3421-5/19/$31.00 ©2019 IEEE
DOI 10.1109/SEAA.2019.00057

the analyzed project: package/class metrics and architectural issues. Arcan detects the architectural smells by focusing on instability metrics, where instability is considered as the predisposition of objects to change that can be captured through the violation of the dependency metrics of Martin [13]. Arcan detects three architectural smells without using the history of a project, Unstable Dependency (UD), Hub-Like Dependency (HL), Cyclic Dependency (CD) and one based on the history of the project, Implicit Cross Package Dependency (ICPD), which captures hidden dependencies among files belonging to different packages. CD smells are detected according to their shapes as those described by Al-Mutawa et al. [14]: tiny, clique, circle, chain and star shapes. The details of the detection algorithms of these architectural smells can be found in [15] and [16].

We have considered these four AS since they represent critical problems related to dependency issues: components that are highly-coupled and with a high number of dependencies cost more to maintain and hence can be considered more critical. In particular, the Cyclic Dependency smell is one of the most common smells and considered the most critical by developers [17]. Of course dependencies are not the only source of architecture problems and architectural technical debt; we consider other AS or debt indicators as interesting future work.

IV. Definition and setup of the case study

We are interested in understanding if through the history of the existing architectural smells in the project it is possible to predict the presence of architectural smells in future versions. This leads to the following research question:

RQ: *How well does the presence of architectural smells in the project's history support the prediction of architectural smells in the future?*

The answer to this RQ can help us to understand if there is a relation between the presence of architectural smells in past versions and future versions of the software. Developers can exploit such information to focus their attention on the refactoring of the involved smells. For example, if a Cyclic Dependency is a good predictor of itself or of another kind of smell, developers can try to remove this smell as soon as it appears. More generally, predicting the presence of architectural smells in the future can lead to understanding the co-evolution of architectural smells i.e. the way they appear and influence each other along the history of a project.

The replication package of this study is available online [1].

A. Selected projects

We analyzed four projects chosen from Github[2]: JGit, JUnit4, Commons-Math and Apache Tomcat. Table I summarizes the main features of the analyzed projects. For every project, we analyzed all versions (commits) available in the `master` branch. We selected these projects since they are written in Java and have more than 5 years of activity, at least 2K commits and at least 20 major releases. A dataset of projects with all their commits compiled was not available. Hence, we decided to create our own dataset, even if this task involved a great time investment for problems such as managing heterogeneous building systems (e.g. maven, ant), code compile errors and operating system dependent properties.

Table I: Selected projects

	JGit	JUnit4	Commons-Math	Apache Tomcat
Domain	library	library	library	app server
# Github releases	83	20	65	127
# Versions (Commits)	4840	2187	6286	6853
Date Start	2009-09-29	2000-12-03	2003-05-03	2006-03-28
Duration (years)	8	17	15	6
# Classes (first commit)	454	60	9	83
# Packages (first commit)	20	10	1	77
# Classes (last commit)	1315	445	1393	2218
# Packages (last commit)	45	32	20	141

B. Predictors and class labels

Since all AS are defined at package level, but they are not all defined at class level, we chose to work at package level: hence each data point in our dataset will refer to a package. In particular, since we detect AS on several versions (commits) of a project, we will have a data point for each (version, package) combination. For each data point, we represent each of the four AS as a binary feature, with value 1 if the package is involved in an architectural smell in that version, 0 otherwise. Especially for Cyclic Dependency we also distinguish between `tiny`, `star`, `chain`, `circle`, `clique`: binary features, with value 1 if the package is involved in a cyclic dependency of the respective type in that version, 0 otherwise. AS features were used also as class labels i.e. as target of the predictions. In brief, we used the information about AS in past versions to predict AS labels in the next version.

C. Representing architectural smells in history

To be able to predict AS in future versions using the AS detected in past versions, we have to find a suitable representation enabling the application of prediction models on these data. We choose to use *lags* to represent past data, i.e. we associate past data to current data by adding features to the same data point to represent both current data and the value of the predictors in the past at different times. For example, if we are dealing with project packages, for each package in each version (commit) we will have some features representing AS. If we want to use the past 12 versions ("lags") to predict the presence of AS in the next version, we will add to each row (package, version) the AS features of (package, version - 1), (package, version - 2), . . . , (package, version - 12). In this way, every data point contains AS information about the current version and the 12 previous versions of the same package.

[1] https://drive.google.com/open?id=1wfi4AJr7DC-Hjv0zw_whuhgZDqztjrxJ

[2] https://github.com/

Since this procedure is applied to all available versions, lags create a "sliding window" where for each version we represent also the information regarding a fixed amount of past versions. In the training phase, learning models use lagged information to model the current information. In the test phase, when using the models to predict new AS, models will use the current and lagged data as input. This representation is common when dealing with multivariate time series and has the advantage that after this pre-processing phase, where lags are created, the remaining part of the study can be carried out as a standard supervised learning task. We started from a representation of the AS on each version (commit) of the analyzed project, then we generated a variant of the dataset by aggregating by month. In this way we can synthesize information and look at a large timespan, i.e. more information in the past to predict further information in the future.

D. Machine learning models and Performance estimation

We selected the following machine learning models for our study, using R implementations: Naïve Bayes[3] (NB), Decision Trees[4] (C5.0), Random Forests[5] (RF) and Support Vector Machines[6] (SVMs), of which we tested two particular types provided by the R package: *svmRadial* (SR) and *svmLinear* (SL).

For the estimation of the performance of our models, we rely on a standard repeated k-fold cross validation procedure, with 10 repetitions and 10 folds. Therefore, each model is evaluated 100 times on different portions of the dataset. We rely on the `caret`[7] R package to perform all pre-processing, learning and performance evaluation tasks. As performance indexes, we use the typical indexes found in machine learning and information retrieval in the case of supervised learning: Accuracy and F-Measure. We computed also other metrics, as Precision and Recall, available at the replication package [8]. We consider the above indexes as 'positive' if the AS is present on a package, i.e. its respective feature is 1. We apply two pre-processing transformations to the dataset before using it for learning. First, we remove features with variance near to zero (using the `nearZeroVar()` function in `caret`), mostly addressing columns having only one constant value. Then we apply SMOTE [18] sampling to balance the positive and negative class. This technique down-samples the largest class and synthesizes new data points in the smallest class. Both operations are carried out with the default settings of the `caret` package. We have not used SMOTE in the entire dataset (including the test set), but only in the training set.

To support an informed comparison of the different machine learning models performance, we apply the Wilcoxon signed rank test [19] to the F-Measure values obtained on the 10 folds of the cross validation procedure. Only performances of different models on the same dataset (system and pre-processing setup) are compared. Since we compare all the models of each group, we apply a p-value correction using Holm's method [20]. In the discussion of the prediction results (see Section V), the test results[9] are used to understand if the difference in the average F-Measure is significant, i.e. if the performance of the different models is distinguishable. We consider a significance of $\alpha < 0.05$.

[3]https://cran.r-project.org/package=klaR

[4]https://cran.r-project.org/package=C50

[5]https://cran.r-project.org/package=randomForest

[6]https://cran.r-project.org/package=kernlab

[7]https://cran.r-project.org/package=caret

[8]https://drive.google.com/open?id=1wfi4AJr7DC-Hjv0zw_whuhgZDqztjrxJ

Table II: Accuracy and F-measure results of ML models

		Accuracy					F - Measure				
	AS	C5.0	NB	RF	SL	SR	C5.0	NB	RF	SL	SR
Commons-Math	CD	**.995**	**.981**	**.995**	**.858**	**.823**	**.997**	**.989**	**.997**	.537	.421
	CD-chain	**.837**	**.624**	**.831**	**.765**	**.732**	**.788**	**.643**	**.783**	**.804**	**.776**
	CD-clique	**.945**	**.764**	**.942**	**.928**	**.920**	**.767**	.384	**.762**	**.960**	**.956**
	CD-star	**.897**	**.610**	**.894**	**.922**	**.922**	.553	.269	.557	**.958**	**.958**
	CD-tiny	**.845**	**.666**	**.846**	**.665**	**.676**	**.816**	**.694**	**.815**	**.706**	**.716**
	UD	**.669**	.504	.597	.571	**.699**	.039	.419	.221	**.685**	**.779**
	ICPD	**.711**	**.768**	**.712**	**.887**	**.862**	.235	.211	.255	**.938**	**.924**
JGit	CD	**.996**	**.986**	**.996**	**.999**	**.999**	**.998**	**.992**	**.998**	**.999**	**.999**
	CD-chain	**.833**	**.665**	**.846**	**.999**	**.993**	**.757**	.437	**.775**	**.999**	**.990**
	CD-circle	**.765**	**.630**	**.782**	**.999**	**.995**	**.756**	**.713**	**.769**	**.999**	**.994**
	CD-clique	**.953**	**.799**	**.955**	**.999**	**.997**	**.849**	.500	**.855**	**.999**	**.990**
	CD-star	**.785**	**.646**	**.800**	**.999**	**.994**	**.683**	.443	**.726**	**.999**	**.992**
	HL	**.684**	**.747**	**.753**	**.999**	**.985**	.175	.211	.121	**.999**	**.827**
	UD	.585	**.638**	**.732**	**.999**	**.990**	.233	.255	.132	**.999**	**.959**
	ICPD	**.671**	.590	**.710**	**.999**	**.981**	.252	.305	.232	**.999**	**.950**
JUnit	CD	**.998**	**.896**	**.999**	**.999**	**.958**	**.998**	**.871**	**.999**	**.999**	**.977**
	CD-chain	**.998**	**.899**	**.998**	**.999**	**.847**	**.998**	**.884**	**.998**	**.999**	**.875**
	CD-star	**.994**	**.753**	**.993**	**.999**	**.903**	**.995**	**.793**	**.993**	**.999**	**.622**
	CD-tiny	**.997**	**.884**	**.997**	**.999**	**.857**	**.997**	**.893**	**.997**	**.999**	**.724**
	UD	**.939**	.587	**.937**	**.999**	**.823**	**.943**	**.715**	**.941**	**.999**	.524
	ICPD	**.997**	**.959**	**.997**	**.999**	**.874**	**.996**	**.954**	**.997**	**.999**	**.898**
Tomcat	CD	**.999**	**.977**	**.999**	**.999**	**.999**	**.999**	**.986**	**.999**	**.999**	**.999**
	CD-chain	**.996**	**.913**	**.997**	**.999**	**.998**	**.997**	**.933**	**.998**	**.999**	**.999**
	CD-clique	**.996**	**.793**	**.996**	**.999**	**.999**	**.982**	.409	**.982**	**.999**	**.995**
	CD-star	**.989**	**.760**	**.986**	**.999**	**.995**	**.98**	**.666**	**.976**	**.999**	**.990**
	ICPD	**.750**	**.620**	**.723**	**.999**	**.996**	.292	.269	.289	**.999**	**.978**

V. Results on prediction

To answer our RQ, we analyzed if we can use the presence of architectural smells in the history of the projects to predict the presence of AS in the future. SVM models performed well, hence AS evolution could be used for the prediction of AS in the future. We report the prediction results and the most important rules that were extracted using the data.

Table II reports computed F-measure and Accuracy metrics of all the models analyzed on the four projects: the values higher or equal to 0.6 are highlighted in bold, indicating the best performances. The table reports prediction performance of five models for the next-*month* setup: we used the preceding 12 lags to predict the next one, e.g. we used the features of the last 12 months to predict the AS in the next month. Moreover, 12 lags represent available features of the year before the considered commit. We also experimented with periods of 12 days and 12 weeks before the considered commit but we discarded these options and chose to consider months, since architectural problems tend to affect the project for a longer period of time. The first column of the table reports the different target AS, where for the CD smell we have considered all the different shapes individually (chain, clique,

[9]https://drive.google.com/file/d/0B40F7lXuMsRaajRQR0x5SXVzbE0/view?usp=sharing

circle, star, tiny) and globally (CD row); the second and the third column reports namely the value of Accuracy and F-measure achieved with the corresponding model, decision tree (C5.0), Naïve Bayes (NB), Random Forest (RF), Support Vector Machine Linear (SL) and Support Vector Machine Radial (SR) (see Section IV-D).

The reported performance is high with very few exceptions. The highest performance is obtained in the prediction of Cyclic Dependency (CD) smell and its shapes. This result could be influenced by the class imbalance: accuracy values for *NB* predictions are much lower than the others (and greater than F-Measure) for all the projects. This means that where class imbalance is higher, classifiers chose the largest class (absence of AS). As for the classifiers, the best ones are SVM Linear and SVM Radial: this conclusion is supported by the Wilcoxon test that we conducted and explained in Section IV-D. The test shows that p-values of SL and SR models are significant (p-values between 0 and 0.03) for the majority of the tests. C5.0 and Random Forests are also good (value > 0.9 in average) and have comparable performance among each other.

We analyzed rules computed by the JRip algorithm available in Weka accessed through R using the RWeka[10] package, because we experienced that the rules extracted by JRip are more understandable than the ones reported by C5.0, while the latter can have higher prediction performance. We used the rules to investigate which conditions can lead to the creation of architectural smells. We also simplified the training dataset, representing only the presence or absence of architectural smells in lagged data (0 absence and 1 presence). We discovered that the main high-level rule is that the presence (or absence) of an architectural smell in the past is confirmed in the future. This can be justified by the fact that, architectural smells are large in granularity and can have a slow evolution.

We now report the most important rules that were extracted using the data. The presence of Cyclic Dependency smells indicates a possible presence in the future of this architectural smell and this holds also for the absence (having high accuracy and precision): if the cycle is not present in the history, it is unlikely to appear in the future (97% of the time as Table III shows). This is true for all the systems and for all the shape types of Cyclic Dependency smells.

For example, rule $(CD1 = 0) \Rightarrow CD = 0$ was extracted from the analysis of JGit (with 97.6% correctly classified instances). While the rule does not indicate the way CD is introduced or removed, it shows that the absence of CD in the last month indicates its absence in the future. What we found in JGit is confirmed also for all the other projects and for all the shapes of CD. Table III shows the rule performance for all the projects and indicates the total number of instances of CD per project. Tomcat is the only project where the rule is always true, but the other projects achieved good results with correct classification percentage over 97%. This has some practical implications: since the introduction of CD is rare, paying attention in the early phases of the construction of a system or module could decrease the possibility of incurring CD during evolution. Obviously, since the rule does not have 100% confidence on all projects, developers have to proactively avoid the introduction of CD. As future work, it will be interesting to analyze the reasons why this AS has been introduced in the analyzed projects, i.e. the 3% of the cases where the rule fails. Concerning Hub-Like, Unstable Dependency and Implicit Cross Package Dependency smell, their rules were not extracted or were not significantly relevant.

[10]https://cran.r-project.org/package=RWeka

Table III: Prediction performance rule of CD Smell

Project	Instances classified by $(CD1 = 0) \Rightarrow CD = 0$ Right	Wrong	Right(%)	Total number of instances
Commons Math	516	14	97.4%	3288
JGit	332	8	97.6%	2989
JUnit4	690	20	97.2%	6057
Tomcat	791	0	100%	5306

VI. Threats to validity

Threats to *construct validity* of our study may arise from the representativeness of the measures we applied. In our analysis, we used datasets where data points have been aggregated. In fact, we consider data representing the projects' characteristics on a monthly basis, but the raw extracted data refers to single commits. Since we applied standard aggregation approaches, i.e. aggregating measures representing counts using the sum and aggregating ratios using their mean, the aggregation may have biased the dataset composition. In fact, since the original data refers to single commits, it is not directly tied to a time interval (such as hours, days, and weeks). By aggregating the measures taken at irregular time intervals, we may suffer from distortions or "masking effects"; for example when the same portion of code has been modified many times between two points of interests (months in our case), we count them differently than if we observed the state of the repository on a monthly basis. We mitigate these possible issues by choosing aggregation strategies keeping (as much as possible) the original meaning of each measure. Moreover, the possible models' overfitting has been manually studied by tuning different values of the k-fold cross validation parameters (i.e. we used different k values: 3, 5 and 10) and checking the root-mean-square error (RMSE) obtained. We can observe that machine learning can be a viable solution to apply customized prediction models which adapt on project-specific features by relying on past data.

Further threats to *construct validity* can occur due to errors in the data extraction and preparation phases and the accuracy of the AS detection tool. We rely on a tool (Arcan) to extract dependency metrics and detect architectural smells in the analyzed projects. The tool could be subjected to a systematic bias in the detection. However, a validation of Arcan results has been performed on two industrial projects [4], and on 10 open source projects [15]. Furthermore, since we are predicting the output of the tool in the future, this systematic bias, if existing, has been incorporated into the learning models. Before using

the data extracted by the tool to perform our study, we carefully checked its output on many different projects to optimize the settings of the detection rules and to verify the correctness of the tool output according to the defined rules.

Threats to *reliability* are partially mitigated by an elaborate replication package and the fact that the tool is available and can be applied to any compiled Java project.

Threats to *external validity* may arise from the number of projects used in our study. The results obtained on the four projects may not be replicable on other projects, in particular if they do not rely on the same technology or belong to different domains. Moreover, we only analyzed projects written in Java. Another factor that may have affected our analysis and any empirical study working on the evolution of software systems, is the set of practices applied by the teams that develop the analyzed projects.

VII. Conclusions and Future Developments

In this paper we applied machine learning techniques to predict AS based on historical information to answer one Research Question: *How well does the presence of architectural smells in the project's history support the prediction of architectural smells in the future?*

From the results obtained in the four analyzed projects, the prediction performance is high or very high. Hence, historical architectural smell information can be used to predict the presence of AS in the future. In particular, we found that CD is a good predictor of CD smells. This can be useful to developers/maintainers: if they pay particular attention to this AS and try to remove it as soon as the AS is introduced and detected, they could reduce the possibility of incurring further CD in the future.

Thus, our study encourages researchers to take AS seriously into account when studying the underlying dynamics of software evolution and stimulate developers to monitor software quality through an AS detection tool during their development and maintenance activities.

In the future, we would like to extend this study by analyzing more projects, also in other domains, to consider other architectural smells and extend the analysis to a larger set of lag settings, in order to have a more precise view of the prediction performance further in time. We are particularly interested in understanding and studying the co-evolution of architectural smells. This is particularly useful to better understand the AS, their evolution and also their removal. Through architectural refactoring and software re-modularization we would like also to analyze the impact of refactoring on system complexity, on other software quality metrics and non-functional properties (e.g., performance, security, reliability) [21]. Moreover, we aim to extend the study according to the prediction of changes through AS history, to check if the presence of architectural smells in the projects evolution can be used to predict software changes and vice-versa. We could also study the evolution of the AS and check if a specific AS that tends to be present in the history of a project is more critical with respect to the ones already removed, according to some AS criticality evaluation.

References

[1] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, Apr. 2006.

[2] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *CSMR 2009*. IEEE, 2009, pp. 255–258.

[3] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *Proc. 28th IEEE Int'l Conf. Software Maintenance (ICSM 2012)*.

[4] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. A. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A tool for architectural smells detection," in *Int'l Conf. Software Architecture (ICSA 2017) Workshops*.

[5] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *2013 IEEE Int'l Conf. Software Maintenance*, Sept 2013.

[6] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *2016 IEEE Int.Conf. Soft. Maint. and Evol. (ICSME)*, Oct 2016, pp. 244–255.

[7] ——, "Toward a smell-aware bug prediction model," *IEEE Trans. Software Eng.*, vol. 45, no. 2, pp. 194–218, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2017.2770122

[8] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques," in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, May 2011, pp. 331–336.

[9] T. D. Oyetoyan, J. Falleri, J. Dietrich, and K. Jezek, "Circular dependencies and change-proneness: An empirical study," in *22nd IEEE Int.Conf. Soft. Analysis, Evolut. and Reengineering, SANER 2015, Canada, 2015*.

[10] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, and Y. Cai, "A study on the role of software architecture in the evolution and quality of software," in *Proc. 12th Working Conf. Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 246–257.

[11] J. Díaz-Pace, A. Tommasel, and D. Godoy, "Towards anticipation of architectural smells using link prediction techniques," in *18th IEEE SCAM, 2018*, pp. 62–71.

[12] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 235–245.

[13] R. C. Martin, "Object oriented design quality metrics: An analysis of dependencies," *ROAD*, vol. 2, no. 3, Sept–Oct 1995.

[14] H. A. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin, "On the shape of circular dependencies in java programs," in *Proc. 23rd Australian Soft. Eng. Conf. (ASWEC 2014)*. Sydney, Australia: IEEE.

[15] F. Arcelli Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *Proc. 32nd Intern. Conf. on Software Maintenance and Evolution (ICSME 2016)*. Raleigh, North Carolina, USA: IEEE, Oct. 2016.

[16] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Technical Debt track*. Prague, Czech Republic: IEEE, August 2018.

[17] A. Martini, F. Arcelli Fontana, A. Biaggi, and R. Roveda, "Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company," in *Proc. of the European Conf. on Software Architecture (ECSA)*. Madrid, Spain: Springer, Sep. 2018.

[18] K. W. Bowyer, N. V. Chawla, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *CoRR*, vol. abs/1106.1813, 2011.

[19] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods*, 2nd ed. New York: John Wiley & Sons, Aug. 1999.

[20] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[21] C. Trubiani, A. Ghabi, and A. Egyed, "Exploiting traceability uncertainty between software architectural models and extra-functional results," *Journal of Systems and Software*, vol. 125, pp. 15 – 34, 2017.